A Distributed Systems Infrastructure for Open Public Display Research Networks

by

Oliver Dirk Storz

Diplom-Informatiker

(Universität Karlsruhe (TH))

Computing Department

Lancaster University

United Kingdom

Submitted for the degree of Doctor of Philosophy (PhD)

September 2008

Abstract

A Distributed Systems Infrastructure for Open Public Display Research Networks

Oliver Dirk Storz

Computing Department

Lancaster University

UK

Submitted for the degree of Doctor of Philosophy

September 2008

Fuelled by a sharp decline in the cost of large-screen display hardware, we are currently witnessing a continuous increase in the number of public displays being deployed in locations, such as airports, railway stations and shopping malls. These commercial deployments range from single displays in shop windows, to hundreds of displays in airports and train stations.

In parallel, public display systems have provided researchers with a rich topic for research in a range of disciplines, including HCI, CSCW and sociology. However, once deployed, public displays represent a publicly visible "face" of the corporation or entity behind the deployment and the introduction of experimental, research-led content is therefore often undesirable. With advertising being a driving factor behind the deployment of many displays, stake holders are also

less likely to give up valuable "airtime" to experimentation. Researchers willing to overcome these issues by deploying their own hardware, e.g. in a lab or campus environment, discover that the software packages used to drive commercial displays are tailored towards the needs of commercial deployments and are unable to meet the requirements for experimentation and research.

As a result, researchers using public displays as a vehicle for their investigations normally create small-scale, closed deployments in lab or office environments aimed at investigating specific research questions. Applications and content are usually hand-crafted to best support the investigations at hand, and deployments are rather short-lived. While other research communities, e.g. in the context of PlanetLab or the Grid, have identified the benefits of creating open, shared, medium or large-scale research infrastructures, a similar move has so far not taken place in the context of public display networks.

This thesis aims to provide the foundations for the creation of medium-scale, long-lived public display networks for experimentation and research. The thesis presents a distributed systems infrastructure and API for providing experimenters with the means for creating and running interactive experiments on public display networks. Moreover, it introduces a computational model that provides key abstractions over the hardware and software entities present in a public display network.

The provided API is split into two parts: a low-level API, enabling experimenters to directly control the life-cycle and visibility of experimental content on displays, and a high-level API and scheduling service for scheduling experiments based on constraints, such as time, location, and the presence of Bluetooth devices (e.g. mobile phones). The low-level API features transactional semantics that provide atomicity, consistency and visual isolation for orchestrating experiments involving multiple displays. The constraint-based scheduling interface of the high-level API builds on and complements the low-level API, and provides experimenters with an easy-to-use interface for running and displaying content.

The software infrastructure presented in this thesis has been implemented and deployed as part of the e-Campus public display network at Lancaster University. The software infrastructure is used on a daily basis for scheduling a mixture of research-led experimental content and informational content. We report on both a quantitative and qualitative evaluation of the system based on performance measurements and a study of longitudinal use.

Acknowledgements

First of all I would like to thank my supervisors Prof. Nigel Davies and Dr. Adrian Friday for their continuing support and patience with me over the past years. I first met Prof. Davies in 2000 during my internship at Sony Electronics and was immediately captivated by his approach to supervision: he always had time for me, no matter how busy things got. Moreover, from the start he did not merely "supervise" my work, but also shared his experiences and insights that he had gained over the years and therefore enabled me to catch a glimpse behind the facade of academic research that I had witnessed so far as a student. When I arrived in Lancaster in 2002 I was pleasantly surprised to find a similarly open, accessible and pragmatic style of supervision being used by Dr. Friday. In the years that followed, both my supervisors always made me feel as if I was working "with" them, instead of "under" them, for which I am deeply grateful. It is needless to say that this thesis owes a lot to the many discussions we have had and the constructive feedback they have provided me with over the years based on their enormous wealth of knowledge and experience.

Moreover, I would like to thank Prof. Jochen Seitz, not only for his encouragement and support during the time that I worked for him as a student researcher, but also for establishing the contacts that led to my internship at Sony without which I would almost certainly not have ended up in Lancaster.

I owe a great debt to Dr. Marc Bechler who not only acted as an excellent supervisor of my Diplomarbeit (master's thesis), but who also encouraged me to pursue a PhD and therefore laid the foundations for my time here at Lancaster University.

Many thanks to all those people who have contributed to the e-Campus project. In particular I would like to thank Dave Molyneaux for allowing me to use his excellent photos of the Underpass

opening night, Stephanie Sims for her photo of the display in the Great Hall, John Hardy, Ben Sherratt, Aaron Gregory and Sabine Nowak for the time and effort they invested into the creation of the trial applications, Prof. Nigel Davies for the initial designs of both the poster system and the channels system, Dr. Christos Efstratiou for implementing the poster system and important parts of the channels system, and André Hesse for his involvement in the initial design of the channels system and for his excellent graphical design work in the context of the Map application and the Bluetooth friendly name application. Moreover, I would like to thank Craig Morrall who implemented important parts of the probes that were deployed at WMCSA 2004 and the Brewery Arts Centre and helped with their deployment, Dr. Joe Finney and Dr. Andrew Scott for providing us with network connectivity at WMCSA 2004 and the Brewery Arts Centre, Matthias Joest and Karsten Renhak for their help with the Brewery deployment, and Peter Newman for his work on the Bluetooth friendly name application.

My thanks also go to the various colleagues whom I have shared offices with over the years here at Lancaster University, who have helped to create a pleasant and productive work environment, and with some of whom I have also enjoyed some very pleasant and memorable evenings at the pub.

I would like to express my gratitude for the financial support that I have received in the form of employment on a number of research projects: the EPSRC-funded projects “Grid-Based Medical Devices for Everyday Health” (GR/R85877/0) and “Equator” (GR/N15986/01) , and the EU-funded projects “Simplicity” (IST-2004-507558) and “Simple Mobile Services” (IST-2006-034620).

Finally, my greatest thanks of all go to my parents for their unfaltering love and support. They were always there for me, and this is something that I will never forget.

Declaration

This thesis has been written by myself. The design of the low-level software infrastructure and API, and the high-level scheduling service and API were the result of discussions with my supervisors Prof. Nigel Davies and Dr. Adrian Friday. Aspects of the computational model were first described in an email by Prof. Davies following one such discussion. Initial implementations of the client-side API and Display entities were created by Dr. Adrian Friday. The author later extended the client-side API and re-implemented the Display processes. Parts of the initial probes deployed at WMCSA 2004 and at the Brewery Arts Centre (described in chapter 3) were implemented by one of our former PhD students.

The software infrastructure described in this thesis was developed as part of the e-Campus project that was funded by the Higher Education Funding Council for England through the Science Research Investment Fund.

During the course of this research the author was also supported by the EPSRC under grant references GR/R85877/0 ("Grid-Based Medical Devices for Everyday Health") and GR/N15986/01 ("Equator"), and the European Union under grant references IST-2004-507558 ("Simplicity") and IST-2006-034620 ("Simple Mobile Services").

The work reported in this thesis has not been previously submitted for a degree in this or any other form. Some of the work reported in this thesis has previously appeared in a different form in published papers and articles [SFD06a, SFD+06b] of which I was the principal author.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Over the past decade we have witnessed a noticeable increase in the number of displays that have been deployed into public and semi-public spaces. Despite recent research results that suggest that members of the public largely ignore the content shown on public displays [Hua07], displays for advertising and entertainment purposes have become commonplace in airports and have started to appear in train stations and even on trains themselves. For example, a total of 240 flat-panel displays providing a mixture of advertisements, information and entertainment have been installed in tram carriages in the city of Graz in Austria [INF08b], and more than 160 back-projected displays have recently been installed above check-in desks at Manchester Airport's Terminals 1 and 2 [Pas08].

Besides commercial deployments we have also witnessed deployments of public displays for non-commercial purposes. The BBC, for example, has deployed large outdoor screens in central locations in eight major cities in the UK that are used to display a mixture of non-commercial content, including news, selected BBC feeds and artistic content [BBC08]. Artistic communities have discovered large displays and even whole building fronts as outlets for their work [Urb08].

Moreover, over the past two decades, public and semi-public display systems have provided the research community with a rich platform for experimentation and research in a wide range of

disciplines, including HCI, CSCW, sociology and electrical engineering (see chapter 2). Research using displays can be found as early as the 1980s when researchers at Xerox PARC and Bellcore independently started to experiment with displays and full-duplex audio/video connections to create virtual windows between physically distant spaces [GA86, BHI93, FKC90]. In the following decades display systems were, for example, created that provided new forms of visualising information (e.g. the Dangling String [WB96]), that allowed users to share information with their colleagues (e.g. the Learning Community Newspaper [HBL98]), that acted as catalysts for social interactions (e.g. the Opinionizer [BR03]), or that simply provided new forms of entertainment (e.g. Fancy a Schmink [RLHS04]). Public displays were used by researchers as testbeds for experimentation in a wide range of areas, including user interface design (e.g. Dynamo [IBR+03]), interaction techniques (e.g. Ballagas et al. [BRSB04]), novel display hardware (e.g. the Interactive Fog Screen [RDO+05]) or systems support for public displays (e.g. Stahl et al. [SSKB05]).

1.2 A Case for Creating Shared Deployments of Public Displays as an Infrastructure for Research

Deploying technology for research purposes into public spaces is both costly and time-intensive, as we ourselves learned from first-hand experience when we deployed a series of public display systems as technology probes into various public environments [SFD+06b]. These costs include expenses for display hardware – although these costs have been coming down significantly over the past decade – and for the installation of that hardware. For example, prior to installation the owners of the selected deployment locations have to be liaised with, and compliance with health and safety regulations needs to be ensured. Once installed, deployments have to be maintained on a permanent basis, involving both routine tasks (e.g. updating software packages as vulnerabilities are uncovered or running regular back-ups) and unexpected problems (e.g. power cuts, network outages, failures of hardware components, theft of display hardware, or bugs in the display software). The public nature of such deployments also means that many problems that occur are instantaneously visible to members of the public, and therefore have to be addressed more quickly than in the case of lab-based experiments. These problems may be aggravated even further if the nature of the research demands a larger number of displays to be deployed.

As a result of these difficulties, deployments of public display systems for research purposes are often – as we will show in our survey in chapter 2 of this thesis – rather short-lived and small in scale, with around half of the surveyed pieces of work being based on deployments of single displays. Moreover, with few exceptions, deployments of public displays for research purposes are typically not re-used for additional display-based research. Deployments are typically created with a specific research agenda in mind, and once researchers have followed through with this agenda, the displays involved are dismantled again.

However, the problem of deploying infrastructures for research and maintaining them over a longer period of time is not new. Researchers from other research communities have faced similar problems in the past, and this has, for example, led to the development of PlanetLab [BBC+04, Pla08b] and the vision of Grid computing [FK98]. In both cases, the problems of feasibility and cost have been addressed by creating long-lasting shared infrastructures for research that can be re-used by peer researchers over and over again.

PlanetLab [BBC+04, Pla08b] is a global-scale research infrastructure for research and experimentation in the field of wide-area computer communications and networking. The infrastructure currently comprises around 800 independent networked computers around the globe that have been opened up by individual and organisational members to other researchers in the field. By making nodes from all over the world available, PlanetLab members enable researchers to carry out highly distributed, wide-area experiments under real-world conditions across the Internet that would otherwise be very difficult and costly to undertake. Researchers using PlanetLab are able to deploy and execute research-related code on a subset of these PlanetLab nodes. Each node runs an adapted version of the GNU/Linux operating system. Through the use of virtualisation technology multiple independent experiments can be executed on each node. Each experiment is allocated a "slice" (corresponding to a virtual machine) of the node's resources, such as CPU cycles, network bandwidth, memory and hard disk space. As a result, PlanetLab is able to provide each experiment with reasonable levels of isolation against the effects of other experiments that are executed on the same node. Moreover, quality of service guarantees can be made to individual experiments, should they be required. PlanetLab provides central administration tools, e.g. for node monitoring and managing the life-cycle of slices.

Another development aimed at creating shared research infrastructures can be witnessed in the context of Grid computing. The vision of Grid computing is about facilitating the sharing of

computational resources (e.g. processing cycles and data storage facilities) across organisational boundaries. In analogy with the electrical power grid, one of the visions of Grid computing was to provide researchers with access to computational resources whenever and wherever they needed it, independent of whether they had expensive supercomputing facilities available at their local site. Moreover, the sharing of facilities would also improve the utilisation of existing supercomputing facilities, as sites would be able to share these facilities with other users around the globe whenever they were not needed locally.

The vision also foresaw the construction of Grids using a heterogeneous set of underlying computational resources. The use of abstractions was to ensure that, as far as users were concerned, executing calculations on a supercomputer would ideally be no different from executing the same calculations on a farm of PCs. Grid middleware was to provide these abstractions. Globus [Fos06], for example, one of the most widely used Grid middleware platforms in research, provides standardised interfaces and services for functions such as resource discovery, resource monitoring, job execution, data management, and security.

Despite the encouraging levels of success that PlanetLab and Grid computing have had in their own research communities, we have not yet seen similar developments in the context of research into public displays. However, sharing and re-use have the potential to provide benefits to researchers in this area:

extended duration. If public display deployments are jointly used by a range of different researchers, then these deployments can indeed be funded like other pieces of shared research infrastructure. This means that funding may either be provided directly by the different research groups who have access to the public display deployment, or that funding can be provided by public funding bodies in the same way as these bodies might fund the construction and maintenance of a chemistry lab or a supercomputer centre. As a result, deployments of public display systems can be maintained over extended periods of time, enabling researchers to carry out research of a medium-term to long-term character that would have been more difficult to perform using short-term deployments.

increased scale. Shared, and therefore also increased funding potentially means that more displays can be deployed. Compared with single-display deployments, deployments of increased scale allow for experimentation with novel applications that, for example, span multiple dis-

plays, or collaboratively involve users in front of different displays.

reduced novelty factor. Deployments of experimental technology into public spaces often experience a certain novelty factor. Fleck states that "people's attention is automatically drawn to things which are novel in our environment" [Fle03]. Using short-term deployments of public display applications to perform user studies may therefore return distorted results. The ability to evaluate experimental content over a longer period of time on display hardware that has already been in place for a while can therefore help to reduce these effects and provide more accurate results with respect to longer-term use.

reduced cost of entry. We expect that from the point of view of individual researchers and research groups the costs of using a shared public display infrastructure would be much lower than the investment in display hardware, installation and maintenance that would be required to create a separate public display deployment for each experiment. We hope that this reduced cost of entry will lead to a general growth in public display research and enable more experimental and playful research to take place that would otherwise not have been carried out.

Similar to PlanetLab and the Grid, opening up deployments of public displays for sharing and re-use will require software infrastructure support. Researchers will have to be provided with the means for discovering and acquiring resources, e.g. displays. Interfaces are required that allow researchers to transfer their experimental content onto targeted displays and for making this content visible. Once displays become shared resources it is likely that times exist when several experiments compete for airtime on the same set of displays. Mechanisms for handling contention are therefore required to be in place. Researchers may also, for example, require audit trails providing information about the displays their content was displayed on, the time and date it was shown, and for how long it stayed visible. Moreover, activity in the display network will have to be monitored by its owners, e.g. for the purpose of detecting faults.

1.3 Contributions of this Thesis

In this thesis we present the design, implementation and evaluation of a software infrastructure and API that enables networks of public displays to be used as shared platforms for research.

Specifically we focus on providing researchers with control over the life-cycle and visibility of experimental content on shared public displays, and on enabling them to audit the experiments they carry out. For the purposes of this thesis we define "public displays" as electronic visual displays that are deployed into public or semi-public environments (see [HM03] for a discussion of displays in semi-public environments). The software infrastructure is targeted at supporting medium-scale public display networks of up to a few hundred displays.

The following individual contributions are provided in this thesis:

- *a historic review of research into public display systems and applications* covering over 70 individual pieces of work performed between the early 1980s and the year 2008. The review provides an overview of the types of experiments that public displays have been used for in the past and demonstrates that public and semi-public displays have provided researchers with a rich platform for experimentation and research in a range of disciplines (including HCI, CSCW, sociology and electrical engineering) over the past two decades.

- *an analysis of key properties of the surveyed systems and applications* along four main dimensions: "objectives", "content", "deployment", and "evaluation techniques".

- a *computational model* of a low-level software infrastructure that meets the requirements for supporting the use of public display networks as shared platforms for research. The computational model defines the functional entities in our software infrastructure and enables researchers to reason about the state of displays and content in public display networks. The model allows researchers to develop experimental content and schedulers that control the life-cycle and visibility of this content. Moreover, the computational model provides means for hiding the complexity of non-standard and dynamic hardware setups from experimenters.

- *an extension of the concepts of atomicity and isolation to the visibility of content* that provides the means for controlling multiple content items on different displays as atomic units and ensures that intermittent states of the software infrastructure are not made visible to users observing the displays.

- *an engineering model* that demonstrates an approach for mapping the functional entities of our computational model onto machines and processes in a public display network.

- a *qualitative and quantitative evaluation of the low-level software infrastructure and API*

that demonstrates the ability of the low-level software infrastructure and API to support both experimental and day-to-day content in a shared public display network.

- *the identification of the requirement for a high-level constraint-based scheduling API* based on the experiences gained with the use of our software infrastructure to support research-related content and day-to-day content.

- *a design for a high-level constraint-based scheduling service and API* that is layered on top of the low-level API. The high-level scheduling service and its associated API simplify the use of the software infrastructure for experimentation by providing a secure, platform-independent API to a constraint-based scheduling service that is hosted and managed centrally as part of the overall software infrastructure.

- a *qualitative evaluation of the high-level scheduling service and API* that shows how these were successful in supporting a range of experimental applications. Moreover, the high-level scheduling service serves as a further example for the power and flexibility of the low-level software infrastructure and API.

1.4 The e-Campus Project

The work presented in this thesis was undertaken in the context of the e-Campus project. e-Campus is a network of public displays that were deployed on the campus of Lancaster University in the UK. The display network was funded by the Science Research Investment Fund of the Higher Education Funding Council for England and was designed to serve a dual role: as an infrastructure and testbed for local researchers and artists, and as a device for improving the experience of students, visitors and staff on campus. In line with Weiser's vision of seamlessly embedding technology into our surroundings [Wei91], e-Campus particularly aimed at supporting and encouraging experimental, potentially multi-disciplinary research in the fields of mobile and ubiquitous computing.

Display deployments commenced in the autumn of 2005 with the installation of three projected displays in an underground bus stop on campus (called the Underpass, see figure 1.1). Since then approximately 65 additional displays have been installed in various locations on campus. These include around 40 electronic door displays that have been installed outside offices and lab spaces

Figure 1.1: The e-Campus screens in the Underpass.

in the Computing Department. Besides the projected displays in the underground bus stop, a back-projected display that can be viewed from the outside at night-time was installed above the entrance of the theatre on campus (see figure 1.2). Approximately 25 large 40" LCD displays have been installed so far in a range of locations, e.g. outside lecture theatres (figure 1.3(b)) or in the foyer area of colleges and departmental buildings (figure 1.3(a)). A further 5 displays are scheduled to be installed within the next few months, including two daylight-visible 40" LCD displays for outdoor use that are to be deployed in the main square of the campus, and one outdoor projected display.

In addition to supporting research on top of traditional public display hardware (LCD displays, projectors), e-Campus has also acted as catalyst for the development of non-standard public display systems and hardware. The Firefly project [Fir08] has demonstrated how novel forms of display hardware can be created by embedding intelligence into individual lighting elements and by networking them together. Firefly displays consist of strings of individually addressable and controllable LED-based lighting elements. These string are in appearance not unlike the strings of fairy lights that are typically used to decorate Christmas trees. Like fairy lights, Firefly strings can be wrapped around objects in the environment, pinned onto walls or hung from objects, allowing users to create large 3-dimensional displays that use individual Firefly lighting elements as pixels. Once deployed, cameras are used to auto-detect the topology and geometry of the deployed Firefly

Figure 1.2: Back-projected e-Campus display at the Nuffield Theatre. (Photo: Stephanie Sims)

(a) e-Campus display in the foyer of InfoLab21.

(b) e-Campus displays outside the Faraday lecture theatres.

Figure 1.3: Examples of e-Campus displays.

lighting elements. The resulting model is used as input to an API that allows developers to display content on the Firefly display.

The e-Campus project is jointly managed by Lancaster University's Computing Department, Information Systems Services, and Estate Management.

1.5 Roadmap of this Thesis

The remainder of this thesis is structured as follows. In chapter 2 we provide a survey of the use of public display systems and applications in research, based on an historical review of work in this area. The results of this survey are used to analyse key properties of the surveyed systems. Following this analysis we present an overview of public display and digital signage products and projects for commercial and non-research use.

In chapter 3 we focus on the requirements for our software infrastructure. The requirements presented are drawn from multiple sources, including the analysis of existing public display systems and applications for research that is presented in chapter 2.

The consolidated set of requirements presented in chapter 3 is then used as the foundation for the design of the computational model underpinning our low-level distributed systems infrastructure, and of an accompanying API, both of which are described in chapter 4.

The chapter is followed by a description of the engineering of the computational model and its mapping onto machines and processes in chapter 5.

We then present an overview of the implementation status of the low-level software infrastructure and a qualitative and quantitative evaluation of the implemented system in chapter 6. The qualitative evaluation is based on our experiences with supporting research-related and community-provided content using our software infrastructure and API. Performance measurements complement the qualitative analysis to provide quantitative data about the implementation presented in chapter 5. Both qualitative and quantitative results are analysed with respect to the requirements outlined in chapter 3.

The evaluation also identifies a number of shortcomings of our low-level software infrastructure

that became apparent while using the system to support a set of experimental and day-to-day content. As a result, chapter 7 describes the design, implementation and evaluation of a high-level scheduling service and API that addresses these shortcomings.

The thesis closes with a summary of the presented work, a discussion of the main contributions, and an overview of planned future work (chapter 8).

1.6 Summary

In this chapter we have presented a case for creating deployments of public displays that are treated as infrastructures for research and are therefore opened up to and shared with other researchers. We have presented examples from other fields that have created such shared research infrastructures, namely PlanetLab and computational Grids, and identified benefits that the sharing and re-use of public display deployments may provide to researchers. We have argued that the creation of shared public display networks as platforms for experimentation will require support in the form of an underlying software infrastructure that will handle, for example, the contention for display resources. In this thesis we present such a software infrastructure that is aimed at supporting a wide range of experimental applications on public display networks by enabling researchers to control the presentation of content in such networks. We have outlined the key contributions that are made by this thesis and closed with an overview of the structure of this thesis.

Chapter 2

Existing Public Display Systems

2.1 Introduction

Public display systems have a rich history dating back to 1928 when one of the world's first public displays – the Motogram [New97, Sig08a, Sig08b] – was installed in New York's Times Square. In this chapter we present a summary and analysis of past and present public display systems. We start by presenting an historical review of public display systems in research from the early 1980s up to the present time. The review is followed by an analysis of the key properties of the surveyed systems. We close the chapter with an overview of public displays systems outside the research domain, including digital signage systems and deployments in the commercial and artistic sectors.

2.2 An Historical Review of Research into Public Display Systems

2.2.1 The 1980s

Public display research systems in the 1980s were mainly centred around the notion of virtually connecting physically separate spaces together using audio and video links. The developments during this decade were fuelled by the emergence of portable electronic video cameras in the 1970s

and improvements to the communication infrastructure, such as the expansion of satellite-based communications during the same time.

In November 1980 artists Kit Galloway and Sherrie Rabinowitz created "Hole-in-Space" [GR80]: two large back-projected displays, speakers and cameras were installed in a sidewalk-facing window in the Lincoln Center for the Performing Arts in New York City and in a display window at the "The Broadway" department store in Los Angeles. A satellite link was used to establish a full-duplex audio and video connection, essentially creating a virtual window between the two places. The artists deliberately decided not to provide any instructions or explanations about the purpose of the displays. The display remained active for a total of three evenings.

While "Hole-in-Space" was only short-lived, during the second half of the 1980s researchers began to experiment with the concept of "media spaces" [BHI93] – long-lived audio and video links that were designed to enable casual, informal interactions with distant co-workers. While most of these systems were installed on people's desks in their offices, a small number displays were deployed in semi-public spaces as additions to office-based experiments.

Researchers at Xerox PARC created a full-duplex audio and video connection between commons areas in research offices in Palo Alto, California, and Portland, Oregon [GA86, BHI93]. Cameras and monitors were installed in each location. Video compression equipment and a permanent 56 kbit/s video link were set up between the two sites. Audio was transmitted using regular telephone lines and speakerphones in each of the spaces. The link was kept active for 24 hours a day, seven days a week and was in place from September 1985 until early 1988 when the offices in Portland were finally closed.

A similar system was created by Fish et al. as a medium for informal communication. The "VideoWindow" system [FKC90] used back-projected displays that had dimensions of 3x8 feet to provide near life-size images of remote conversation partners. A 2-D audio system was employed to help users to determine the spatial location of remote human speakers. The system was deployed and trialed for a total of 3 months in 2 common areas on two different floors of the Bellcore labs. An evaluation of the system based on recordings of both camera and audio feeds during this three month period revealed that many conversations using the system were in fact "indistinguishable from similar face-to-face interactions" [FKC90], but that encounters with remote colleagues were less likely to lead to a conversation than encounters that were made directly with co-located

colleagues.

2.2.2 1990 – 1995

In 1991 Mark Weiser published his seminal ideas about 'ubiquitous computing" [Wei91]. Ubiquitous computing envisions a world in which interaction with computational devices is as natural and simple as using a light switch. This is achieved by specialising computational devices – away from the concept of multi-purpose desktop computers – and embedding them into our surroundings.

In his paper "The Computer for the 21st Century" [Wei91] Weiser illustrated his ideas using the examples of display devices that he and his colleagues at Xerox PARC had built. He outlined how displays of different sizes could be embedded into the work environment to assist users with different tasks. Besides portable inch-size and foot-size displays for personal use ("tabs" and "pads"), Weiser described the use of large-scale wall-mounted displays ("boards"/"liveboards") as both collaborative electronic whiteboards and situated community bulletin boards and blackboards, i.e. as situated public or semi-public displays.

According to the paper, boards had already served as platform for research and experimentation: "Others use the boards as testbeds for improved display hardware, new 'chalk' and interactive software" [Wei91]. In addition, boards were networked, for example enabling users to engage in collaborative whiteboard sessions with colleagues in remote locations.

Moreover, Weiser briefly mentioned a prototype public display application serving personalised content to passers-by without requiring explicit user interaction with the system. The "scoreboard" application (unpublished, developed by Marvin Theimer and David Nichols) used an active badge location system to identify users in proximity to boards and adapted the content shown on the display according to their interests.

In their 1995 paper "Designing Calm Technology" [WB96] Weiser and Brown followed up on the theme of specialising and embedding computational devices into people's surroundings. The authors argued for deliberately designing information displays for deployment into the periphery and for making technology calm by empowering users to decide when to bring it to the focus of their attention. The concept of "calm technology" and the example of the "Dangling String" described in the paper later served as inspiration for numerous of ambient display systems all over

the globe. The Dangling String was an "8 foot piece of plastic spaghetti that hung from a small electric motor mounted in the ceiling" in the corner of a hallway at Xerox PARC. The speed at which the motor rotated the string indicated the level of activity on the local area network. The string provided both audible (sound of the motor) and visual (rotation of the string) information to users and seamlessly blended into the environment of the research lab.

2.2.3 1995 - 2000

Possibly influenced by Weiser's ideas, public display research during the second half of the 1990s was characterised by public display systems that delivered information in a situated, peripheral fashion. While some prototypes used monitors to convey information in textual and graphical form, others employed abstract ambient displays that were designed to blend into the surroundings.

The Flexible Ubiquitous Monitor Project

Similar to the *scoreboard* prototype described by Weiser in 1991, the *FLexible Ubiquitous Monitor Project* (FLUMP) [FWD+96] by Finney et al. aimed to provide personalised information to passers-by. The FLUMP infrastructure consisted of regular-size computer monitors and Apple Macintosh LC II workstations. Each workstations ran a web browser that received updated pages via server pushes. Personalisation was achieved by employing an Active Badge system to identify users and by enabling users to create personalised pages using an extension of the HTML format. Users were able to embed special markup tags (similar to tags used in PHP's [PHP08] templating system) into their HTML pages that were expanded by the FLUMP infrastructure during run-time to reflect dynamic information, such as the current time, the number of new and old mail messages, along with sender and subject line for each unread message, upcoming appointments, a cartoon of the day, or the opening status of a local coffee bar. FLUMP displays cycled through a set of default content if no registered users were within range. The system was deployed in the Computing Department at Lancaster University in early February 1996. According to the paper, a single display was deployed in the Department's central staircase. A total of eleven users used the system on a daily basis. The evaluation of the system is based on performance measurements and anecdotal reports.

Wearable Displays and the Community Mirror

Wearable displays are, as the name suggests, displays that can be worn by a person. Content on these displays is typically outward-facing and for consumption by members of the public. Wearable displays can therefore nevertheless be considered as public displays.

"Thinking Tags" [BMMR96, BMRS98] were wearable, electronically enhanced name tags developed by Borovoy et al. in 1996. Like regular name tags they displayed printed information about the name and affiliation of the wearer. However, Thinking Tags were also able to act as catalysts for conversations by indicating how much conversation partners' opinions concurred or differed regarding a certain set of topics. Users primed their Thinking Tags by answering a pre-defined set of multiple choice questions. Users selected specific answers by dipping the tags into buckets corresponding to the different answers. After this initial setup phase, conversation partners received indications about their level of agreement to these questions through a row of bi-colour LEDs mounted on each tag. For each question answered in common, a green LED was lit, for each disagreement a red LED was lit. Thinking Tags exchanged information with each other (and with the initial programming buckets) using infrared communication. Thinking Tags were deployed and trialed during a two-day anniversary event at MIT involving a total of 200 users.

In 1997 Borovoy et al. developed a modified version of their original wearable tags. "Meme Tags" [BMV+98] enabled users to pre-load their wearable tags with memes – short snippets of text representing ideas or opinions. If two wearers encountered each other, each tag selected one of the memes it stored and presented it on its built-in LCD display. Conversation partners could then decide to copy the displayed meme into their own tag by pressing a button. Similar to the original Thinking Tags, memes were transferred using infrared communication. Meme Tags were only able to store a limited number of memes, forcing users to be selective when copying memes and to regularly delete memes that they considered to be less important, overall leading to a Darwinian selection of ideas and opinions. Meme Tags were trialed during a two-day event at MIT in October 1997 that involved around 400 participants comprising sponsors, staff and students. During the event a large-scale public display – the "Community Mirror" – was used to present information about the evolution of memes and the group dynamics at the gathering. Researchers used a combination of system logs, observations and anecdotes to evaluate the system.

The concept of wearable displays was followed-up by Falk, Björk and Ljungstrand in 1999

in their publications about “WearBoy” [LBF99], the “BubbleBadge” [FB99] and “ActiveJewel” [LBF99]. WearBoy represented an attempt of creating a flexible general-purpose hardware platform for wearable public displays by modifying off-the-shelf Nintendo Color GameBoy devices. The displays featured a 2.5” graphics-capable colour screen, 4-channel stereo sound, infrared and serial port communications, a number of buttons for user interaction, all at a reasonably low power consumption of around 0.6 Watt. The units were programmable using a freely available SDK. Exchangeable cartridges were used to load programs into WearBoy/GameBoy devices, facilitating the deployment of different applications to the wearable displays.

BubbleBadge and ActiveJewel were both applications built on top of the WearBoy platform. ActiveJewel represented a piece of wearable digital jewellery in a brooch-like shape, showing changing computer-generated patterns on its display. The BubbleBadge aimed at enhancing face-to-face interactions between users. The display was intended to be worn close to wearer’s face to be easily visible to communication partners. The authors foresaw information displayed on BubbleBadge displays to be provided by three potential sources: by the wearer himself, but also by the environment, or by the viewer or entities acting on behalf of the viewer, in which case the viewer would effectively appropriate the wearer’s display to access information relevant to the viewer. In addition to wearer-supplied information, the prototype system implemented by the authors was capable of displaying information about local activities supplied by the environment and communicated to the device using an add-on radio transceiver. The system was informally evaluated by a number of students.

The Learning Community Newspaper

The desire to improve awareness of events and accomplishments, and to encourage conversations in the Learning Communities Group at Apple Computer Inc. motivated the development of an electronic community news display by Houde et al. in 1998 [HBL98]. The system consisted of a single large-screen projected public display in a communal kitchen in the group’s research labs. Members of the lab were encouraged to send in stories via email. A tool-chain of scripts extracted message subject and body, and inserted the information along with information about the author into an HTML template creating the appearance of a newspaper front page when displayed on the public display. The front-page was designed to simultaneously display all stories that were sent in during the course of a day. In addition to the front-page visible on the public display, a web-based

version offered additional functionality to users accessing the electronic community newspaper using their workstations. The web version included an ”inside” section containing articles that were for lab-internal consumption and should therefore not be visible to visitors, and an archive of all articles submitted to the electronic newspaper.

The electronic newspaper system underwent an iterative design process using two experimental prototypes before reaching the final design described above. The first prototype was based on physical, paper-based artefacts attached to a wall in the kitchen area. A pen-based barcode reader attached to a computer screen next to the wall could be used for retrieving additional information about an artefact. This prototype was abandoned in favour of a fully electronic display as the overhead of creating both appealing physical and electronic content was deemed to be too large.

Houde et al. noted that the public display and its peripheral nature proved to be a vital part of the system. Even though the newspaper system was accessible using the web, users mainly used the public display to access information. They also reported that a version of the system that was deployed with a distributed community of teachers was far less successful than the system in their own lab. The authors believed that this was largely caused by the lack of a public display in a common space. According to Houde et al., looking at the news stories while passing the public display was felt to be “lightweight, fun”.

Moreover, the authors described a conflict between the attempt to keep the process of submitting news stories simple and the desire to gain additional control over their presentation and scheduling, such as the ability to make a news item appear during a specific period of time [HBL98].

Ambient Displays

The Water Lamp, Pinwheels and the ambientROOM: In 1997 and 1998, Wisneski et al. created a set of ambient peripheral displays to promote awareness and understanding: the “Water Lamp” [DWI98] and “Pinwheels” [DWI98, IRF01]. The Water Lamp used a number of solenoids to create ripples on the surface of water in a bowl that was mounted on a stand. Lighting from below the bowl created projections of those ripples onto the ceiling. The authors described the ripples as the effects created by virtual raindrops of “bits from cyberspace” falling into the bowl.

A precursor of the Water Lamp had already been part of Ishii’s ambientROOM. Inspired by

Weiser's vision of calm, peripheral technology, Ishii created a workspace that used physical artefacts as user interfaces in an attempt to bridge the divide between the physical and the digital world. Besides various tangible input devices and a water lamp display used to visualise the activity of loved ones, the room featured illuminated patches projected onto the walls of the ambientROOM to reflect activity in workspaces nearby, and sound-based displays using a set of ambient sounds at variable intensities to convey information.

The second ambient display system described by Wisneski et al., Pinwheels, comprised electric motors attached to regular pin wheels. The speed of the rotation could be used to visualise information. A redesigned version of Pinwheels was later deployed at the NTT-ICC museum in Tokyo during summer 2000. A total of 40 Pinwheels were deployed in an 8x5 array under the ceiling of an exhibition room. The deployment lasted for a total of 17 days. The wheels were used for visualising physical traffic, e.g. cars or people on the streets, and traffic in cyberspace, such as email traffic.

Water Lamp and Pinwheels both included an embedded iRX 2.0 PIC micro controller. The controllers themselves were attached to a regular computer via a serial cable and could be controlled using TCL and Java programs.

Office Plant #1: Office Plant #1 [BM98] was an ambient display created by Bohlen and Mateas in 1998 and featured a mechanical sculpture vaguely resembling a flower. The sculpture consisted of an aluminium bulb that could open and close and was mounted on an extendable stem, and a set of fronds made from piano wire that could be moved individually. Office Plant #1 also featured a built-in speaker that, according to the authors, could be used to make the plant "whistle, chant, sing, moan, complain, and drone". The plant was programmed to display classifications of incoming email messages.

The Information Percolator: The "Information Percolator" [HHT99] comprised an array of 32 vertical tubes filled with water. The display used air bubbles rising from the bottom to the top of tubes to represent pixels. To influence the release of air, each individual tube was equipped with an air pump and a micro controller. The controllers themselves were attached to attached to a networked computer (the "Bubble Server") using a serial cable. The whole display measured approximately 1.4m x 1.2m x 0.2m and was capable of displaying small black and white images

and text.

The Bubble Server exposed a low-level API for content creators that was accessible using Java RMI. The API operated on a queue of bitmap images, each of which would be displayed for a duration specified by the user. Consequently it provided operations for adding new bitmaps to queue, as well as for interrogating and managing the queue itself.

A total of three prototype applications were created for the Information Percolator: a clock-like application that was linked to the display owner's diary and would, besides indicate the passing of time, show certain patterns if scheduled events were imminent, an application that was linked to a camera in a hallway outside the owner's office and used individual tubes to represent activity in a certain parts of the hallway, and a third application displaying the output from a random poetry generator.

Dynamic Photos: "Dynamic Photos" [Gre99], published in very early stages of implementation by Greenberg in 1999 was a picture frame that provided information about the availability of other (possibly geographically distant) people. Moreover, Dynamic Photos could also be used to establish live video connections to these individuals. The main aim of the system was to raise awareness and to enable users to use that awareness for establishing direct interactions with other people. The display consisted of a touch-sensitive CLEO CE tablet PC with dimensions of approximately 16x20 cm and was designed to be hung like a photo frame onto a wall. It showed a group photo in which faces were dynamically distorted to reflect availability for communication: larger faces indicated increased availability. The display also featured ambient audio cues to reflect changes in users' availability. To establish a live video connection to a person in the picture, users could walk up to the display and stroke the photo of that person.

Windows as Display Surfaces: Rodenstein developed an ambient display technology that allowed regular glass windows to be used as information sources [Rod99]. The prototype used a foil (3M Privacy Film) whose opaqueness could be controlled using an electrical current to render parts of a window into a projection surface. The resulting displays was used to provide a short-term weather forecast by projecting typical images and animations of the forecast weather onto the window, essentially providing a preview of the upcoming weather: a frozen window would be used to indicate that frost was forecast, lightning was shown if a thunderstorm was expected to

arrive. The authors foresaw the use of the display not only for providing a glimpse into the future, but also for looking into the past, e.g. enabling users to reason about plant growth during different times of the year.

2.2.4 2000 – 2004

The new millennium saw a noticeable and continuous drop in the cost of flat-panel displays and projectors, coupled with a steady increase in their capabilities. The resulting increased affordability and use of flat-panel displays and projectors over traditional CRT monitors meant that displays could now be installed more easily into public spaces. For example, while it had been very difficult to deploy bulky CRT displays into narrow corridors, flat-panel displays suddenly made such deployments significantly easier. This increased deployability of public displays was certainly a major contributing factor to the growing number of public display research prototypes that were created during this time period.

Other examples of developments in the context of display hardware during this time include the "Interactive FogScreen" and Pinhanez' "Everywhere Displays": In 2001 Pinhanez demonstrated how to combine steerable projection and camera-based gesture recognition to create projectable, steerable touch-enabled user interfaces on common surfaces within our environment [Pin01]. During the following years the technology continued to be improved [PPL+03] as part of the "Everywhere Displays" project at IBM Research and served as the underlying display platform for a number of public display applications, e.g. the use of steerable displays for navigation and product information in a retail environment [SPK+03].

The "Interactive FogScreen" [RDO+05] demonstrated at SIGGRAPH in 2005 created large-screen displays by using artificially generated fog as a projection surface. A unit mounted below the ceiling generated a constant downwards flow of air and fog. A separate projector could then be used to project onto this semitransparent fog curtain. The FogScreen was viewable from both sides and even enabled users to reach through or walk through the screen. Interactivity was realised by tracking laser pointers that users held in their hands while interacting with the display. FogScreen became a commercial product and has been used in museums, live events and trade shows [Fog08].

We have arranged the surveyed systems in this time period under four headings: door displays,

informative art, public display systems for sharing information and for encouraging social interactions, and other systems. While these headings do reflect key properties or objectives of the systems presented, we nevertheless acknowledge that these headings are far from being exclusive, and that certain overlaps between the headings exist. For example, messages left on door displays by their owners may lead to social interactions with passers by, and the encouragement of social interactions might even have been one of the objectives when these displays were deployed.

Door Displays

During the early 2000s, researchers started to look at office doors as potential locations for public situated displays. The aim was to build on existing practices of leaving notes and announcements on and next to these doors.

Dynamic Door Display: Unlike traditional office door notes the "Dynamic Door Display" [NTDM00] aimed to investigate the use of door displays for displaying both public information and private information that would only be available to a specific set of people. The prototype display, based on a handheld computer with a touch-sensitive display, was mounted on a door outside shared lab space. It welcomed visitors with a group photo of the lab members. Selecting an individual member allowed visitors to obtain information about that person's last known location and provided access to a subset of his calendar entries. Visitors could leave voice messages or select from a list of default messages.

Display owners were given control over which calendar entries would be disclosed to visitors. In addition owners were able to personalise information disclosure, enabling selected visitors to access additional information. Visitors using this feature used iButton-based [Max08] authentication to identify themselves towards the system.

Similar door display systems were developed independently by researchers at Accenture Technology Labs and Lancaster University.

Outcast: "Outcast" [MCL01] enabled owners to leave information on the display that was intended to be seen by passers-by and visitors. Available types of information, all web-based and displayed inside a browser on the display, included a short bio, all non-private entries in the owner's

calendar, the owner's current location (sensed using an infrared badge system), information about the owner's projects and demos, and a list of URLs. Visitors were able to leave text-messages using an on-screen keyboard. If not used interactively, the display cycled randomly through the set of content specified by the owner. While the prototype described in the publication did not provide any means for personalising the information on the display to individual viewers, this was described as part of future work.

Hermes: The first "Hermes" [CFDR02] door display was deployed at Lancaster University in spring 2001. It underwent a longitudinal multi-year deployment and user study, and a successor version of the system is still in use today. Like the Dynamic Door Display, Hermes enabled display owners to leave messages for visitors in the form of pictures, text or stylus-based scribblings. Messages could be set using a web interface, the display itself or via text messages sent from mobile phones. Visistors were able to leave messages using an attached stylus. In addition to being notified via email, owners were able to retrieve messages using a web interface that could also be accessed from remote locations, e.g. while being away from the office. A similar system, was later developed by Beale and Jones at the University of Birmingham [BJ04].

The RoomWizard: Unlike the door display systems presented so far, the "RoomWizard" [OPL03] was not designed to provide information about individuals. Instead, RoomWizard aimed at providing booking information outside shared meeting rooms. A total of 5 displays were deployed outside meeting rooms in two buildings of a large multinational petroleum organisation in the UK and were used for more than 4 weeks. The prototypes showed textual information about their current booking status and the individuals who carried out the bookings. Additional light strips on the side – green for "free", red for "reserved" – enabled passers-by to quickly assess the booking status of individual rooms. Bookings could be made using a web interface that was directly hosted by each individual RoomWizard machine. In addition, if multiple displays were present on a local network, the displays exchanged booking information, and consequently booking for any room could be performed through any display's web interface. RoomWizard displays also allowed impromptu bookings to be made by directly interacting with the touch-sensitive displays themselves.

Informative Art

While the concept of using abstract, ambient displays for presenting information had already been introduced during the second half of the 1990s, researchers revisited that theme at the start of the new millennium. Unlike earlier ambient displays that had mainly used physical artefacts as display devices, a part of this new generation of ambient displays used newly available flat-panel displays as output devices. Researchers surrounding Tobias Skog at the Interactive Institute in Sweden used these displays to deliberately explore the boundaries between abstract ambient information displays and pieces or art. "WebAware" [SH00, RSH00] used a large flat-panel display to visualise hits on the group's web pages. Pages were represented as dots, arranged in a circular pattern around the root page of the server. The distance of a dot from the centre represented that page's directory depth on the server, giving the whole display the appearance of "orbits" around a common centre (the root page). When a page was requested, its corresponding dot slowly illuminated to full brightness on the display, after which it slowly faded back to the background colour.

In contrast to "WebAware", which represented an information display that could be seen as a piece of art, Redström, Skog and Hallnäs also re-purposed pieces of art as information displays. "De Stijlistic Dynamics" [RSH00] used a flat-panel display to represent information in the form of a Mondriaan-style painting, for example to represent email traffic or weather conditions in different cities. A later publication [HS03] by Holmquist and Skog described the "Soup Clock", a clock whose appearance resembled Andy Warhol's pop-art depicting Campbell's Tomato Soup cans, "Motion Painting", in which a display/canvas were slowly filled from one side to the other with vertical lines of different colour representing levels of activity, and "Stone Garden", an image of a patch of grass into which stones and of different sizes were placed to represent earthquakes of different magnitudes and the locations at which these had taken place.

Hello.Wall

While the first ambient displays that surfaced during the second half of the 1990s had all been rather limited in size, "Hello.Wall" [PRS+03, SPR+03], demonstrated by Streitz et al. in 2003, on the other hand represented a large wall-size ambient display. The display comprised an array of LED clusters and was able to convey information using light patterns. Users were able to interact with the display using "ViewPorts", PDA-like devices. ViewPorts were equipped with long range

RFID transponders, enabling Hello.Wall to sense users in its vicinity, and with short range RFID readers. These short range readers could be used to select individual LED clusters on Hello.Wall by bringing the ViewPort device close to them (LED clusters were equipped with short range RFID transponders). ViewPort also contained a built-in LCD screen and a number of buttons, enabling the display of more detailed information and more sophisticated interactions to be performed on ViewPort devices. Hello.Wall was used for playing a form of Memory in a conference setting – delegates used ViewPorts to switch individual LED clusters on or off – and to visualise the general atmosphere and activity levels in the labs of one of the project partners.

Virtual Windows Revisited

The new millennium also saw a brief revival of public display systems aiming to provide virtual windows, i.e. audio and video links to physically remote spaces. Unlike the original systems developed in the 1980s which were all based on separate, dedicated video and audio installations, progress in the fields of computer hardware and networks meant that regular computers could now be used to capture and display audio and video streams. In addition these streams could be transported using existing IP-based networks. "The Virtual Kitchen" [JVG+01] employed a set of three projected displays with full duplex audio and video to link together three kitchens in separate buildings at Microsoft Research with the aim of facilitating and encouraging informal interactions between employees. Displays were tiled into four areas: two areas were showing feeds from the other kitchens, one area was used to mirror the local feed, and one area showed a CNN news feed with subtitles in order to attract users. The prototype system was in use for more than four weeks and was evaluated using a combination of observations, informal discussions, email feedback and system logs.

Unlike the systems deployed in the 1980s, the Virtual Kitchen received strong opposition by users who were concerned about having their privacy violated by the use of audio and video capture technologies in the workplace. The authors report that individual users went as far as sabotaging the system. Additional modifications had to be made to the system to address these privacy concerns. Privacy was also a topic of the "Telemurals" system [KD04], developed by Karahalios and Donath in 2003. The system provided full-duplex audio and video connections between two semi-public spaces located in student dormitories. Like the Virtual Kitchen, an initial version of the system delivered both audio and video straight to the other display without

any additional processing. However, later versions of the system processed video feeds using an edge detection algorithm, rendering objects and people into comic-like shapes to address privacy concerns. By default, shapes would additionally appear very faint, making it hard to clearly identify individuals in the picture. If a conversation took place, the system gradually enhanced the pictures rendering shapes in recognisable fashions. Feedback of the locally captured video feed was provided by overlaying and merging the two feeds into a single picture, resulting in similar images being displayed at both sites. Different colours were used to distinguish between local and remote shapes. Additionally, overlapping areas were depicted using a special colour. Audio was delivered unprocessed, but was also fed into a text-to-speech program and displayed as "graffiti". The system was in use for a total of two months and was evaluated using a combination of direct observations of use, observations of the camera feeds, observations of the abstracted video feeds, studio critique, as well as recordings of audio and video feeds.

Raising Awareness and Encouraging Social Interactions

Building on earlier work undertaken by Houde et al. [HBL98] during the late 1990s, a whole range of public display systems and applications were developed that were targeted at raising awareness of activities within social groups and to provide clues for conversations and social interactions between members of a community.

"GroupCast" [MCL01, McC02], developed by McCarthy et al., consisted of a single large flat-panel display deployed in a coffee corner. The system's aim was to create opportunities for conversations between people who would normally not talk with each other. The display was able to identify people in its vicinity using an infrared badge system. The initial system required users to use a web form to specify topics of interest. If people met in front of the display, the application aimed to identify common topics and displayed content that reflected these common interests. However, the authors later integrated GroupCast with a personal display system called "UniCast" [MCL01] that users had already personalised to show content that was of interest to them. The authors acknowledged that users were more likely to spend time personalising something they saw permanently on their desks (UniCast) than something they only passed a few times a day (GroupCast). As a result, the GroupCast application was modified to no longer look for common topics of interest, but to simply display content taken from the UniCast profile of one of the user's lingering in front of the display. The authors argued that although the content displayed by GroupCast

only matched one person's interest, it would still be able to spark conversations.

A second system developed by McCarthy to foster conversations was the "Interactive Wall Map" [McC02]. The idea was that people, when they are in front of a map, generally like to talk about things they have done in certain places. The system consisted of a large wall map of the world (3.96m x 2.64m) into which a total of 6 computer monitors were embedded – two displays for the Americas, two for Europe, Middle East and Africa, and two for the Asia and Pacific region. In addition 24 cities and places had been equipped with LED-topped button switches. If an LED was lit green, content was available for that location. If the LED was lit red, content associated with that location was being displayed on one of the monitors. The Interactive Wall Map was based on a modified version of the UniCast [MCL01] personal display system. Content was geographically indexed and added to a single UniCast profile that was shared across all the displays.

While GroupCast and the Interactive Wall Map were designed to provide clues for conversations in an office-type environment, "Ticket2Talk", and the "Opinionizer" were system for fostering conversations between participants at larger gatherings of people, for example at conferences or parties. Ticket2Talk [MDS+04] was trialed and demonstrated at the UbiComp 2003 conference. It consisted of a single flat-panel displays situated behind a coffee area. An RFID reader located behind a coffee table was used to identify users standing in front of the table. Once a user had been identified by the system, the display showed the user's name, his photo, and an optional image with a caption describing the user's personal interest. The information was taken from a central database of profile information that users provided when they signed up for the system.

The "Opinionizer" application [BR03] was developed by Brignull and Rogers to study user interaction with public displays. The authors were particularly interested in transitions from peripheral awareness of a display to focused attention and finally to direct interaction with the display. More specifically, the authors aimed to determine why and when such transitions occurred. The Opinionizer was based on a large projected display in a party setting. The display showed a question, picture or statement and enabled users to post their opinions about the subject shown on the display. Users were able to enter opinions using a laptop that was set up close to the public display. Opinions appeared on the display in the form of avatars with speech bubbles. Additionally, the application on the display was modelled similar to a dart board. Different segments of the board indicated different social/professional/educational backgrounds of the poster ("techie, softie, designer, student"). People were able to select which region to place their avatar in. Alternatively

they were able to place it outside of the board, i.e. outside of the given categories. Studies using the display system were conducted at a book launch party at the CHI'02 conference and at a student welcome party at the University of Sussex. Data was collected using a combination of observations, camera recordings of the crowd, and interviews with individual users.

While the applications described above were mainly targeted at fostering immediate and direct interactions between users looking at the displays, a range of public display systems and applications were created to raise awareness of activities within communities by simplifying the dissemination of information by individuals to those communities.

The "Notification Collage" [GR01] represented a virtual blackboard that enabled users to post items that they considered to be of interest to others. The Notification Collage client software could be executed on both desktop PCs and public displays. Supported content types included sticky notes, videos, desktop snapshots, image slide shows, thumbnails of web pages, and activity indicators showing the level of activity in a space based on input from proximity sensors. Once a new item was posted, it automatically appeared on each Notification Collage display. New items were always placed on top of older items by the Notification Collage software, causing older information to be occluded and fade into the background over time. In addition, the system caused old items to rise to the surface from time to time. When run on their desktop machines, users were able to interact with content items and invoke contextual actions on them, including emailing the contributor, initiating an MSN chat with the contributor or visiting the contributor's home page. Users were also able to split their display into two areas: an area into which the system would place new content items, and another area into which users were able to drag items of interest that they wanted to keep in sight at all times. The architecture of the Notification Collage was based on a central directory server storing key/value pairs. On start-up, clients requested and downloaded a complete copy of the dictionary. Posting, modifying and deleting content mapped down to operations for adding, modifying, deleting key/value pairs. The operations were relayed by the central server to all client applications, which would in turn modify their local caches accordingly.

"Aware Community Portals" [SWS01] employed a single projected display in a hallway at MIT Media Lab. Content for the display prototype was automatically harvested off the Web and displayed in a loop. Users were able to interact with the display based on movements, proximity and glancing. The display was equipped with a camera system capable of detecting whether any users

were present, whether these users were moving or stationary, and roughly which direction users were looking into. Different levels of detail were presented on the display depending on whether a user was walking past, whether he was stationary and briefly glancing at the display, or whether he continued to glance for an extended period of time. Additionally a photo was automatically taken of each user and added to a timeline at the bottom of the display, effectively providing a historical summary of past interactions with the display. The Aware Community Portals system was partly implemented using ISIS [AWB97], a programming language for multimedia content developed by Agamanolis et al. Please note that this "ISIS" is different from the distributed systems platform for reliable communication between members of process groups that was developed by Birman et al. [BJ87].

The "Plasma Poster" network [CNDG03, CND+03, CND+04] was developed by Churchill et al. as means for publicising and sharing non-urgent low-key content with colleagues. It used large touch-enabled plasma displays in portrait orientation as display hardware. Users were able to post content (images, movies, formatted text, URLs) via email and a web interface. If not interacted with, Plasma Poster displays cycled through available content in 30s intervals. Users were able to interrupt this cycle by touching the display. Entries shown on the displays were split into a main central body (occupied by the content item itself), the name and a small photo of the author, the submission date, and two buttons allowing viewers to see other posts from the same author or to send him a message via email. Additionally, a horizontal row of thumbnails at the bottom of the screen provided an overview of other content items in the loop, allowing people to scroll to a specific item and to have it displayed. Moreover, buttons at the bottom of the screen allowed users to print, forward the current content item via email or view an overview of all postings currently available. Initially, a total of three displays were deployed in semi-public locations at FXPAL and remained active for at least 20 months. The Plasma Poster network represented a foundation for a range of public display-related research at FXPAL, e.g. covering interaction techniques [CCD+04] and content presentation [DNC03]. Two additional displays were later deployed in Japan and one in San Francisco, USA.

Other display systems for raising awareness and disseminating information within communities created during this time period included "MessyBoard" [FFP02, FP04], the "Community Wall" [SG02, GMS03] and a research prototype described by Huang et al. in 2002 [HTCM02].

Besides systems focused on sharing user-contributed information within a group setting, other

systems were targeted at raising awareness about the state and the dynamics of individual group members and the group as a whole.

In 2001 Farrell described a public display system using the metaphor of a fish tank to visualise group behaviour [Far01]. Fish were used to represent people or things and were able to disseminate information through colour, texture, behaviour or speech bubbles. The fish tank prototype was deployed on a large touch-enabled 50" display. A keyboard enabled users to send messages to fish in the tank. Users could tap on an individual fish to specifically direct a message to that fish, otherwise messages would be sent to all fish. The movement of fish could be used to indicate activities and relationships with other fish, e.g. two fish oriented face-to-face could be used to indicate a meeting. Fish were implemented using an event-based framework, in which fish behaviour and appearance was defined as a reaction to events in the system, such as speech events and motion events. Users were given the option of implementing their own fish using Java code and the "fishdk". Using fishdk users were, for example, able to control the appearance of their fish, it behaviour and movements, and what their fish said in reaction to events.

The "Semi-Public Display" [HM03] provided users of a small work group with a small set of applications to raise awareness of activities within that group. The touch-enabled display was split into four panels, each dedicated to a specific application. The "Reminders" panel displayed a rotating schedule of requests for assistance by individual group members that were auto-extracts from weekly email reports. "Collaboration Space" presented a rotating schedule of topics that users could comment on using a freeform drawing space. "Active Portrait" indicated individuals' activity levels in the lab through dynamic modification of a group photo of all lab members. Activity levels were sensed by monitoring keyboard input. If keyboard activity of an individual stopped, the system very slowly faded the photo of that person to white. Finally, the "Attendance Panel" provided an overview of planned attendance at group events, such as seminars. Events were represented as flowers whose petals are initially all white. Users could walk up to the display and flick one of the petals to pink or blue to indicate that they were or were not planning to attend.

"Online Enlightenment" [HHTM04, TM06] provided information about the instant messaging presence status of members of a lab at Virginia Tech. The displays were custom-built using wood and other materials and represented a floor plan of the lab. Representations of lab inhabitants in the form of caricatures of their faces were mounted onto buttons and added in locations where inhabitants would usually have their desks in the real world. A multi-colour LED next to each

button was used to indicate users' status on MSN Messenger. Additionally, users were able to walk up to the display and press one of the buttons to obtain additional information about that user on an LCD display mounted at the top of the display frame. The display was controlled using a set of 4 Phidget sensor/actuator boards that were in turn connected to a Windows PC.

"AutoSpeakerID" [MDS+04] was deployed in the main auditorium at the UbiComp 2003 conference to automatically provide the audience with information about delegates who were asking questions during the Q&A slots of paper sessions. Like Ticket2Talk, the system used RFID readers that were co-located with microphones to identify users. A large projected display was employed to show name, email address, affiliation and a photo of the person standing in front of the microphone.

"InfoRiver" [PSR+04], demonstrated by Prante et al. at CHI 2004, was a suite of public display devices and applications designed to visualise information flow in an organisation. "InforMall", a large touch-sensitive wall-mounted flat panel display depicted in its upper third a moving river or stream with bubbles representing iconified pieces of information in them. Users were able to drag individual bubbles onto the lower two-thirds of the display surface to access detailed information associated with those bubbles. Moreover, users were able to download information onto "ViewPorts", PDA-like handheld devices. InfoRiver was also visualised on "GossiPlace" devices, ambient light sources capable of displaying notifications. Like InforMall displays, GossiPlace devices could be interacted with using ViewPorts. Finally users were able to use "ConsulTables", touch-sensitive interactive table-top displays, to collaboratively view information and to exchange it with ViewPort handheld devices.

Another semi-public display prototype, presented by Vogel and Balakrishnan in 2004 [VB04], enabled users to use gesture-based interaction to access information on a public wall-mounted display. Information comprised current weather conditions and forecasts, activity levels in branch offices and an event and appointment calendar. The authors were particularly interested in designing a system that would enable multiple users to interact with a display at the same time. During times when no interaction with the display took place, the display showed ambient information with slow update rates. If a user was entering the display's range the display added information from the user's own information sources to the display, e.g. overlaying the user's calendar entries with the public calendar timeline. Depending on their distance from and levels of engagement with the display users were able to use a combination of hand and body gestures and touch-based interaction to inspect information items and retrieve more detailed information. Gesture recogni-

tion was based on a commercial motion tracking system using a combination of wearable reflective passive markers and cameras installed in the surroundings of the display.

Other Display Systems

Between 2000 and 2004 a number of public display prototypes were developed whose main function was to act as interactive kiosks for information access or as shared work surfaces, but that also had the ability to display information peripherally if they were not interacted with.

BlueBoard: "BlueBoard" [RG01, RTW02, RDS02, RTD04], conceived by Russell and Gossweiler at IBM Almaden Research Center provided users with large touch-sensitive plasma displays situated in semi-public spaces that could be used for accessing personal information, collaborative work and the exchange of content between users. Users were identified using RFID badges. If not interacted with, BlueBoard displays showed a loop of generic pages that could be adapted to the time of the day and the display's location.

IM Here: "IM Here" [HRS04] represented a publicly accessible IM kiosk deployed next to a meeting room. The kiosk could be used, for example, by meeting organisers who were already at the meeting room to notify other meeting participants that the meeting was about to start. When not in use, the display showed a set of rolling content, such as announcements for local events. Content could be entered into the system via a web interface and could be supplied either in the form of text or images. An expiration date associated with each content item enabled the system to garbage-collect content that was out-of-date. Each posting was displayed for approximately 25 seconds.

Dynamo: "Dynamo" [IBR+03, BIF+04], published by Izadi et al. in 2003, enabled users to view and share media on large public display surfaces. The authors foresaw a scenario where such surfaces would be installed in public spaces such as hotel meeting rooms. Displays were not required to have network connectivity. Instead users were expected to bring media to the display using USB memory sticks or laptops. The latter could be used to execute instances of Dynamo, rendering them into fully featured extensions of the public display surface. Dynamo supported simultaneous interaction involving multiple users. However, Dynamo displays were not

touch-sensitive. Interaction was performed using keyboard and mouse combinations attached to the display or using laptops running instances of the Dynamo software. The system underwent a series of trials and evaluations: an early evaluation using a projected display involving 30 users at a workshop, a lab-based study using a projected display and involving three groups of four users, and finally a 10-day deployment of Dynamo into the common room at high school using two 50" plasma screens mounted side-by-side as display devices.

2.2.5 2004 – 2008

While the costs of display hardware continued to drop, it was the increasing proliferation of mobile phones that represented one of the major influences in public display research during the years following 2004. Researchers started to use mobile phones as interaction devices. For example, solutions were developed that allowed users to actively and explicitly use mobile phones to influence content on public displays. Moreover, phones were used for more implicit forms of interaction. For example, it was now possible to detect and identify users in the vicinity of displays by scanning for their Bluetooth-enabled mobile phones.

Other themes included the use of public displays for entertainment, novel organic plant displays, the use of public displays for navigation, and investigations into content selection. Moreover, researchers also revisited the theme of using public displays for encouraging social interactions and promoting awareness.

Mobile Phones as Interaction Devices

With the proliferation of mobile phones during the first few years of the new millennium and a continuous increase in processing power, hardware capabilities and communication bandwidth of these devices, mobile phones became the focus for researchers developing techniques for interaction with public display systems.

Miyaoku et al. used colour displays of mobile phones to transfer information to public displays and to render them into rudimentary pointing devices. "C-Blink" [MHT04], published in 2004, used hue differences between subsequent frames of blinking mobile phones to encode information. The signal was picked up by cameras co-located with public displays. Miyaoku et al. reported

about frame rate limitations of the cameras used in the experiment and about difficulties in generating a blinking pattern on the mobile phone with a consistent frame rate, especially if the client application had to be executed inside a Java virtual machine. The resulting system, blinking at 15Hz, enabled Miyaoku et al. to communicate short pieces of information to the display (8 bits in 500ms). While the authors acknowledged that the achieved rate of 15Hz was not sufficient for fluid interaction with the display using the phone as a pointing device, they were confident that these limitations could be overcome in the future using higher-quality cameras and more powerful mobile phones.

Another approach to using mobile phones as pointing devices was presented in the same year by Ballagas et al. [BRSB04]. This approach was based on detecting the relative movement of phones by analysing the optical flows in the input captured via the phones' cameras. Image processing and movement detection were fully performed on the phone. The resulting motion information was communicated to public displays via Bluetooth. Users specified which display to interact with by scanning Visual Codes [RG04] that were co-located with displays and encoded the displays' Bluetooth addresses. The resulting phone-based interaction provided three degrees of freedom, i.e. translation along two axes and rotation around a third axis, with a typical latency of 200ms. While the authors acknowledged that latencies of this dimension may impair user experience, they remained confident that future increases in the processing power of mobile phones would help to alleviate this problem.

In 2005 Berger et al. proposed a solution [BKN+05] for overcoming privacy concerns when displaying personal information on large public displays. Berger et al. proposed to use mobile devices, such as mobile phones, to complement public displays as output devices. Their specific solution was based on blurring sensitive parts of text displayed on a public display. Users were able to selectively retrieve and view those blurred sections on their mobile devices. The mobile device also served as main point of interaction, i.e. users were seen to primarily interact with applications hosted on the mobile device. These applications were able to appropriate public displays as output devices if this was requested by the user.

"TxtBoard" [OHU+05], a display for in-home use presented in the same year by O'Hara et al., allowed family members and friends to display messages on a semi-public display installed in a family's home by sending SMS text messages to the display. A system offering somewhat similar functionality – the ability to send text messages to situated displays – was published one year later

by Holleis et al. [HRKS06]. However, the work presented by Holleis et al. primarily focused on exploring issues surrounding gesture-based interaction with small-scale situated display devices.

A study by Cheverst et al. in 2005 [CDF+05] investigated user acceptance of Bluetooth-based interaction with public displays using mobile phones. The prototype system enabled users to send photos to and receive photos from a public display using the built-in Bluetooth Object Exchange (OBEX) protocol on their mobile phones. The public display itself was touch-sensitive, enabling users to browse through the set of photos that had been uploaded. When not interacted with, the display automatically cycled through its set of photos. The mobile phone interaction solely relied on functionality and user interfaces already built into standard mobile phones and did not require any additional software to be installed on the mobile device.

Finally, in 2006 "iCapture" [MRS06] demonstrated the use of Visual Codes [RG04] to select and download content displayed on public displays to mobile phones. The articles shown on public displays were augmented with Visual Codes. Users wishing to download an article used an application on their camera-enabled mobile phone to capture and recognise the associated Visual Code. The application then encoded a textual representation of the captured code as parameter of an HTTP GET request that was subsequently used to request and retrieve the article in question from a web server.

Using Real Plants as Ambient Displays

The concept of visualising information in an ambient fashion by using plant-like displays had already been demonstrated by Bohlen and Mateas in 1998 with Office Plant #1 and later by Antifakos and Schiele with LaughingLily [AS03]. Researchers took this idea one step further and developed displays that used real plants as display devices. In 2004, Holstius et al. presented results of a two-week trial involving both mechanical plants and real plants as ambient displays [HKH+04]. The plants were deployed in a university cafeteria and were used to indicate the ratio between recycled material and residual waste at a number of trash and recycling containers. Each plant had two light sources as stimuli associated with it that were placed exactly opposite to each other around the plant. By activating the light sources for different amounts of time, researchers were able to make plants lean towards one of the lamps and hence convey a tendency to users.

In the same year, Easterly presented "Spore 1.1" [Eas04], a plant that was used to display the

development of a company's share prices. The system was controlled by an embedded computer that monitored share prices on the web and was able to actuate water pumps to water the plant. Depending on share price development, the plant would be watered or not. The display was in use for more than four months and eventually died of over-watering.

A similar system, "PlantDisplay" (later called "I/O Plant"), was presented by Kuribayashi and Wakita in 2006 [KW06]. An embedded server was used to control the amount of light and water a plant would receive depending on an input variable. Plant growth and health could then be used to reason about the underlying input variable. During a two-month trial phase a total number of four PlantDisplays were used for visualising user happiness (indicated by the number of "good events" users wrote about in their weblogs) and the amount of time users spent communicating with others.

Public Displays for Entertainment

Dropping costs of large flat-panel displays and projection hardware also meant that displays were increasingly used in public multi-user games or interactive art. In some instances, displays represented integral parts of the experience, while in others public displays were used as scoreboards or for enabling users to access additional information.

"Jukola" [OLJ+04], published by O'Hara in 2004 was designed to democratise music choice in bars and cafés. Users used PDA devices associated with each table to vote for songs. A separate 15" touch-enabled public display used as a kiosk enabled users to browse and nominate songs. The system was deployed for a total of one week at a café bar in Bristol.

The same venue also served as a deployment location for "Fancy a Schmink" [RLHS04], a PDA-based multi-player game. A separate public display was used to visualise social networks formed by individual players. If not interacted with, the display cycled randomly through the players. Users were also able to interact with the display to select specific players and view their social networks. Networks were not only presented visually but were also translated into auto-generated music that could be listened to using headphones attached to the public display. The system was in use for 2 days.

Jukola and Fancy a Schmink both used PDAs as interaction devices. One year later, Scheible

and Ojala published the results of their trial of "MobiLenin" [SO05]. MobiLenin enabled users to use their mobile phones to influence music choice in a restaurant by voting for a particular track of a multi-track music video. The mobile phone application was implemented in Python and could be executed on Symbian Series 60 mobile phones. A large projected public display was used to display the music video and the results of the voting phase. The results presented by Scheible and Ojala are based on a combination of questionnaires, server logs, and camera observations taken during a short trial that lasted approximately 12 minutes.

The "CityWall" [PSJ+07] public display, published in 2007, also employed mobile phones as interaction devices. The CityWall display was part of a study investigating "active spectatorship" at large events. Study participants were equipped with camera phones and encouraged to take photos during the event which were then instantaneously uploaded to a web-based photo album service using built-in software on the phone. Once uploaded, the photos were not only accessible on the web, but were also displayed on a large public display (the "CityWall"). The back-projected display itself was located in a shop window. The displays used camera-based hand-tracking (similar to Rekimoto's HoloWall [MR97]) to allow multi-user touch interaction. Study participants and members of the public were able to use the public display to browse and view the photos taken during the event by the participants of the trail. The study was carried out during two events: a Eurovision Song Contest opening party and the Helsinki Samba Carnaval, both lasting for several days.

Public Displays for Navigation

The falling costs of display technology also meant that displays could be deployed more densely, a prerequisite for being able to use public displays for displaying personalised navigation information. In 2005 Kray et al. presented "GAUDI" (Grid of Autonomous Displays) [KKK05], an application for using public displays as dynamic navigation signs. A central server was used to store geometric models of buildings, to calculate routes and to disseminate route information to individual displays. This route information was then further evaluated locally on each display, taking into account display-specific context information before being rendered as output.

Another public display prototype that had navigation as one of its target scenarios was presented by Stahl et al. in 2005. The authors described a framework for scheduling content onto

public displays and for resolving conflicts between content items [SSKB05]. Applications, e.g. navigation applications, issued requests for content to be displayed to a central "Presentation Manager". Requests contained a URL and a description of the type of the content that was to be displayed, as well as a specification of the targeted display or location. Additionally requests could contain an optional set of constraints, such as the minimum display time in seconds, the minimum display size (specified as "small", "medium" or "large"), the minimum display resolution, and whether the presentation required audio output or user interaction. Requests between applications and the Presentation Manager were communicated asynchronously using an instance of Johanson's Event Heap [Joh03], a tuple-space-based coordination platform. The Presentation Manager was responsible for interrogating the system's central device manager to find displays matching the provided constraints and to create a system-wide plan for all presentation requests. Conflict resolution was achieved via a combination of display pre-emption, the division of display real-estate into multiple display areas, and re-planning.

A prototype of the system was deployed in a test environment and comprised a large wall-mounted plasma display, two wall-mounted tablet PCs, five PDAs acting as electronic door plates, a display that was projected onto a semitransparent glass door, a tablet PC mounted onto a shopping cart, a display projected onto a table, and a steerable projector.

Raising Awareness and Encouraging Social Interactions Revisited

Revisiting the theme of using public displays for encouraging social interactions, "Sparks" [CLS+05] was a conceptual peripheral display system to facilitate and mediate conversations in conference settings. Unlike previous systems that were either based on situated public displays or on wearable displays, Sparks employed steerable projection units that followed users as they moved around. Users were required to initialise the system by specifying a set of keywords describing their interests. Sparks subsequently used computer vision to track users' positions and orientation and projected a ring ("aura") on the floor around each of them displaying their name and keywords. Additionally, lines were projected on the floor connecting identical keywords in the auras of different individuals, effectively creating paths to other individuals with similar interests. Lines varied in thickness, indicating distance, and in colour to differentiate individuals a person had already spoken to from new conversation partners. If a group conversation was detected by Sparks, the area around the group was marked off by projecting a "group pad" around participating users.

Sparks also enabled users to retrieve additional information about past conversations. To access information, users were able to use their name tags or badges. If brought it into a horizontal position, Sparks projected information directly onto the tag. Users were able to scroll information by tilting their tags. The height at which the tag was held influenced the level of detail of the information displayed on the tag. In their publication, Chew et al. describe the results of an initial user study based on mock-ups, but it remains unclear whether Sparks was ever prototyped.

While previous systems for raising awareness that we had encountered had all been designed for and deployed into industrial and academic research offices and labs, "AwareMedia" [BHS06], published by Bardram et al. in 2006, represented a display designed for use in hospital wards and operating theatres. Its main function was to facilitate the planning of interventions. The displays provided overviews of staff assignments and locations, and of past, present and planned surgical interventions. Moreover, AwareMedia was capable of displaying live video feeds captured in operating theatres and enabled staff to send instant messages to other displays and to individuals' mobile phones. Displays traced the locations of members of staff by tracking their Bluetooth-enabled mobile phones or wearable Bluetooth tags. A total of 10 displays were deployed into a real hospital. The deployment consisted of three 20 inch touch displays, equipped with webcams, that were deployed into three operating theatres, two 40 inch touch screens arranged side-by-side in a coordination room, and several standard PCs and monitors in a patient ward and a recovery ward. The system was in use for more than two months.

Optimising Content Selection

The issue of optimising content selection for different display locations in a network of situated public displays for information dissemination was the topic of a paper published by Müller and Krüger in 2006 [MK06]. The paper proposed a methodology for creating models, based on a combination of detecting and identifying users using Bluetooth scanners, gaze recognition techniques, and the use of user-contributed personal profiles.

The authors also described a public display testbed that they intend to use to implement and evaluate the proposed approach. The testbed comprised one display in a lobby and one in a hallway, with a third deployment being planned at the time of publication (2006). Displays were split into three regions: a ticker bar at bottom, a main area showing a list of auto-expanding news

headlines and announcement, and a side-bar displaying showing various pieces of information, including the menu at the local cafeteria, a bus timetable and the local weather forecast. News headlines and announcements could created using a Java-based WYSIWYG editor. At the time of publication, the display in the lobby had been deployed for more than five months and had displayed around 100 news items.

The topic of content selection for different display locations and users was also central to the "Prospero" project [Con07], whose final report was published by Congleton in May 2007. Prospero aimed to empower users by transforming them from recipients of broadcasts to active contributors who shape what is presented on public displays. The Prospero prototype system consisted of a wall-mounted 30 inch LCD screen and a display projected onto the ceiling. Both displays were installed in lab space that was used for lecturing and research activities. The displays were capable of displaying a range of content that was either harvested from external sources, such as Flickr and Google Maps, or auto-generated from local sources, such as LDAP directories and swipe card readers. Specific "Display Modules" on the displays were responsible for rendering specific types of content. Prospero displays were able to split a display surface into multiple regions, each capable of displaying content independently. "Rotation Modules" and "Priority Modules" were used to influence this behaviour, defining for example how the different regions were laid out on the screen. Rotation and priority modules also determined the types of content that were displayed at any point in time, depending on users' and global preferences. The combination of configurable modules and a mechanism for detecting users based on swipe cards and PC log-ons provided a basic framework for displaying personalised content to lab inhabitants. However, it is unclear how Prospero handled conflicting user preferences. Moreover, it is unclear how much personalisation the deployed prototype actually supported. Prospero was used for a total duration of six weeks.

2.3 Analysis

In this section we classify the public display systems presented so far. We employ a taxonomy based on four major categories:

- *objectives*, i.e. the purpose of public display systems when viewed from the perspectives of users, researchers and display owners.

- *content*, i.e. properties of the material that was shown.
- *deployment*: parameters regarding the implementation and use of public display systems, such as numbers of users, deployment durations, hardware and software platforms used, and deployment locations and contexts.
- *evaluation*, providing an overview of the techniques used to evaluate the surveyed public display systems.

2.3.1 Objectives

Typically, public display systems provide different values to different people, depending on the roles these people hold in relationship to displays. From a researcher's point of view, the objectives of a deployment of a public display system or application may be to trial and study new interaction techniques. For actual users, the same deployment may primarily provide them access to information or make them aware of events in their local community. We therefore distinguish between objectives for researchers, users and owners.

Researchers' Objectives

Researchers create and deploy public display prototypes for a multitude of reasons. We have classified the systems and applications encountered in our survey into three high-level research themes: research into public display hardware, research into systems support for public displays, and research related to the presentation of and interaction with content on public displays (human computer interaction).

- *human computer interaction* was the most commonly encountered theme in our survey, with more than 75% of the surveyed system having an HCI-related aspect to them. Within the broad field of human computer interaction we were able to identify the following subcategories:
  - *social aspects*, for example investigating the motivations users have for looking at and interacting with public display systems, and how the use of public display technologies

impacts individuals and social groups. Early public display systems, such as Hole-In-Space[GR80], the audio/video link between Xerox PARC's Portland and Palo Alto offices [GA86, BHI93], and VideoWindow [FKC90] were used by researchers to study how the use of technology changed people's communication practices and how it influenced physically distant communities that were connected via virtual windows. Brignull et al. [BR03] studied user involvement with public display systems and the motivations that caused users to transition between different levels of engagement with public displays.

- *novel applications*: the design and evaluation of experimental applications and content for public displays was a focus of a range of research prototypes. An early example can be found in the form of the Learning Community Newspaper display [HBL98] that focused on the design and evaluation of an application for improving awareness of events and accomplishments within a professional community. Later examples include Ticket2Talk and AutoSpeakerID [MDS+04] (applications aimed at enhancing conference settings), GoupCast and the Interactive Wall Map [McC02](aimed at fostering social interaction between members of a community), and IM Here [HRS04] (exploring the use of public displays for informal communication).
- *user interaction techniques*: Vogel et al. investigated multi-user interaction with public displays based on gestures [VB04], while Ballagas et al. [BRSB04] and Miyaoku et al. [MHT04] both focused on using mobile phones as interaction devices.
- *user interface design*: Brignull et al. used the Dynamo display prototype [IBR+03] to study and evaluate techniques for simultaneous multi-user interaction with a shared public display surface. Plasma Poster displays served as a testbed for document windows that mimic the behaviour of paper notes in the real world [DNC03].
- *user involvement and user-contributed content* were, for example, topics of research in the context of the Learning Community Newspaper [HBL98] and the Community Wall [HBL98], and included investigations into ways of enticing users to contribute content to public display systems.
- *privacy*: Berger et al. [BKN+05] proposed to use mobile devices to overcome privacy concerns when visualising information on public displays by using users' personal devices as output devices for sensitive information.
- *information visualisation*: novel forms of presenting information to users was, for example, a topic of research in the context of ambient display systems, including the Water

Lamp [DWI98] and Infotropism [HKH+04].

  - *content selection* was one of the topics investigated by Müller and Krüger [MK06]. The authors described a methodology for modelling user preferences by combining explicitly provided user profile information with passive sensing of user behaviour.
  - *art*: while some display prototypes were created with the aim of disguising the visualisation of information as works of art (e.g. Redström et al. [RSH00], and Holmquist and Skog [HS03]), other researchers aimed to directly create public display prototypes as pieces of art (Scheible and Ojala [SO05]).

- *display hardware*: research into the development of novel forms of display hardware were, for example, undertaken in the context of the Interactive Fog Screen [RDO+05] (projection onto a curtain of fog), by Rodenstein [Rod99] who developed a technology allowing regular windows to be used as projection surfaces, and by Pinhanez et al. [Pin01] who investigated the use of steerable projectors and cameras to project touch-sensitive user interfaces onto regular surfaces.

- *systems support* for public display systems was the topic of a relatively small number of the surveyed pieces of research. Examples include work performed by Stahl et al. [SSKB05] (describing a framework for scheduling content onto public displays and for resolving conflicts between content items), and Prospero [Con07, Pro08] (featuring a range of interchangeable modules to influence content selection and visualisation).

Usage Objectives

Different public display systems may provide different benefits to users. The surveyed public display systems were designed to provide value to users in five main ways:

- *awareness*: the peripheral nature of public display systems makes them suitable for disseminating information to individuals and communities. Awareness was already a theme in some of the earliest research into public display systems: the virtual windows created at Xerox PARC [GA86, BHI93] and Bellcore [FKC90] were both partly aimed at increasing awareness for the activities in physically separated research groups. The Dangling String described by Weiser and Brown [WB96] made office inhabitants aware of varying levels of network activity. The Learning Community Newspaper [HBL98] developed by Houde et al. was aimed

at providing a tool for raising the awareness for activities, news, and achievements among members of the Learning Communities Group at Apple Computer. Overall, more than 75% of the surveyed pieces of research had an awareness-related theme.

- *access to information* is often coupled with an awareness theme. For example, systems like the Learning Community Newspaper [HBL98], the Plasma Posters [CND+03], the Notification Collage [GR01], MessyBoard [FP04], and the Community Wall [SG02] allowed users to use these systems to share pieces of information with other users with the ultimate aim of making them aware of activities taking place within the community. Other research prototypes focused more directly on providing access to information. Examples include the steerable projector and UI system for augmenting retail environments that was developed by Pinhanez et al. [Pin01], and iCapture [MRS06]. The latter allowed users to select and download news stories (sourced from local news feeds and the BBC web site) from the public display system onto their personal mobile phones.

- *catalyst for social interactions*: some public display systems aim to assist users in sparking off conversations with other users. These systems typically visualise information about individual users in an attempt to provide clues for conversations. GroupCast [McC02] and the Interactive Wall Map [McC02] displayed content taken from the personal profiles of users that were located in the vicinity of the displays. Thinking Tags [BMMR96] and Meme Tags [BMV+98] provided potential conversation partners with information about the wearers' opinions regarding certain topics. System such as the Plasma Posters [CND+03] allowed individuals to raise awareness (potentially leading to an increase in interpersonal interactions) for their own activities by posting content on the displays. Ticket2Talk [MDS+04] and Sparks [CLS+05] displayed information about other users' interests with the expressed aim of fostering conversations. Other systems, such as Telemurals [KD04] and VideoWindow [FKC90] were aimed at providing channels for spontaneous interactions between physically distant users through the use of audio and video links.

- *communication*: some of the surveyed prototypes provide users with means for electronically communicating with other users. Electronic door displays, such as Hermes [CFDR02], enabled users to walk up to the displays and leave messages for the display owners. TxtBoard [OHU+05] and the situated displays developed by Holleis et al. [HRKS06] allowed users to leave messages for other users by sending messages to the displays. Other display prototypes

provided means for synchronous communication. Users were, for example, able to walk up to an IM Here [HRS04] display and communicate with other users using an instant messaging client. Dynamic Photos [Gre99] allowed audio and video connections to be established to remote users by stroking the photo of a person on the display. Communication also played a major role in the various display prototypes that provided virtual windows into distant locations (e.g. Goodman and Abel [GA86, BHI93] and the Virtual Kitchen [JVG+01]).

- *entertainment* was obviously the main focus of interactive games, such as MobiLenin [SO05], Jukola [OLJ+04] and Fancy a Schmink [RLHS04], but also played a role in other systems. Examples include the Hermes Photo Display [CDF+05] and City Wall [PSJ+07] that both enabled users to browse user-contributed photos on a public display.

Owners' Objectives

In the context of this survey we define display owners as individuals or groups of people who control the spaces that public displays are deployed in. Display owners are therefore able to influence the selection of applications and content that are made available on these displays.

There were relatively few examples in our survey in which the display owners were different from the researchers supervising the deployments. Jukola [OLJ+04], MobiLenin [SO05] and Fancy a Schmink [RLHS04] were all deployed into café/bar settings. None of the surveyed publications about these deployments provided any insights about the owners' objectives. However, we expect that increased *publicity* as a result of the display deployments most likely played a role.

CityWall [PSJ+07] was deployed into a display window in the city centre of Helsinki. However, it is unclear whether this space was owned by 3rd parties or by the researchers themselves.

Electronic door displays, such as Hermes [CFDR02], OutCast [MCL01] or Dynamic Door Displays [NTDM00], provided owners with means for *asynchronous communication* with visitors, and for the *dissemination of information*. Similar objectives can be found in the case of TxtBoard [OHU+05].

Some wearable displays (e.g. Thinking Tags [BMMR96]) acted as *catalyst for social interactions* with their owners. In other cases where wearable displays could be appropriated by other users (see Falk and Björk [FB99]) we suspect that reciprocity was one of the motivations for wearing such

a display: owners wore displays and allowed others to appropriate them because they expected other people to do the same.

AwareMedia [BHS06] was deployed into a hospital ward for use by doctors and nurses. In this case, the display users were also acting as display owners, and hence the owners' objectives were equivalent of those of the displays' users.

However, most of the surveyed public display systems that we surveyed were either deployed into academic and industrial research labs and offices, or into academic conference settings. Researchers therefore acted in most of these cases also as display owners. Examples include the Plasma Poster displays [CND+03] which were deployed into the researchers' own premises at FXPAL in Palo Alto, GroupCast [McC02] (deployed at Accenture Technology Labs), and the Learning Community Newspaper [HBL98] which was deployed into the research group's premises at Apple Computer. It is obvious that the owners' objectives in these instances are equivalent to the researchers' objectives.

2.3.2 Content

Content is one of the most central aspects of any public display system and plays a significant role in making public displays interesting to users. Content therefore has a major influence on the success of public display deployments.

Nature

During our survey of research into public display systems we have identified three major classes of content: *information, interactive applications*, and *virtual windows*.

In our survey, information was by far the most frequently encountered type of content, with over 70% of the surveyed systems displaying information of one form or the other. The Dangling String described by Weiser and Brown visualised *measurements* of the current network activity at their premises [WB96]. *Navigation* information was at the heart of the prototype by Kray et al. [KKK05] and served as scenario for both Stahl et al. [SSKB05] and Pinhanez [Pin01]. The display prototypes created by Huang et al. [HRS04] showed *announcements* for events when not

used interactively. *Recommendations* and *items of interest* were at the heart of systems such as the Plasma Posters [CND+03], the Community Wall [GMS03] and the Notification Collage [GR01]. *Signage* information was provided by electronic door displays, such as Hermes [CFDR02] or the RoomWizard [OPL03]. Some displays aimed at visualising *opinions*. Example include Thinking Tags [BMMR96] and the Opinionizer [BR03], which both visualised information about users' attitudes towards certain topics.

Interactive applications were, for example, encountered in the form of shared workspaces (e.g. Dynamo [IBR+03], BlueBoard [RTD04]), applications for communication (e.g. IM Here [HRS04]) and applications for entertainment (e.g. MobLenin [SO05]).

Virtual windows to physically distant locations were the main type of content encountered in the early public display systems, such as Hole-In-Space [GR80], Video Window [FKC90], and work performed by Goodman and Abel at Xerox PARC [GA86]. Later examples include Telemurals [KD04] and the Virtual Kitchen [JVG+01].

Ingestion and Formats

One of the strengths of public display systems over traditional signage technologies (e.g. posters) is that the technology enables information on the displays to be updated or changed with relatively little effort. It is therefore not surprising that most of the surveyed prototypes provided some form of support for ingesting new content into the system. In some instances ingestion was performed automatically from external sources, e.g. from shared calendars in the case of Dynamic Door Displays [NTDM00] or from external websites in the case of the Stone Garden [RSH00]. In other cases tool support was added to allow owners or users to add content with minimal effort (e.g. see Houde et al. [HBL98] who used a tool chain of scripts to extract text-based content from email messages). Finally, in some instances content ingestion was a manual process, for example in the case of IM Here where a single member of staff was responsible for manually adding announcements for events into the system.

Different display prototypes provided support for ingesting different types of content that can be classified into the following categories (please note that display systems that supported user-contributed content often provided support for a combination of content formats):

- *text*-based content was for example supported by the Community Wall [GMS03]. The Community Wall enabled users to contribute text-based content via email, a PDA-based interface, a custom recommender system back-end with its own user interface, or by using optical character recognition software to process physical forms. The Hermes electronic door plates [CFDR02] enabled display owners to leave text-based messages on their door displays.

- *images*. The Plasma Poster displays [CND+03] allowed users to add image-based content, such as photos. The Hermes Photo Display [CDF+05] encouraged users to transfer photos from their mobile phones to a public display using a Bluetooth connection.

- *audio and video* was used as content formats in some of the earliest public display systems in the research domain: Hole-In-Space [GR80], Xerox PARC's virtual window between their office in Palo Alto and that in Portland [GA86, BHI93], and Bellcore's VideoWindow System [FKC90].

- *web pages* in the form of HTML files or URLs. Examples include FLUMP [FWD+96], which allowed users to add personalised HTML pages to the system, and the Notification Collage [GR01], which was capable of displaying thumbnails of user-supplied references to web pages.

- *office documents*, such as PDF or MS Office files, were supported by a rather limited number of systems: Word, PowerPoint, Excel and PDF documents could be viewed and exchanged on Dynamo displays [IBR+03].

- *custom formats* were mainly encountered in the context of ambient display systems. These systems typically sourced information from physical or virtual sensors and visualised them in an abstract manner. For example, Weiser and Brown's Dangling String sensed and visualised network activity in the building. Wearboy [LBF99], a modified GameBoy device, allowed users to contribute whole applications in a proprietary format. The applications were deployed onto the displays using cartridges.

Provision

The provision of a continuous stream of incoming content to keep users interested is often a key factor determining the success of a public display deployment (see Houde et al. [HBL98]). In our survey we have identified two main providers of content: *display owners* and *users*.

In some cases, content for public displays is *provided by the owners* of those displays. As deployments for the purpose of research are mostly owned by researchers themselves, content accompanying deployments and trials is in some cases provided by those researchers: Researchers provided the initial questions in the case of the Opinionizer system [BR03] and Borovoy's Thinking Tags [BMMR96, BMRS98], multi-track videos in the case of MobiLenin [SO05] and announcements in the case of IM Here [HRS04].

Owners are also typically responsible for providing content for electronic door plate displays that are associated with individuals, such as Hermes [CFDR02], Dynamic Door Displays [NTDM00], OutCast [MCL01] and IMMS [BJ04].

Requiring owners to provide content is obviously only feasible if the amount of content that is to be generated is small and if content does not have to be updated very often. These conditions can, for example, be found during short-lived trials. If a larger amount of content and more frequent updates are necessary, content may, for example, be *contributed by a larger base of users*: the Learning Community Newspaper [HBL98] relied on users to send in news stories via email. In their publication Houde et al. specifically acknowledged the challenge of maintaining a constant stream of incoming content to keep people interested in the displays, and hence to attract more content. The Plasma Poster Network [CND+03], the Notification Collage [GR01], Messy Board [FFP02, FP04] and the Community Wall [SG02, GMS03] relied on content contributed by users. GroupCast selected content from users' UniCast profiles. CityWall [PSJ+07] and the Hermes Photo Display [CDF+05] encouraged users to submit photos to the displays. Displays such as Sparks [CLS+05], and AutoSpeakerID and Ticket2Talk [MDS+04], whose aim it was to provide information about their users, naturally sourced their content from these users.

Content that was provided by users or owners sometimes consisted of a combination of a customised presentation and information that was automatically obtained from sensors or other external data sources, including 3rd party web pages. This technique was, for example, used if the native format of the sourced information was unsuitable for presentation on a public display system. Examples include ambient displays, such as PinWheels [IRF01] or Office Plant #1 [BM98], that were not able to render standard content types. Content also had to be processed and re-formatted if it was available in a format that was not directly suitable for presentation on a display: Dynamic Door Displays [NTDM00] included information sourced from a location system to show the owner's current location. FLUMP [FWD+96] showed, among other things, the number of new

and old mail messages in a user's inbox, along with the sender and subject line for each unread message.

Moderation

Depending on whether content for a public display system is provided by trusted individuals or communities, or by the general public, different levels of moderation might have to be present to prevent the use of offensive or inappropriate content.

Of the surveyed public display literature, only the publication about Jukola [OLJ+04] explicitly mentioned the inclusion of content moderation policies. Jukola allowed users to upload additional songs using a web-based interface. The songs were manually moderated by researchers to ensure that only royalty-free songs were played back by the system.

In the case of the surveyed ambient displays, content is typically selected and ingested at deployment time only. The appropriateness of content is therefore checked by researchers at deployment time, and there is no need for the moderation of content during run-time.

Moderation is also rendered unnecessary if content is directly supplied by display owners, e.g. in the case of electronic door plate systems like Hermes [CFDR02] or OutCast [MCL01], or if content is supplied by the researchers themselves (see MobiLenin [SO05] or Thinking Tags [BMMR96, BMRS98]). In these cases, posting inappropriate content would only reflect badly on the display owners themselves. Similarly, content does not require moderation if it is sourced from trusted external sensors, such as the location information shown on Dynamic Door Displays [NTDM00].

Research prototypes using user-contributed content often rely on the fact that these research systems mostly have only small, closely knit communities of users. Churchill et al. mentioned that the Plasma Poster Network only had a "minimal content moderation policy, relying on social accountability to ensure appropriate content is posted" [CND+03]. Moreover, the authors stated that this was feasible due to a "shared sense of content appropriateness" [CND+03].

Scheduling

The scheduling of content is obviously not an issue in case of displays that support only one piece of content. Examples include ambient displays and informative art, such as WebAware [SH00] or the Soup Clock [HS03].

However, if multiple pieces of information that cannot be shown simultaneously are to be shown on a public display, mechanisms are required to arbitrate between these content items and to determine when each piece of content is to be made visible on the display. In our set of surveyed systems for research, content was either scheduled by the *public display system software* (e.g. randomly or based on policies) or *interactively by users*.

For example, the system developed by Müller and Krüger [MK06] cycles through available news items in an endless loop. Loops of content were also used to cycle through announcements in IM Here [HRS04], and to underpin the "attract loop" in BlueBoard [RTD04]. Other systems used events to determine which piece of content display: GroupCast [MCL01, McC02] showed pages taken from the UniCast profiles of users that were detected in the vicinity of the display. Ticket2Talk received events from RFID readers that indicated the presence of users. Events were then individually queued, and content associated with each identified user was displayed for a certain period of time. Prospero [Con07] employed policies to select and schedule content based on priorities specified by users. FLUMP [FWD+96] displayed personalised pages if registered users were detected by the associated Active Badge system. If no registered users were in range, FLUMP cycled through a loop of default content.

In other systems, content was accessed by users in a purely interactive fashion. Examples include the Dynamic Door Displays [NTDM00], where additional information was made available interactively to visitors, or Dynamo [IBR+03, BIF+04], where users were able to interactively select, view and share content.

Finally, many systems employed a combination of system-scheduled and interactively scheduled content: The Plasma Poster Network [CNDG03, CND+03, CND+04] allowed users to pause the system's content loop and interactively navigate through available content items. Similar functionality existed for GroupCast [MCL01, McC02] and the Hermes Photo Display [CDF+05].

Interactivity

The display prototypes we encountered in our survey can in general be classified into *interactive* and *non-interactive* systems.

About half of the surveyed public display systems were *non-interactive*. Examples include the early public display systems that used audio and video content to create virtual windows between physically remote places (e.g. "Hole-in-Space" [GR80], Xerox PARC's Palo Alto – Portland link [GA86, BHI93] and "VideoWindow" [FKC90]), ambient displays (e.g. the "Water Lamp" and "Pinwheels" [DWI98], the "Information Percolator" [HHT99], "Office Plant #1" [BM98]), and the "Learning Community Newspaper" [HBL98].

To be able to provide users with personalised content, some non-interactive systems included the ability to sense and identify users (e.g. using infrared badges or RFID technology) and adapt their content accordingly. Examples include FLUMP [FWD+96], GroupCast [MCL01, McC02], AutoSpeakerID and Ticket2Talk [MDS+04] and Prospero [Con07, Pro08].

Besides obvious uses for interactivity, e.g. in the context of interactive games such as Fancy a Schmink [RLHS04], *interactive* systems may allow users to select between different types or items of content. Electronic doorplate systems may, for example, allow users to obtain additional information about the owner by interacting with the display, as was the case for Nguyen et al.'s "Dynamic Door Display" [NTDM00]. Through interaction systems may reveal additional levels of detail about a piece of information (examples include Vogel et al. [VB04], the Plasma Poster Network [CND+03], Online Enlightenment [TM06] and Sparks [CLS+05]). In other cases interactivity is used browse and select content (e.g. in the CityWall [PSJ+07], Aware Community Portals [SWS01] and the Plasma Poster Network [CND+03]).

Interactivity may serve as means for allowing users to add information to the system and to exchange this information with other users. This information may be of private nature and targeted at specific individuals. For example, electronic doorplate systems, such as Hermes [CFDR02], Dynamic Door Displays [NTDM00], OutCast [MCL01] and IMMS [BJ04] provided visitors with the means for leaving messages addressed at the displays' owners. Information may also be of public nature and designed to be displayed on the public displays themselves. For example, the Plasma Poster Network allowed users to add comments to content items shown on a display

[CCD+04]. CityWall [PSJ+07] provided users with the means to upload photos taken at an event to a public display. Dynamo [IBR+03, BIF+04] enabled users to upload information to a public display surface, on which it could be interactively viewed and shared with other users.

Context-Sensitivity

Some of the encountered public display systems used context-sensitivity to adapt content to better match the requirements of users. In the context of our survey we have identified the following types of context-sensitivity: *personalisation*, and adaptation to the *time of the day*, *display location*, *display orientation*, *user location*, and *levels of activity* in the vicinity of displays.

Content on displays may be affected by a range of contextual parameters. A relatively large number of the surveyed display systems provided support for *personalised content*, i.e. content was adapted to the identity of the individual(s) interacting with it. For example, GroupCast [MCL01, McC02] displayed pages that were selected from the profiles of users that were in the vicinity of the display. Ticket2Talk [MDS+04] and AutoSpeakerID [MDS+04] both provided information about users in the vicinity in an attempt to inform other users. If registered users were present, FLUMP [FWD+96] showed pages that these users had previously personalised. Prospero [Con07] attempted to adapt content to the overall preferences of users present in the deployment location. Both Hello.Wall [PRS+03, PSR+04] and GossiPlace [PSR+04] were able to sense users in the vicinity and adapt information to their presence. While personalisation in these systems relied on sensing users in displays' vicinities, other systems relied on explicit log-in and log-out actions. In the cases of Dynamo [IBR+03, BIF+04] and BlueBoard [RTD04] users explicitly logged into the system to access personalised workspaces.

Although being public or semi-public, some display systems were owned by single individuals and were set up to provide information about those individuals. Examples include electronic door displays (e.g. Hermes [CFDR02], Dynamic Door Displays [NTDM00] and IMMS [BJ04]), wearable displays (e.g. Meme Tags [BMV+98]) and Sparks [CLS+05].

Content on a smaller number of display systems was sensitive to the *location* these displays were deployed in, the *time of the day*, or their *orientation*: BlueBoard [RTD04] displays featured an "attract loop" that showed pages that were tailored to each display's location and the time of the day. Unlike electronic door plates that were owned by individuals, RoomWizard [OPL03]

displays provided information about the meeting room they were deployed in front of. GAUDI [KKK05] navigation displays were capable of adapting content to their deployment location and orientation. Müller and Krüger [MK06] argued for an adaptation of content to location and time to best reflect users' interests.

The Community Wall [SG02, GMS03] adapted its content to the presence of users by modifying the font sizes and levels of detail depending on whether users were distant or close by. *Distance* also influenced the types of information and the presented levels of detail in both Hello.Wall [PRS+03, PSR+04] and the public display prototype developed by Vogel et al. [VB04].

Other displays adapted to the *levels of activity* present in the environment: LaughingLily reacted to different levels of activity in meetings, while increased levels of activity and interaction led to increased sharpness and clarity of the obfuscated video feed in the Telemurals system [KD04].

Coordination

In display systems comprising more than one display, support for the coordination of content across displays may exist, for example making sure that related videos that are shown on two neighbouring displays are played in a synchronised fashion. We found that the surveyed display systems *mostly provided no support for coordination*, but also found a few examples where *support for the coordination of content* was present.

The version of Pinwheels [IRF01] that was deployed at the NTT-ICC museum in Tokyo consisted of an 8x5 array of coordinated pinwheels. Information, symbolised by a row of spinning wheels, moved through the array from front to back to represent time. However, we acknowledge that instead of viewing this installation as a deployment of 40 individual displays that operated in a coordinated fashion, it would be equally valid to view it as a single, uncoordinated display that consisted of 40 pinwheels.

MessyBoard [FFP02, FP04] used a centralised architecture, in which all displays connected to a single "MessyBoard space" displayed exactly the same output: "Users view and interact with the space in a web browser on their own computers and a server keeps everyone's view synchronized" [FP04].

When deployed, RoomWizard [OPL03] door displays buddied together and exchanged infor-

mation about room bookings. However, the visualisation of content itself was not coordinated.

Sparks [CLS+05] displayed lines between two users' auras if those two users had common interests, potentially requiring coordination if more than two projectors were involved in creating the auras. However, the publication does not explicitly mention multi-projector set-ups or the need for coordination.

GAUDI navigation displays used a central server as point of coordination. The server was responsible for evaluating route information and for distributing resulting content to all involved displays.

The software infrastructure presented by Stahl et al. [SSKB05] used a central "Presentation Manager" to process incoming requests and plan display use, but did not support the visualisation of content in a coordinated fashion.

2.3.3 Deployment

In this category we provide an overview of deployment-related aspects of public display research, based on the results of our survey. We have structured the overview to focus on the duration and scale of the deployments, the technologies used, and the contexts that the public displays were deployed into.

Please note that in the context of this section we consider a system as "deployed" if it was trialed or demonstrated, even if this trial or demonstration took place in a lab setting.

Deployment Scale

Number of Displays. Our survey concluded that about 50% of the deployments only used single displays (e.g. GroupCast [McC02], TxtBoard [OHU+05] or AutoSpeakerID [MDS+04]), with about a further 20% using from two to six displays: Hole-In-Space [GR80] was based on a pair of displays deployed in physically distant locations. The Interactive Wall Map [McC02] used two displays embedded into a wall map of the world, and a total of five RoomWizard door displays [OPL03] were deployed and trialed.

Slightly more than 10% of the systems surveyed were either in a prototype or pre-prototype stage, and had not been deployed yet. Examples for undeployed displays include Mosaic View [Mis06] and Office Plant #1 [BM98]. For slightly less than 10% of the surveyed systems we were unable to obtain information about the number of deployed displays.

Only about 10% of the surveyed public display systems used ten or more displays: the Hermes door displays [CFDR02] saw an initial deployment of ten displays, with a subsequent deployment of 40 displays once the Computing Department had relocated to a new building. Badram et al. deployed a total of ten AwareMedia displays into a hospital ward [BHS06]. Borovoy et al. conducted trials with roughly 200 Thinking Tags wearable displays [BMMR96, BMRS98] and 400 Meme Tags wearable displays [BMV+98].

Number of Users. In many cases it is difficult to make accurate statements about the numbers of users a public display system had while it was in use. This is often due to the total lack of information about the actual number of users that used or interacted with a system.

Some publications report the numbers of direct interactions with displays, but omit figures for the peripheral use of the displays. Such peripheral use may, for example, include instances where people glanced at displays while they walked past or looked at display content from a distance without explicitly interacting with the display. The evaluation of the CityWall [PSJ+07] public display focused on a small number of test subjects that had been equipped with camera phones to provide content. However, it did not provide any in-depth information about the use of the public display by other passers-by although the publication mentions that the display was used by people that were not involved in the trials. Müller and Krüger reported [MK06] that they had 10 users who regularly contributed content for their displays, with around 600 users walking past the displays every day. During the first trial of the Opinionizer [BR03] at a book launch party at CHI'02 a total of 40 users interacted with the displays. However, the launch party was attended by a total of 300 people who therefore all represented potential users of the Opinionizer display.

Other reports are based on user trials, but do not provide information about day-to-day use of the system. For example, the Hermes Photo Display [CDF+05] remained installed and operational for more than a year. However, the only usage data available stems from a short trial with a total of 17 users.

Large numbers of users are often the result of deploying display prototypes into conference settings. Ticket2Talk and AutoSpeakerID were deployed at the UbiComp'03 conference. The authors had previously encouraged all delegates via email to sign up to the system, resulting in a total number of 201 registered users. Moreover, roughly 600 delegates attended the conference and therefore all represented potential users. For the Thinking Tags [BMMR96, BMRS98] and Meme Tags [BMV+98] trials, Borovoy et al. simply equipped all attendants at a "10 years MIT Media Lab" event and a conference with wearable displays, resulting in user numbers of 200 and 400. However, in other cases large numbers of users simply reflect the overall scale of the deployment. The deployment of about 40 Hermes door displays [CFDR02, CAS08] at the Computing Department at Lancaster University, combined with the fact that a significant number of those displays were installed outside offices with multiple occupancy, resulted in a constant base of about 100 display owners. The number of potential users is much larger and includes other members of staff, students and visitors.

Duration

Deployments are costly, as is the maintenance of displays once they have been deployed [SFD+06b]. Some of the surveyed systems were therefore never deployed. Instead they were evaluated using short user trials inside the lab, or were deployed for rather brief periods of time that enabled researchers to collect enough evaluation data to publish their work. We found that the deployments and trials in our survey could be classified into deployments that lasted:

- *a few hours*: of those deployments for which information about the deployment duration was available, the user trial of MobiLenin [SO05] represented the shortest deployment with a total duration of 11 minutes and 45 seconds. The Opinionizer [BR03] underwent two trials with durations of 2 hours and 5 hours.
- *one or more days*: Thinking Tags [BMMR96, BMRS98] and Meme Tags [BMV+98] both underwent 2-day trials. The CityWall [PSJ+07] public display was used during one 3-day and one 2-day trial. Galloway's Hole-In-Space linked New York City and Los Angeles during 3 evenings.
- *one or more weeks*: TxtBoard [OHU+05] was in use for 2 weeks, Huang's Semi-Public Display [HM03] for more than 2 weeks.

- *one or more months*: the Telemurals system [KD04] linked two semi-public spaces located in student dormitories for a total duration of 2 months. Kuribayashi's PlantDisplay [KW06] was in use for 2 months, Fish's VideoWindow [FKC90] for 3 months, and the Learning Community Newspaper [HBL98] for 6 months.

- *one or more years*: about 10% of the surveyed systems represented longer-term deployments that lasted one year or longer. The Plasma Posters [CNDG03, CND+03, CND+04] were in use for at least 20 months, the Community Wall [SG02, GMS03] for at least 13 months. The Hermes electronic door plates [CFDR02, CAS08] underwent longitudinal use of more than 2.5 years, and a successor system has recently been installed in a new building.

Context Sensing

The public displays we encountered employed a range of different technologies for obtaining context information, e.g. for detecting and identifying users, for recognising gestures or for locating users:

- *scanning of Bluetooth MAC addresses*: Bardram et al. [BHS06] scanned for users' Bluetooth-enabled mobile phones to determine the location of individuals. Müller and Krüger [MK06] planned to add similar functionality to their display deployments.

- *microphones*: LaughingLily [AS03] used microphones to determine the level of activity in a meeting situation.

- *cameras*: cameras were mostly used to detect user presence or for detecting gestures and gazes. Sparks [CLS+05] used cameras to track individuals. Sawhney et al. [SWS01] employed cameras to detect whether users were present and to determine whether users were merely walking past, glancing or looking at the displays for an extended period of time. Vogel equipped users with personalised reflective markers that could be tracked by a camera system aimed at identifying users and detecting gestures.

- *passive infrared sensors*: the Community Wall employed a grid of infrared movement sensors to detect user presence combined with a camera-based face-detection mechanism.

- *infrared*: four of the surveyed systems relied on infrared technologies to identify users. Thinking Tags [BMMR96, BMRS98] and Meme Tags [BMV+98] both used custom infrared communications to detect and identify communication partners, and to exchange information.

The Interactive Wall Map [McC02] and GroupCast [MCL01, McC02] employed an infrared badge system to detect and identify users in the vicinity of displays.

- *RFID tags and readers*: five systems used RFID tags and readers to identify users. Examples include BlueBoard [RTD04], AutoSpeakerID [MDS+04] and Hello.Wall [PRS+03, PSR+04].
- *swipe cards* were only used by Prospero [Con07], where they allowed users to indicate their presence to the display system.
- *username and password* were used by Prospero [Con07] as an alternative form of authentication.
- *USB sticks*: users of the Dynamo [IBR+03, BIF+04] were able to log on to public display surfaces using USB sticks. In addition, the sticks also served as transportation media for content.
- *iButtons* [Max08] were used by the Dynamic Door Displays [NTDM00] to authenticate users.
- *Context Toolkit*: Dynamic Door Displays [NTDM00] also retrieved information about users' locations from the Context Toolkit, which could in principle be used on top of arbitrary underlying sensor hardware.

Interaction Techniques

A whole range of different technologies are used in the context of public display systems to provide users with the ability to interact with public display content and applications. We can roughly classify the techniques encountered during the survey into the following categories:

based on mobile phone cameras: Ballagas et al. [BRSB04] employed optical flow processing of camera images to detect phone movement relative to the environment. The resulting system enabled users to use their mobile phones as pointing devices with three degrees of freedom. Mitchell et al. [MRS06] used cameras on mobile phones to capture and recognise visual codes that were shown on public displays.

activity and gesture recognition using environment-based cameras: Vogel et al. [VB04] employed wearable reflective markers in combination with a camera to identify users and to recognise user gestures. The display prototype produced by Sawhney et al. [SWS01] was equipped with a

network camera whose feed was processed using image differencing and face detection algorithms. The prototype was able to detect whether users were present in front of the display, whether these users were moving or stationary, and roughly which direction people were looking into. Using cameras the Sparks system [CLS+05] enabled users to select keywords by tapping them with their feet, causing a "pulse" to be sent to users with similar interests. Moreover, users were able to use their name tags as projection surfaces and perform gestures with them, e.g. to modify the level of detail of the projected information. In the context of the Interactive FogScreen [RDO+05] laser pointers were used as input devices, and these were tracked by external cameras.

gesture recognition using built-in accelerometers: the tangible displays trialed by Holleis et al. [HRKS06] featured built-in acceleration sensors. As a result, users were able to interact with the displays by performing gestures with them, e.g. by rotating a display into a specific direction.

physical buttons: both the Interactive Wall Map [McC02] and Online Enlightenment [HHTM04, TM06] used physical buttons as interaction devices. For the Interactive Wall Map, LED-topped button switches were used. Online Enlightenment employed buttons labelled with caricatures of lab inhabitants. By pressing one of the buttons, additional information about the selected inhabitant was displayed on a small LCD display.

touch sensitivity is by far the interaction technology most commonly used in the context of the surveyed public display systems. Of the 37 systems for which information abut the interaction technology used was available, 17 offered the ability to interact using touch-sensitive overlays. Most of these systems used off-the shelf touch-sensitive displays or overlays. Examples include the Community Wall [GMS03, SG02] and Dynamic Photos [Gre99]. CityWall [PSJ+07] on the other hand used non-standard touch interaction hardware reminiscent of Rekimoto and Matsushita's "HoloWall" technology [MR97, RM97], in which a combination of infrared lights and cameras were used to detect user gestures and provide multi-user, multi-touch surfaces.

keyboards and mice: Dynamo [BIF+04, IBR+03] provided wirless keyboards and mice as means for interacting with the display prototypes. Alternatively users were able to turn their own laptops into extensions of Dynamo work surfaces, enabling users to use their laptops' input devices for interaction with the Dynamo surface. A laptop also served as input device for the Opinionizer [BR03].

UIs rendered on users' mobile devices: public displays may employ dedicated user interfaces

on users' mobile devices (e.g. PDAs, mobile phones, smart watches) as proxy interaction devices. In these scenarios users interact with applications on their mobile devices to influence the behaviour of public display systems. Users of MobiLenin [SO05] interacted with the public display application through a custom-crafted Python application that was installed on their mobile phones. Information between the phones and the display application were exchanged using HTTP over GPRS. A smart watch was used as an interaction device for Berger et al.'s Symbiotic Displays [BKN+05]. PDAs served as interaction devices for both Jukola [OLJ+04] and Hello.Wall [PRS+03, PSR+04, SPR+03].

RFID tags and readers: Hello.Wall's [PRS+03, PSR+04, SPR+03] LED clusters were equipped with short-range RFID tags. Readers built into users' PDA-like ViewPort devices recognised these tags as users held ViewPorts close to LCD clusters to select them.

Display Hardware

Some of the early public display systems used off-the shelf *CRT monitors* as output devices. Examples include FLUMP [FWD+96] and the video link between Xerox PARC's Palo Alto and Portland offices [BHI93, GA86].

However, by far the most commonly used type of display hardware among the surveyed research systems are *flat-panel displays*. In their various incarnations as plasma or LCD displays, flat-panels were employed by approximately 40% of the surveyed systems. We encountered displays in a variety of sizes, ranging from small displays of merely a few inches in size (such as those used for Thinking Tags [BMMR96, BMRS98]), over regular desktop-size displays (e.g. in the case of Jukola [OLJ+04]) up to large 50 inch displays that were used, for example, by Vogel et al. [VB04]. A small proportion of the systems used displays that were integrated into PDAs or Tablet PCs: Greenberg used a tablet PC as display device for his Dynamic photos prototype [Gre99]. PDAs were used as electronic door plates in the first incarnation of the Hermes system [CFDR02].

The next most frequently encountered type of display hardware was represented by *projected or back-projected displays*, which were used in approximately 25% of the surveyed systems. Projected displays were, for example used by Sawhney et al. [SWS01], MobiLenin [SO05] and Sparks [CLS+05]. Examples for back-projected displays include the Interactive Fog Screen [RDO+05] where a unit mounted at ceiling generated a downwards flow of air and fog, which was then pro-

jected onto, and Hole-In-Space [GR80], where sidewalk-facing windows were back-projected onto. Other prototypes used commercially available back-projection units, such as the 72" rear-projected SmartBoard used by the Notification Collage [GR01].

Finally, approximately 20% of the surveyed systems employed custom-built display hardware. Obvious examples are ambient displays, such as the Water Lamp and Pinwheels created by Wisneski, Ishii et al. [WID+98, DWI98, IRF01] or Office Plant #1 [BM98]. Other examples include the Thinking Tags wearable displays [BMMR96] that used a row of LEDs as output device. Online Enlightenment [HHTM04, TM06] employed a custom-built display in the shape of a floor plan to represent the users' MS Messenger status. The display was made out of wood, multi-colour LEDs, push-buttons and a small textual LCD display.

Software Environment

A wide range of operating systems and programming languages was used to underpin the surveyed public display systems. In our survey we found systems running *standard Mac OS, MS Windows, and GNU/Linux operating systems*. Embedded systems, e.g. those used to control ambient displays, typically used *proprietary firmware* instead of an operating system. The WearBoy wearable display platform [LBF99] was based on Gameboy hardware, software and SDKs.

The programming environments encountered used to create public display software include programming languages, such as *Java, C++, TCL, PHP and Visual Basic*, and markup languages, such as *HTML and SMIL* (see Kray et al. [KKK05]).

Access and Organisational Deployment Context

We found that of the systems where information about the deployment was available, only about 10% were deployed into truly *public contexts*. Jukola and Fancy a Schmink [RLHS04] were both trialed in a *café/bar*, Pinwheels [IRF01, DWI98] were exhibited in a museum in Tokio. CityWall [PSJ+07] and Hole-In-Space [GR80] used sidewalk-facing store windows to create back-projected displays. MobiLenin [SO05] was deployed in a restaurant.

However, most public display systems were deployed into *semi-public contexts*. Roughly 25% of the display systems for which information was available were deployed into *academic research labs*

and offices. Examples include FLUMP, which was deployed into a central staircase at the Computing Department at Lancaster University, and Sawhney's Aware Community Portals [SWS01], a prototype of which was deployed into a combined open-plan hallway that also served as shared workspace and communal area. Approximately a further 20% were deployed into industrial research labs and offices. For example, the Learning Community Newspaper display prototype, which was deployed into a kitchen at Apple Computer's Learning Communities Group. Other examples include GroupCast [MCL01] and the Plasma Poster displays [CNDG03]. Moreover, about 20% of the display systems for which information about the deployment context was available were deployed into *academic conference contexts*, e.g. as part of a demonstration. AutoSpeakerID and Ticket2Talk [MDS+04] were trialed during the Fifth International Conference on Ubiquitous Computing (UbiComp 2003). The first user trial of the Opinionizer system [BR03] was carried out during a book launch party at the CHI'02 conference.

Finally, a number of *unconventional* semi-public *contexts* were encountered. These included a family home (TxtBoard [OHU+05]), a hospital ward and operating theatres (AwareMedia [BHS06]), a commons room in a school (Dynamo [BIF+04, IBR+03]), a pair of university dormitories (Telemurals [KD04]), and the office buildings of a large corporation (e.g. the RoomWizard [OPL03]).

Deployment Location

Not surprisingly, the public display systems surveyed were largely deployed in areas where they could be seen or encountered by more than one person. Popular deployment locations included:

- *common areas* where users would frequently linger, such as kitchens (e.g. the Learning Community Newspaper [HBL98] and the Virtual Kitchen [JVG+01]), coffee corners (e.g. GroupCast [McC02, MDS+04]), breakout areas (e.g. the Hermes Photo Display [CDF+05]), cafés and restaurants (e.g. MobiLenin [SO05] and the Community Wall [GMS03]), but also in shared offices and labs (e.g. Farrell et al. [Far01] and Online Enlightenment [TM06]).
- *areas with high amounts of through-traffic*, such as hallways (e.g. the Dangling String [WB96]), staircases (e.g FLUMP [FWD+96]), foyers (e.g. the Plasma Posters [CND+03, CND+04]), or shop windows (e.g. Hole-In-Space [GR80])

Sometimes the nature of the main application supported by a display mandated the use of *application-specific deployment locations*, e.g. in case of electronic door displays that were naturally deployed on or next to the relevant doors, or in the case of displays that were designed to support large gatherings of people and were consequently deployed in function rooms during conferences and receptions, such as the Opinionizer [BR03] or Ticket2Talk [MDS+04]. Wearable public displays, such as Thinking Tags [BMMR96], are another example for the use of application-specific deployment locations.

Audience

The audience of a deployed public display system strongly reflects the selected deployment location. In the case of deployments into fully public locations that we encountered as part of our survey, target audiences were *general members of the public* that were visiting cafés, bars and restaurants, passing by in front of store windows, or visiting museums. *Staff, students and visitors of academic and industrial research labs and offices* were targeted by the large number of display systems that were deployed into these organisational contexts.

Application-specific target audiences included staff at a hospital in the case of AwareMedia [BHS06], and members of a family in the case of TxtBoard [OHU+05].

Software Infrastructure

Connectivity and Level of Distribution. A large majority of the displays that were surveyed had some form of network connectivity. In some cases, this connectivity was used to obtain network-based content and information. For example, De Stijlistic Dynamics [HS03] used a network connection to obtain weather information from different cities. Online Enlightenment [TM06] interfaced with MS Messenger to obtain status information about registered users. Weiser and Brown's Dangling String [WB96] was connected to the group's local area network to obtain and visualise information about network activity. The Learning Community Newspaper [HBL98] received content from users via email.

Other systems employed network connections to allow users to use public displays to communicate with other people. IM Here [HRS04], for example, was connected to traditional desktop-based

instant messaging technologies to allow users to walk up to public displays and send messages and reminders to colleagues.

A number of display prototypes distributed parts of their functionality between different computational entities and used network connectivity for interconnecting these entities. For example, the Stone Garden [HS03] used a back-end server for downloading and processing earthquake-related information, which was then rendered and displayed by a client laptop. The software deployed on BlueBoard [RG01] displays was responsible for handling user authentication, interaction and for rendering output onto the displays. Back-end servers were used to hold content and a database of infrared badge IDs that were used for authentication. GAUDI [KKK05] employed a central server that was responsible for calculating route information and for disseminating adaptable interface descriptions to display clients. These client then further evaluated and modified the provided interface descriptions using local knowledge.

The distribution of functionality is often a necessity in the case of ambient displays that are controlled by embedded systems, as resource limitations frequently prevent these systems from carrying out more complex computational tasks. The Information Percolator [HHT99] used a server to expose a Java RMI-based API over the network to applications. The server processed API operations and communicated with a micro controller that was responsible for controlling the actual display hardware. Holleis et al. [HRKS06] used small embedded systems for driving their tangible situated displays. The displays were connected to a PC over an RF link. The PC was responsible for receiving incoming messages and for maintaining a stateful representation of the received messages. The PC also received information about detected gestures from the attached displays and modified its representation of message state accordingly, for example marking off a message as read if the associated gesture had been performed.

Reusability. In our survey we were also interested whether the different software infrastructures we encountered were designed with only specific applications in mind or whether they were designed as general purpose software infrastructures for supporting a wide range of public display applications.

The general-purpose scheduling infrastructure developed by Stahl et al. [SSKB05] allowed users to requests content to be displayed based on a set of constraints that were passed along with

requests. A central Presentation Manager was responsible for interrogating the system's central device manager to find displays matching the provided constraints and to create a system-wide plan for all presentation requests. Conflict resolution was achieved via a combination of display pre-emption, division of displays into multiple display areas, and re-planning.

The second example for a reusable software infrastructure can be found in the context of the Information Percolator [HHT99]. The system exposed a generic paint API that operated on a queue of bitmap images, each of which would be displayed for a duration specified by the user. The API that was remotely accessible using Java RMI provided operations for adding new bitmaps to the queue, as well as for interrogating and managing the queue itself.

McCarthy et al. reused parts of the UniCasts personal display system [MCL01] in some of their public display prototypes. The Interactive Wall Map [McC02] and GroupCast [MCL01] both accessed users' UniCast profiles to personalise and select content.

Finally, Ticket2Talk and AutoSpeakerID [MDS+04] shared the same application-specific infrastructure that was based on a back-end server holding user profiles in a database. The profiles could be queried by client displays if they detected users' RFID tags in their vicinity.

2.3.4 Evaluation Techniques

Researchers use a variety of techniques for evaluating public display systems. Moreover, frequently a combination of different techniques is used. In our survey we encountered the following methods:

- *anecdotal evidence and informal feedback*, i.e. reports about, for example, the effectiveness and usefulness of a system in the form of hearsay and anecdotes. Anecdotal evidence was used in the evaluation of systems such as MessyBoard [FFP02, FP04], FLUMP [FWD+96], and LaughingLily [AS03].

- *analysis of system logs*, used for example in the evaluation of the Community Wall [GMS03, SG02] and Fancy a Schmink [RLHS04]. During the evaluation of MessyBoard [FFP02] test subjects commented on historic snapshots of the board.

- *observations* were encountered during the survey in two different manifestations: in the first form observers were present on site, for example taking notes as the display systems were

used. This form of observation was, for example, used to evaluate Meme Tags [BMV+98] and Dynamo [IBR+03]. The second form of observation encountered involved the use of cameras to record interactions with displays. The recordings were then reviewed and evaluated at a later point in time. This method of observation was, for example, used during the evaluations of Jukola [OLJ+04] and the Opinionizer [BR03].

- *questionnaires and surveys* in their various forms (paper-based, email-based, web-based) were, for example, used for evaluating Huang et al.'s Semi-Public Displays [HM03], Dynamo [BIF+04], the Hermes Photo Display [CDF+05] and Ticket2Talk [MDS+04].

- *interviews* were encountered in structured (e.g. AwareMedia [BHS06]), semi-structured (e.g. CityWall [PSJ+07]) and unstructured form (e.g. IM Here [HRS04]).

- an *analysis of contributed content* was performed in the context of CityWall [PSJ+07].

- an *analysis of user profiles* was carried out to evaluate GroupCast and OutCast [MCL01].

- *studio critique*: Telemurals was partly evaluated using a series of studio critiques that involved experts from the areas of architecture and design.

- *measurements* were encountered in a variety of forms: Easterly [Eas04] measured plant growth to evaluate his plant display. Measurements of the weight of recycled vs. regular waste were used to evaluate Holstius et al.'s Infotropism displays [HKH+04]. Finney et al. measured "the period between a badge sighting being made and a users page beginning to appear" [FWD+96].

- *informal user trials* – typically involving students, colleagues or team members – were used for example by Mitchell et al. [MRS06] and by Vogel et al. [VB04].

- *journals and diaries*: in case of two of the surveyed systems (RoomWizard [OPL03] and Xerox PARC's audio/video link between Palo Alto and Portland [GA86, BHI93]), users were asked to keep journals or diaries of system use that were later used during the evaluation of the systems. Churchill et al. [CND+03] used diaries about users' existing practices of sharing and disseminating content to influence the design of the Plasma Poster system.

- *design walkthroughs and mock-ups*: finally, some authors performed evaluations using mock-ups or design sketches rather than finished public display prototypes, mainly to identify areas for improvement during early stages of the design process: in the case of Sparks [CLS+05]

informal feedback to mock-ups of the system was used to improve the design of the system. A "design walkthrough study" was performed by Sukaviriya et al. [SPK+03] for their steerable projector system for retail environments.

2.4 Public Display Systems for Commercial or Non-Research Use

Besides the numerous public display prototypes that have been developed in the context of research over the past decades, public displays have a long history of use in the commercial sector. One of the world's first public display systems was installed in 1928. The Motogram [New97, Sig08a, Sig08b] – later nicknamed the "zipper" – was installed on the facade of One Times Square building in New York. The display consisted of over 10,000 incandescent light bulbs arranged in a strip that was several feet high and several hundred feet long and surrounded the whole building. It was used to show scrolling headline news. The display is still in use today, but underwent a refurbishment in 1997, during which LEDs were installed to replace the display's incandescent bulbs.

The following decades saw the installation of a number of different display technologies in various public spaces where information had to be updated frequently. During the 1960s split-flap displays [Sol08] began to be introduced as departure boards in airports and large train stations. Scoreboards in sports stadium that were previously operated manually were replaced with electronic versions. One of the world's first deployments of electronic electronic scoreboards into sports stadiums took place in 1930 at the Michigan Stadium of the University of Michigan in the U.S. [Uni08]. The New York Yankee Stadium saw the introduction of its first electronic message board in 1959 [The08b]. Within the context of the Olympic Games, the first electronic scoreboard was used during the 1960 Olympic Games in Rome, with the first large video-capable displays and colour video displays being introduced at the Olympic Games taking place in Montreal in 1976 and in Seoul in 1988 respectively [Wat08].

The advent of affordable projectors and flat-panel displays in recent years has further fuelled a significant growth of the number of displays deployed into public spaces. Heathrow Airport's recently opened Terminal 5 alone features over 200 flat-panel displays for advertising purposes [The08c]. A total of 40 70 inch flat-panel displays were installed side-by-side in a 300 feet long

connecting tunnel at New York's JFK airport [Sou08].

2.4.1 PrintSign

Basic digital signage solutions have started to replace traditional static signs, particularly in locations where information needs to be updated relatively frequently. Typical application areas include hotel lobbies and corporate reception areas. "PrintSign" [Stu08], developed by the Appliance Studio, is a digital signage display in the shape of a traditional sign with stand. PrintSign allows users to simply "print" content onto digital signage displays. Once plugged into a network, PrintSigns appear as network printers. Users are able to use their own favourite applications to author content and to simply print the results onto one or more of the displays. Content printed to a display may comprise more than one page, in which case the content is displayed page by page in a loop. The print dialogue enables users to specify how long each page should be displayed for.

2.4.2 Sony Ziris

The very basic and simple digital signage functionality provided by PrintSigns is certainly sufficient for many applications. However, the types of content that can be displayed on PrintSigns is constrained by the use of the printer paradigm. As a result, only media types that can be represented on a sheet of paper can be displayed. This prohibits the use of dynamic content (e.g. audio or video), interactive content, or content that is to be updated dynamically (e.g. the current outside temperature, or plane departure and arrival information at an airport). Support for these more advanced media types is, together with additional monitoring and scheduling functionality, provided by more elaborate digital signage solutions.

"Ziris" [Son08], developed by Sony in the UK, is such a digital signage suite. The Ziris range includes software for content creation and scheduling, content transfer, content play-out and monitoring of software and hardware on Ziris play-out devices. Moreover, Sony also offers a small set of play-out (i.e. digital signage) hardware for use with the Ziris product range. Each administrative Ziris domain is powered by a central server running instances of the "Ziris Create" and "Ziris Manage" software and serving web-based front-ends to users.

Ziris Create

"Ziris Create" is the heart of the Ziris product range. It provides users with the ability to ingest content into the Ziris system and to schedule ingested content for play-out on associated play-out devices. Ziris supports a range of content types including videos, audio files and images. During playback, Ziris is also able to dynamically download and display HTML-based content from external web sites.

Content that is to be scheduled has to be arranged by users into multi-track timelines that are called "playlists". Rudimentary editing functionality offered by Ziris Create enables users to control the screen layout of content items in overlapping channels. To schedule a playlist users specify the dates content should be played at by selecting a start and end date and optionally a set of days of the week to further constrain the schedule. Users are able to specify that playback should either be performed all-day on the selected dates. Alternatively users may select a start and end time that is applicable for all dates. Finally, users specify which "channels" to schedule the content in. Channels represent pre-defined sets of Ziris play-out devices. Ziris Create automatically detects conflicts with other scheduled playlists and immediately rejects scheduling requests if overlaps exist.

Once scheduled, content is uploaded onto the individual play-out devices. In networks containing only small numbers of play-out devices, Ziris Create can be configured to handle content upload. Sony offers a separate product ("Ziris Transfer") for handling the transfer of content in larger display networks.

Ziris View

Public display hardware in Sony Ziris display networks typically consists of a flat-panel display or projector that is attached to either an off-the-shelf Sony play-out device, or a PC or Apple Mac computer that is converted into a play-out device by installing an instance of the Ziris View software. Ziris View is mainly responsible for playing back schedules created using Ziris Create. Additionally, Ziris View is capable of communicating with attached Sony display devices, such as projectors, using RS-232 connections and a proprietary communication protocol. Using this protocol, Ziris View and Ziris Manage are able to inspect the status of attached display devices and

perform a number of actions on them, such as switching display on and off, or changing volume levels.

Ziris Manage

Ziris Manage allows administrators to monitor status information collated from individual instances of Ziris View. Ziris Manage provides an overview of play-out devices and the attached display hardware present in the system. The status is visualised using iconic representations along with indications of the current status and eventual errors on a per-device basis.

Besides this overview, Ziris Manage allows administrators to retrieve and inspect detailed information about each play-out device and its attached display devices. For play-out devices this information for example includes the content currently played back on the device. In the case of a projector administrators are, for example, able to retrieve information about the remaining life time of the lamp and whether the projector is currently powered on.

Ziris Manage can also be used to perform simple actions on display devices, such as turning then on or off, switching display inputs, or setting the output volume. Some of these actions can also be configured to be performed automatically on a regular basis.

Finally, Ziris Manage provides a log of content as it was played out on each Ziris device, allowing commercial customers to produce audit trails, e.g. to comply with legal requirements.

2.4.3 Planar Systems

"Planar Systems" [Pla08a] is a provider of the "CoolSign" digital signage hardware and software products. Like Sony Ziris, this "CoolSign" software suite is divided into different products that in combination provide a complete solution for managing digital signage networks. "CoolSign Manager" provides the means for configuring and monitoring displays, as well as for ingesting content, creating playlist and play-out channels, and for scheduling content. "CoolSign Network Controller" represents a display network's central data repository server that stores both content and configuration and management data, including detailed play-out logs that are accessible using CoolSign Manager.

Ordinary Windows PCs can be turned into play-out devices by equipping them with the CoolSign player software. CoolSign is capable of rendering a variety of content types, including videos, images and interactive content for use on touch-enabled displays. Each display may be split into separate regions, each of which is addressable as an individual channel.

CoolSign also provides specialist solutions for:

- managing content transfer and distribution in large display networks ("CoolSign Transfer Hub"),
- dynamically monitoring and integrating information from external content sources (e.g. databases) into scheduled content ("CoolSign Data Watcher"),
- and for enabling non-experts to rapidly create content through the use of a templating system ("CoolSign Content Creator"). Templates are sold as separate products and may be developed according to customers' specifications.

Besides flat-panel and back-projected display hardware, Planar System also offer an integrated digital signage platform consisting of a flat-panel display and an integrated embedded PC that is running the CoolSign player software.

CoolSign is for example used to drive over 150 displays in Chicago O'Hare International Airport, including a 510 foot long display panel over the check-in counters consisting of 138 back-projected displays installed side-by-side.

2.4.4 Dynamax

"Dynamax" [Dyn08] is a provider of both digital signage software and hardware. Dynamax's digital signage software range comprises the "PointOfView NG^2 Player" and the "PointOfView NG^2 Enterprise Server". PointOfView Players can be used stand-alone without an Enterprise Server and can be run as screensavers, e.g. on interactive kiosks. PointOfView Players provide a web-based interface enabling administrators to schedule content using a calendar-based view and the concept of playlists. Screens can be divided into multiple regions, and besides standard media types, such as videos and images, PointOfView Players are also capable of visualising information sourced from RSS feeds. Rule sets are employed by the system to ensure that conflicting content

items (e.g. advertisements) are not played back-to-back or that related content is indeed shown back-to-back. The addition of an Enterprise Server to a display network provides administrators with a unified management interface for all displays in the display network. Enterprise Servers provide support for ingesting, scheduling and distributing content, and for display monitoring, reporting and configuration.

Dynamax also offer what they call a "digital signage in a box solution". This bundle consists of pre-installed software and hardware for one PointOfView NG^2 Enterprise Server and 16 PointOfView NG^2 Players, which are installed together with an audio/video matrix switch in a single 24u rack. Displays at the deployment location are connected to the rack using VGA-over-CAT5 cabling.

2.4.5 3M Digital Signage

While most digital signage solutions offer Web-based user interfaces for administering displays and scheduling content, "3M" [3M08] has taken this approach one step further. Major parts of the digital signage software are hosted directly by 3M on Internet-based servers. Play-out devices are normally connected to these servers. The servers also provide the user interfaces for administration tasks, content scheduling, content ingestion and content distribution.

The 3M play-out software is available as "Solo Edition" and as "Network Edition". The former does not support schedule creation involving multiple displays. Both editions can be installed on Windows XP PCs. Content scheduling in both editions is based on playlists and a calendar-based view. Displays may be divided into multiple areas and can be grouped if Network Edition is installed. Moreover, Network Edition provides means for monitoring and configuring displays, and for generating audit trails.

Display networks can be enhanced with "Network Edition Content Server" and "Network Operations Manager". Network Edition Content Server allows content to be distributed from and stored within the local network instead of requiring upload to and distribution from Internet-based 3M hosts. Network Operations Manager provides advanced monitoring and management functions for play-out devices. Network Operations Manager consists of a Web-based user interface and management agents that are deployed onto play-out devices.

Finally, 3M offers "Network Edition – Full Server". Full Server essentially provides the same functionality as that offered by 3M's Internet-based servers and is therefore, for example, suitable for use in display networks that do not have a connection to the Internet.

2.4.6 Netpresenter

While many digital signage solutions are used for delivering advertisements and other information to customers, "Netpresenter" [Net08] is mainly targeted at delivering content internally within an organisation, for example to distribute internal announcements to employees. "Netpresenter Player" software is available for use on large LCD displays, but crucially can also be installed as screensaver module on regular desktop PCs running MS Windows operating systems.

In the Netpresenter system, content is represented as "channels" that players subscribe to. Similar to MS Powerpoint presentations, channels comprise a sequence of slides with associated durations and transitions. Channels may be edited using "Netpresenter Editor", a stand-alone software product for use on MS Windows platforms. Alternatively, an extension to MS Powerpoint enables users to directly export MS Powerpoint presentations as Netpresenter channels. Moreover, a separate software product – "Netpresenter Message Server" – may be installed to provide a Web-based user interface for editing channels.

Besides means for authoring and publishing regular channels, Netpresenter also offers the ability to disseminate "urgent announcements or emergency alerts". These will override any other channels. Moreover, if Netpresenter is used as screensaver on desktop PCs, emergency alerts appear in pop-up windows even if screensavers are not active at the time. Emergency alerts can be authored and scheduled using Netpresenter Editor, MS Powerpoint, or via a separate Web-based interface.

2.4.7 AdSpace Networks

With advertising being one of the driving factors behind the deployment of commercial public display systems, companies have formed that operate large networks of public displays to create revenue by selling airtime for advertisements to customers. "Adspace Networks" [Ads08] is an operator of advertising screens for use in shopping malls in the US. At the time of writing, AdSpace

screens were deployed in more than 120 locations throughout the US with, according to AdSpace, around 10-15 displays in each location. Display hardware consists of 60" plasma screens in portrait orientation, coupled with stereo audio equipment.

Unlike solutions that only provide display hardware and software, such as Sony Ziris, AdSpace also offers services for content authoring. In addition, AdSpace also handles the scheduling of content on the displays. Content is programmed in a loop of a total length of six minutes. The loop contains a mixture of general advertisements by companies operating on a national scale and advertisements by local shops. Of these six minutes ten 12-second slots are provided for free to local businesses in each mall. Local stores compete for these slots on a weekly basis. Advertisements for these slots are designed and scheduled for free by AdSpace using the material provided by the winning shops.

2.4.8 InfoScreen

Like AdSpace, "InfoScreen" [INF08a] operates public displays and generates revenue by selling advertising airtime. InfoScreen displays are currently installed in 19 major cities all over Germany with a total of over 220 screens. A subsidiary of InfoScreen is active in the Austrian cities of Vienna, Graz and Klagenfurt. While screens in Germany mainly consist of projected screens installed in airports and bus and train stations, the majority of displays in Austria are 15"-17" flat-panel monitors situated inside trains, trams and buses. Most displays are not equipped with audio capabilities.

Unlike AdSpace, where programming consists almost exclusively of advertisements, InfoScreen schedules a mixture of editorial content – including news, trivia and short documentaries – that is provided by InfoScreen, interspersed with dedicated advertising slots. Additionally, InfoScreen operates two advertisement-only display deployments ("AD WALK") in a train station and a major airport. In each location five displays have been installed along a walkway. AD WALK provides customers with the ability to create advertising spots that are shown simultaneously across all five displays. Spots can be fully synchronised. Alternatively, all five displays can be treated as a single display surface, with different areas of the spot showing on different displays in a coordinated fashion.

InfoScreen offers content authoring services for its traditional screens and for AD WALK.

2.4.9 Blinkenlights

While public displays in the non-research sector are mainly used for commercial purposes this is clearly not always the case. Public displays are increasingly used for artistic purposes. "Blinkenlights" [Bli08] used the windows of an eight-story high tower block in Berlin, Germany, as pixels, turning the whole building into a large public display that could be used to show simple animated light patterns. Each window was equipped with a single 150W lamp that could be switched on and off using a relay. The relays were controlled by a set of three centralised computers. Members of the public were asked to design and contribute content for the displays, and a competition was held to award the best Blinkenlight patterns. The installation remained active from September 2001 to February 2002.

2.4.10 The BBC Big Screens

The BBC has installed a series of large outdoor displays in eight major cities in the UK [BBC08]. Screens show a mixture of programmed content that mainly consists of feeds from the BBC television channels, but also includes information and content provided by local communities, artists and non-commercial organisations. Moreover, the screens are regularly used for showing live broadcasts. Past events shown on the "Big Screens" included Proms in the Park, the 2002 World Cup and at the Manchester Commonwealth Games. Additionally, the screens have been employed in the past for interactive games that used the movements of individuals or groups of people as input. The movements were detected using a combination of cameras that are installed on top of the displays and additional software.

The first screen was installed in Manchester in May 2003. Each screen provides a display surface of about 25 square meters and is daylight visible. Using variable templates screens may be split into different areas, providing support for ticker bars (displaying text-based news and information), text boxes (displaying local information) and video feeds. Video feeds are typically accompanied by audio, but this feature is disabled during the night.

Each screen is managed by a "screen manager", i.e. dedicated BBC personnel who is responsible

for selecting and programming content. The screens are all partly financed by industrial sponsors. However, none of the screens is used to display advertisements.

2.4.11 E Ink

E Ink is a display technology that was developed by researchers at MIT in the late 1990s [JCT+97] and subsequently turned into a commercial product [E I08]. Displays are based on an arrangement of tiny capsules, each of which is filled with two types of differently coloured (typically black and white) and differently charged particles. If an electric field is applied to a capsule, particles of different colour separate and create the impression of an all-white or all-black pixel, depending on the orientation of the applied field.

The original aim of E Ink was to create a replacement for paper-based books. E Ink displays feature a resolution of up to 200dpi, a high contrast, a wide viewing angle and a relatively low power consumption (power is generally only required for changing the content on the displays). The displays are sunlight-visible, flexible, and relatively thin (0.4mm to 1.2mm).

E Ink displays have also found their way into the digital signage domain. Neolux [Neo08] and Midori Mark [Mid08] both offer digital signs based on the E Ink display technology. The displays are pre-programmed by the manufacturer with animations based on the customers' specifications.

Moreover, E Ink display technology has also been trialed in the form of passenger information displays in an inner-city train in Hamburg [IT06] and as a large 2.2 meters high and 2.6 meters wide public newspaper display [E I05] that was installed as part of the EXPO 2005 exhibition in Aichi, Japan.

2.4.12 Visual Planet

Finally, the use of displays in public and semi-public areas has led to the development of a range of specialist hardware, including computers in smaller than usual cases for use as playout devices, vandalism-proof and weather-resistant enclosures for displays and projectors, and customised interaction devices.

"Visual Planet" is a manufacturer of touch-sensitive foils that can be used to convert regular

LCD displays or back-projected displays into touch-enabled interactive displays. The foils are typically installed on the back of a pane of glass. The pane can then either be back-projected onto or mounted onto an LCD display. Glass panes may be up to 25 mm thick, depending on the type of glass that is used, and Visual Planet offers foils of up to 116" diameters.

2.5 Summary

In this chapter we have presented a survey of public display systems and applications, based on a historical review of research in this area. We have used the results of this survey to analyse key properties of the surveyed systems. The properties were classified into four categories: objectives, content, deployment and evaluation techniques. Following this analysis we have presented an overview of public displays and digital signage products for commercial and non-research use. In the next chapter we will, based on the analysis performed in this chapter, investigate the requirements for a distributed systems infrastructure and accompanying scheduling API for controlling the presentation of content on public display networks.

Chapter 3

Requirements

3.1 Introduction and Process Followed

In the last chapter we presented an historical review and analysis of public display systems. In this chapter we focus on the requirements for a software infrastructure to control the presentation of content on medium scale public display networks to enable them to be used to support a wide range of experimental applications. The requirements presented were drawn from three main sources: an initial brainstorming phase, during which members of the e-Campus team collected potential use cases for the e-Campus system; a collection of experimental systems (probes) that were evaluated using short-term deployments to investigate specific aspects of public display system design; and finally, an analysis of the requirements of existing public display systems and applications, building on the survey presented in chapter 2. A concise set of requirements is presented towards the end of this chapter and these requirements represent the foundation for the design of the distributed systems infrastructure and APIs presented in this thesis.

3.2 Initial Brainstorming Process

The first step of the requirements capture exercise involved considering a number of use cases based on fictitious pieces of content. Besides research-related content in areas, such as context sensitivity

and spatial reasoning, we deliberately aimed to consider day-to-day content that was not research-related and would be shown on displays whenever they were not in use for experimentation.

3.2.1 Use Cases

The news every hour on the hour. This use case involved showing an hourly news update on all displays in the display network. The news was to be shown exactly on the hour for a few minutes. We had initially foreseen the news to be delivered in the form of text, but later on also considered showing variable length news videos.

The weather hourly right after the news. In addition to the news we thought of showing the weather forecast right after the news.

Content as screensaver. This type of content represented low-priority background content that would be shown whenever no other, more important, content was available.

The bouncing ball. The bouncing ball use case was specifically designed to make use of the geometry of Lancaster University's main walkway that traverses the whole campus from north to south in a more or less straight line. The e-Campus project was planning to deploy a number of displays along the sides of this walkway. The "bouncing ball" use case foresaw an application that created the illusion of a ball bouncing from display to display along the walkway. The ball would either be able to start on one end of the walkway and bounce independently down the walkway, or it would follow a person as they walked along the walkway, always appearing at the display that was closest to the person. The use case was interesting to us since it represented a personalised, context-sensitive application whose engineering might in the future be coupled with mechanisms for reasoning about geometric relationships between different displays, and also between displays and users.

3.2.2 Derived Requirements

The set of use cases presented above were subsequently used to help derive an initial set of requirements.

Support for research-related and non-research-related content. Medium-size public display networks are expensive in terms of hardware and installation costs, but also require a constant stream of funding for maintenance and repairs. It is therefore unlikely that such display networks will be created solely for the purpose of supporting research. Instead we expect a dual model in which airtime is shared between research-related content and other content that is not research-related, including electronic art, announcements and news bulletins. By opening up the display network for content that is not research-related, additional stakeholders may be motivated to contribute to the costs of public display installations.

Support for showing content based on the time of the day. This requirement was mainly derived from the news and weather report news cases in which content had to be shown at certain times during the day (e.g. on the hour).

Support for bulk operations. The fact that the news and the weather forecast were to be shown on every display in the display network, called for the ability to schedule content on groups of displays without having to individually schedule content separately on each display.

Support for relationships between pieces of content. The weather forecast, while being a separate piece of content that is scheduled separately from the news, is nevertheless linked with the news: it is to be displayed "right after" the news. Support is therefore needed for scheduling content in relationship to other pieces of content, or to link multiple pieces together into atomic units that are always played out sequentially and in the same order.

Support for priorities and preemption. The introduction of background content as described in the screensaver use case requires users to be able to assign different priority levels to content. Moreover, higher-priority content needs to be able to "preempt" displays from lower-priority content, for example making it possible to replace the screensaver with more important content once this becomes available, or to enable the bouncing ball application to preempt displays as it follows a user along the walkway.

Support for context-sensitivity and personalisation. In the case of the bouncing ball following a person down the walkway, the position of that person has to be sensed and used as input to the process of reaching scheduling decisions. In general we found that we would need to be able to influence scheduling decisions based on external events, such as user interaction or input from various types of sensors.

Support for spatial and geometric reasoning. Knowledge about the location and orientation of individual displays is required to construct the bouncing ball application. The application is restricted to displays that are situated along the walkway. To create the illusion of a ball bouncing strictly into one direction, the displays along the walkway have to be ordered according to their location. Moreover, the orientation of a display determines, for example, whether the ball should enter the display from the left and exit to the right or vice versa. Support is therefore needed for selecting and showing content based on geometric properties.

3.3 Requirements of Existing Public Display Applications and Systems

The following set of additional requirements was derived from the survey and analysis of research into public display systems (see chapter 2). We focused specifically on requirements related to the presentation of content and the evaluation of experiments.

Arbitration between conflicting pieces of content. Most of the experiments that we surveyed in chapter 2 were based on single applications that were shown exclusively on the displays. An exception can be found in the work by Stahl et al. [SSKB05] that explicitly investigated solutions for dividing display airtime and screen real-estate between multiple independent pieces of content. It is clear that a distributed systems infrastructure enabling the use of public display networks as shared resources for research will have to provide means for arbitrating between conflicting content that may be part of different experiments.

Scalability up to a few hundred displays. While the vast majority of the surveyed pieces of work used less than ten displays we also encountered some pieces of research that attempted deployments involving larger numbers of displays over longer periods of time: 40 electronic Hermes door displays were recently deployed [CAS08] in the Computing Department at

Lancaster University, and we expect public display networks to grow in scale over the next few years as the price of display hardware continues to fall.

Multiple content sources. Public display deployments often rely on the willingness and ability of users to contribute content. Moreover, in this thesis we envision a public display network that acts firstly as a shared resource for experimentation for a community of researchers, and secondly as an outlet for general day-to-day content related to the local community. Content for both experimentation and day-to-day operation is therefore likely to come from multiple content sources, involving researchers and members of the public.

Support for standard content types and proprietary applications. We have seen in chapter 2 that content is often ingested in the form of standard content types, including HTML-based web content, images and videos. In some cases content is then viewed using off-the-shelf viewer applications, e.g. in a web browser in the case of FLUMP [FWD+96]. However, in other cases custom applications are created, e.g. to allow users to fine-tune the layout of content in the case of the Notification Collage [GR01] or to add the ability to interact with and annotate content in the case of the Plasma Posters [CND+03, CCD+04]. An open network of public displays as infrastructure for research will therefore have to provide support for both standard content types and proprietary applications.

Support for on-demand content. In systems, such as IM Here [HRS04] or BlueBoard [RTD04], airtime on the displays was shared between background content that was shown in continuous loops and interactive applications that were made available to users on demand as a result of user interaction, e.g. if users touched the display surface. A distributed systems infrastructure aimed at providing control over the presentation of content will therefore have to provide support for content that is to be presented on-demand as a result of external events.

Support for interactivity. As we have outlined in chapter 2, researchers have in the past conducted experiments using both interactive and non-interactive content and applications. Moreover, the interactive research prototypes surveyed used a wide range of interaction techniques, including mobile phones, fixed cameras, touch overlays, and traditional keyboard and mouse combinations. When creating a public display network for research one will therefore have to ensure that content for the purpose of research will be able to interface with a wide range of interaction devices.

Support for context-sensitivity. The survey of research into public display systems showed that experimental public display systems make use of context information that may be obtained from a variety of sources, including Bluetooth scans, cameras, RFID technologies, and passive infrared sensors. Public display networks for research will therefore have to provide the means for adding various context sources and allow experimental applications to interface with these sources.

Support for the orchestration of content across displays. Some of the applications encountered during the survey required content to be orchestrated across different displays. In the case of navigation applications such as GAUDI [KKK05] that use a set of displays that are distributed throughout a building or campus to guide users from their starting point to their intended destination, content has to be presented on these displays in a sequential and timely manner. Each display is required to show navigation information as soon as the targeted users approach the display and enter visual range. If information is shown too late ("turn right here"), users may take the wrong direction.

GAUDI solved this problem by employing a central server that, once a request for navigation assistance from a user had been accepted, immediately distributed the resulting navigation content to the GAUDI displays in the network. Once the displays had received this content they displayed it as their only content until they received new information from the server.

However, on a public display network in which multiple experimental applications compete for airtime on the displays, additional support for the orchestration of content across multiple displays will be required. The underlying distributed systems infrastructure will, for example, have to be able to ensure that all displays needed for showing navigation-related content can be acquired in sequence and at the time needed.

Support for audit trails. The collection of data is a crucial element for evaluating research hypotheses. In our initial survey of public display systems and applications we encountered a variety of methods that researchers used for collecting data. Some of these methods required researchers to collect data manually, e.g. via interviews or observations. Other methods relied on information that was automatically collected by the systems and applications themselves. Examples include interaction logs, user profiles and measurements of system parameters. A distributed systems infrastructure for supporting experimentation on public display networks therefore has to provide interfaces that enable researchers to access

and log such information, but may also provide means for simplifying manual data collection, for example by installing cameras next to each display to allow researchers to observe user interaction with research-related content.

Moreover, in a public display network where display airtime is shared between multiple experiments and background content, data collection should not only be limited to application-related data, such as interaction logs, but should also allow researchers to collect information about when their content was shown, on which displays it was shown, for how long it was shown, and on what grounds certain scheduling decisions were made.

3.4 Probes

We decided to further investigate the requirements arising from sharing a public display infrastructure between different applications and pieces of content through a series of probes that we describe in this section.

3.4.1 Installation 1: WMCSA 2004 Conference Signage

Overview

Our first technology probe deployed a digital signage solution at the 6th IEEE Workshop on Mobile Computing Systems and Applications, WMCSA 2004. The WMCSA system consisted of four public displays stationed outside each of the entrances to the main auditorium and demo room. The displays provided a rolling display of information for delegates tailored to the display's location (proximity to ongoing conference activities) and the time of day. Each display was able to show information relevant to the talks being presented in the adjacent rooms, about activities in the wider locale, and navigation symbols directing delegates to refreshments at appropriate times of the day.

Displays were each split into three areas that content could be displayed in independently: a ticker-bar at the bottom, a vertical side-bar and a main content area that covered about 2/3 of the screen real-estate. The side bar and the main content area could be combined and used to display larger pieces of content. Moreover, the displays were all interconnected via a local network,

allowing us to synchronise content across the displays on a per content item basis.

One of the key issues we sought to explore with the WMCSA system was how to simplify the process of injecting content into the system and of mapping that content to displays. We did this by exploiting a separation of concerns: authors could create content items (images, web pages, RSS feeds and videos) and request these to be mapped dynamically to the network of displays using a constraint-based scheduler. The author could specify a range of constraints for each piece of content in a scheduling request, including:

- temporal constraints (i.e. the earliest start time and the latest finish time of the content)[1]
- spatial constraints, i.e. which display area(s) to use
- duration
- whether the content item should repeat
- the set of displays to target
- the required coordination between the displays
- the priority of the content

The content of the WMCSA system was therefore reduced to a set of scheduling requests: some content had a requirement for synchronisation and temporal coherence (e.g. arrows directing delegates to lunch), whereas other content ran for the duration of the paper sessions but only on single (uncoordinated) displays (see figure 3.1). A scheduler associated with each display observed these requests and attempted to construct a timeline for the display that best matched the requested set of constraints. Where content was required to be synchronised across displays, a distributed agreement protocol was used to converge on a mutually agreeable time. The priority system allowed us to easily introduce 'background content' of a low priority.

Once content had been programmed into a time slot, only higher priority content could displace it (reducing the overall search complexity for scheduling each job). Consequently, the order in which the scheduling requests were submitted could affect the resulting schedules in cases where

[1] In our probes these constraints were simply expressed as two absolute times. In more sophisticated systems, formal notations for describing constraints could be used, e.g. based on James F. Allen's work [All83]. Moreover, a discussion of different methods for describing and modelling multimedia presentations can be found in [Lit94].

Figure 3.1: A WMCSA situated public display.

more than one piece of content's constraints were satisfied. The timeline based system allowed us to introduce content items in advance of the point at which they were due to be displayed (we anticipated being able to use the timeline in later iterations to generate electronic program guides and feed deadlines into content caching and distribution mechanisms).

Design Reflections

Following our live deployment, we reflected on the efficacy of our design. We found that the system was adequate for scheduling situated content on a small number of displays in the environment it was intended to operate in, i.e. as a digital signage solution at a workshop. However, there were already a number of concerns, especially relating to the suitability and flexibility of the constraint-based scheduling approach for future deployments:

Scalability. The complexity of planning timelines using a constraint-based approach is tractable with small numbers of displays, content items and relatively short timelines. However, the search time increases in polynomial time as these factors grow. As a result we had to conclude that this approach wouldn't scale for larger deployments.

Types of constraints. We found that the types of constraints we were able to support in this first prototype quite often did not allow us to tune the presentation in exactly the way we intended. For example, we found ourselves unable to schedule content "right after" another content item had finished without tying both content items down to exact start and end times.

On-demand scheduling and interactivity. We planned to introduce on-demand content in the future that would be shown as a direct result of user interaction and that might itself be interactive. We found that in our current model of constructing timelines based on constraints too great a degree of sophistication was required in engineering the constraints to adequately allow us to interrupt the schedule to insert the interactive content elements: the fact that we had to specify a duration for each piece of content, determining exactly for how long a piece of content would be displayed, made it impossible to deal with spontaneous, interactive applications that users would want to interact with for an unspecified amount of time.

Spatial and geometric reasoning. Much of the WMCSA content was location (and in the case of the 'navigation arrows' orientation) sensitive. For this deployment we were able to 'hand craft' the constraints and content to ensure that the correct information was presented on the appropriate display given its position and orientation (e.g. the navigation arrow directing delegates to lunch pointed in the correct direction!). However, we realised that for a larger-scale deployment additional abstractions would be needed that would allow us to directly schedule content based on spatial and geometric constraints, such as 'on all displays in front of meeting room 1'.

3.4.2 Installation 2: Brewery Arts Centre VE Day 60th Anniversary Exhibition

Overview

The second installation took place at a local arts centre (The Brewery Arts Centre in Kendal, Cumbria) as part of their 60th Anniversary VE Day celebrations. The installation was one element of an interactive exhibition of local wartime memorabilia and consisted of four main components: a

Figure 3.2: Photograph of the Brewery exhibition space with the projected displays visible in the background.

set of three large projected public displays (see figure 3.2), a video diary booth, a web based diary, and 'the Kirlian Table' (an interactive art exhibit created by a local arts collective). The public displays showed a series of news footage and radio broadcasts evocative of the era, interspersed with images captured from the interactive table surface and video diary entries contributed by visitors to the exhibition. The video diary entries were also made available via a local web based content management system.

The Brewery deployment took place only a few months after the WMCSA deployment. We had used this time to address issues with the underlying communications infrastructure underpinning the network of displays and schedulers, but apart from a few bug fixes roughly deployed the same constraint-based schedulers that had underpinned the WMCSA deployment. Nevertheless, the brewery deployment represented a significant evolution of the requirements placed on our software architecture: instead of entirely pre-scripted and orchestrated content, the system now needed to support the dynamic introduction of new content, such as recordings from the video diary booth and photos taken from the Kirlian Table, into the live system. As a result we found that we had a glut of content (hours of video and audio, large numbers of still images and an increasing body of visitor contributed content) to choose from, requiring the introduction of random content selection (e.g. 'schedule an item of content from this pool'). We tried to achieve this by creating proxy content items, i.e. PHP scripts hosted on a web server that randomly selected a piece of content from a pool of items and returned it in the HTTP response. This whole process was completely transparent to the schedulers who were now simply scheduling proxy content items identified by URLs instead of "real" content items identified by URLs.

Design Reflections

The Brewery deployment was particularly illuminating: we found we had several unexpected requirements:

Orchestration of content. As part of an exhibition there was a need to use the public displays in an aesthetically pleasing way, this meant creation of schedules that had nice temporal and spatial characteristics. Our schedulers had so far attempted to satisfy constraints by creating timelines that best matched the requirements of individual content items. The requirement for precise orchestration of when and where content would appear in relationship to other content items on the same or other displays required that we increased the determinism of the scheduling process. We achieved this by modifying the schedulers to work in terms of absolute time, but the manual creation of schedules based on absolute times was obviously not going to be tractable in installations with a larger number of displays and content items.

Support for randomly chosen content of varying length. We found the need to handle randomly chosen content had a further impact on the system: the media content (audio, video etc.) was not necessarily of the same duration. This meant that the timeline agreed by the schedulers needed to adapt on the fly as individual content items were chosen (to avoid 'dead air' or truncation of playback). Moreover, matters were further complicated by the need to synchronise presentation across the displays. In contrast, our time-based scheduler was designed to show each piece of content for a previously specified period of time.

Support for on-demand content. During the first few days we also noted that our schedulers were missing a 'schedule now' functionality that would have allowed us to override the generated timeline and start immediate playback of content on one or more of the displays. Due to the emotional nature of the subject (people's war time memories), users who had left entries in the video diary felt that these provided important insights to other people and were therefore very keen to see their own recordings played back on the projected displays. We repeatedly attempted to explain to users that video diary entries were moderated and added to the system for playback at the end of each day, and were then displayed randomly during the following days. However, people were clearly disappointed if their video didn't show up on the big screens immediately after they had recorded it: one elderly lady brought along a chair, and sat down in front of the screens to wait until her video would appear. In

the end we had to manually modify one of the PHP scripts that was responsible for randomly selecting content from the pool of diary entries so that it would deterministically select and return that lady's video diary entry.

3.4.3 Installation 3: The Underpass

Overview

The last in our series of technology probes was deployed in an underground bus station on campus (called 'the underpass', see figure 3.3). The aim of the installation was to enrich this space by providing a mixture of information and interactive content to people waiting for buses. In contrast to our other technology probes, the installation in the underpass was intended to be a long term deployment, i.e. lasting at least several months, possibly up to a few years. To fit the physical dimensions of the space, it was decided to deploy three large-scale projected displays that would be aligned side-by-side. We also wanted to be able to either use each of the projection surfaces independently or in combination as wide-screen displays of 2 or 3 displays.

Figure 3.3: Metamorphosis on the Underpass screens. (Photo: David Molyneaux)

The initial focus was to employ a mixture of content, including artistic material, textual information and videos. Consequently the installation opened up with a piece of interactive art (called

'Metamorphosis' [Wel05]) that consisted of a set of 3 videos that were to be shown side-by-side in a coordinated fashion and were controlled by a Max/MSP [Cyc08] script. Metamorphosis also interfaced with a small number of sensors that, when triggered by passing traffic, influenced the behaviour of the artistic installation.

Being based on Max/MSP, commercial software currently only supported on either Windows or Mac operating systems, the commissioned piece of content was incompatible with parts of the scheduling system we had been using for the previous technology probes, as those parts were heavily tied into the X Window System [SG86]. We were therefore forced to run Metamorphosis on a set of four dedicated Mac Mini machines that were independent from the rest of the installation.

To support additional content besides the artistic installation, we deployed a PC with a multi-headed graphics card that allowed us to either render different pieces of content on each head or render content that spanned across two or more heads. A Sierra Pro XL audio/video matrix switch [Sie08] and an embedded AMX NetLinx controller [AMX08] were put in place to allow us to switch between content rendered on the PC and content rendered on the dedicated Mini Macs.

Design Reflections

Support for non-standard hardware setups. The need to support an interactive piece of art that would not integrate with other parts of our experimental scheduling system and that could only be hosted on separate machines made us aware of the requirement to be able to support non-standard hardware setups that, for example, require video inputs on displays to be switched to different sources during runtime.

Support for dynamic display configurations. The AV matrix switch made it possible to switch any video source to none or one or more projectors. As a result the notion of what computer and projector combination constituted a display could be changed within a matter of seconds. This meant that our previous model for addressing and thinking about displays, in which each video output device, i.e. a monitor or a projector, was permanently attached to exactly one video source, was clearly no longer valid. Moreover, scheduling a piece of content now not only involved handling contention between different pieces of content on each display, but included the need to arbitrate between different machines (PC or Mac Mini) competing for a projector.

Support for scheduling multiple pieces of content as atomic unit. Although Metamorphosis was physically distributed onto a number of machines, the videos were designed to be shown simultaneously and side-by-side on all three projectors. It therefore made sense to only schedule Metamorphosis as an atomic unit that would be made visible if and only if all three projectors were available.

3.5 Consolidated Set of Requirements

We have collated the requirements from the three different sources – use cases, existing applications, and probes – and distilled them into the following list of requirements for a software infrastructure to control the presentation of content with the aim of supporting research and experimentation on medium-scale public display networks:

R1: Scalability up to a few hundred displays. We are targeting public display networks of, for example, campus-size, but do not aim to provide support for nation-wide or global display networks.

R2: Support for research-related and non-research-related content. As outlined before, allowing additional stakeholders to access a public display infrastructure and contribute content might provide reasons for these stakeholders to contribute to the cost of installation and maintenance. As a result, content for the public display network will not only be sourced from stakeholders within the research community, but also from members of the general public. Based on the analysis of existing research into public display systems and applications, we expect research content to be mainly HCI-related.

R3: Arbitration between conflicting pieces of content. With multiple researchers using a public display infrastructure as a shared resource, and with members of the public contributing additional content, content items will be competing for airtime on the public display network. Means for arbitrating between different content items will therefore have to be provided.

R4: Support for a wide range of scheduling criteria. We have seen that a multitude of different criteria is used by researchers to determine when a particular piece of content or application should be displayed. Trivial examples include dates and time ranges. In other cases

content is to be made visible based on external events, that may, for example, be received from interaction devices or sensors in the environment. Navigation applications may require the evaluation of spatial or geometric constraints to determine when and on which displays a piece of content should be shown. Content may have to be displayed 'right after' another piece of content has been shown. As a result a software infrastructure to control the presentation of content to support research and experimentation should be able to support a wide range of scheduling criteria. Moreover, researchers should be able to add new criteria if necessary.

R5: Support for on-demand content. The software infrastructure will have to be able to show content in an immediate fashion, e.g. as a result of interactions with the displays.

R6: Support for bulk operations. The scale of the targeted public display networks mandate support for operations that affect not a single display, but instead a whole group of displays. For example, making a piece of content visible on 10 displays should ideally require little more effort than making it visible on a single display.

R7: Support for priorities and preemption. Priorities will allow the introduction of background content that is played whenever a display is not in use for research purposes. Moreover, priorities can be used to ensure that background content can be superseded once higher-priority content becomes available.

R8: Support for context-sensitivity and personalisation. As we have already seen above, context information may be used to decide when to show particular pieces of content. However, experiments are likely to be undertaken in public display networks in which content and applications themselves react and adapt to external context parameters. To obtain this context information, these applications may require access to native APIs that are not accessible from within sandboxed environments, such as web browsers or Java virtual machines.

R9: Support for interactivity. The argument for interactivity is similar to that for context-sensitivity and personalisation, and so are its implications. As we have outlined above we expect researchers to create experiments involving interactive applications. These applications will have to be able to obtain information from interaction devices, and may therefore require access to native APIs. Moreover, interactive applications are typically to be shown for at least as long as users interact with them, i.e. the period for which an interactive application should be displayed can often not be determined in advance.

R10: Support for content of dynamic length. It is often difficult to predict the duration of user interactions with interactive content. As a result, the duration of such content can often not be determined in advance.

R11: Support for standard content types and proprietary applications. We have seen that public display networks are most likely to be showing a mixture of general, community-provided content and content for research-purposes. As we have also outlined, community-provided content and parts of the research-related content are likely to be supplied using standard media formats (HTML-based web content, images and videos). However, we have also seen that a proportion of the existing applications in the research domain were custom-crafted applications that, for example, allowed researchers to fine-tune the user interfaces or to interface sensor platforms and actuators through native APIs. As a result, our software infrastructure will have to provide support for showing both standard content types and proprietary applications.

R12: Support for orchestrated performances. The probe at the Brewery Arts Centre revealed the need for the ability to arrange content into precisely defined sequences, possibly spanning multiple displays. A similar requirement could be found in the context of the 'weather after the news' use case.

R13: Support for atomicity and isolation. The probes in the Underpass and at the Brewery Arts Centre had both brought forward the need to display content in an aesthetically pleasing manner. It is therefore important that intermittent system states or failure states are not made visible to observers (isolation). Moreover, in the context of Metamorphosis in the Underpass we had encountered content that was distributed across multiple displays, but that nevertheless had to be treated as atomic unit, i.e. either all pieces of content were made visible, or none at all (atomicity).

R14: Support for audit trails. If researchers are to use shared public display networks as platforms for experimentation, means have to be provided for researchers to observe their experiments, particularly in the presence of other, potentially conflicting content. The software infrastructure underpinning such networks should therefore ensure the observability of scheduling actions. It should also allow interaction events and context events to be monitored and logged. Moreover, the infrastructure may aid users with the task of data collection by providing means to observe user interactions with the displays, e.g. through the provision

of a network of cameras.

R15: Support for non-standard and dynamic hardware setups. The Underpass has provided us with an example of a public display installation that goes beyond the traditional public display hardware consisting of a computer and a flat-panel display or projector. Besides traditional public display hardware, the software infrastructure presented in this thesis should therefore be able to support such non-standard hardware setups, e.g. displays with multiple inputs that can be switched to show output produced by different computers. Specifically, the software infrastructure should take into account that such changes may be performed at run-time, for example leading to conflicts between content on different computers that are attached to the same physical display hardware.

Looking back at the public display systems surveyed in chapter 2 it becomes clear that none of those systems is able to meet the full set of requirements. Many of the surveyed research systems were designed to support specific experiments and therefore lack the flexibility to support a wider range of experimental applications and content. Ambient display systems obviously lack the ability to support the wider range of content types and are therefore not appropriate as platform for general-purpose research. Systems, such as Dynamo [IBR+03, BIF+04] were purely interactive and did not provide any facilities for scheduling different pieces of content. Other systems, such as BlueBoard [RTD04] or the Plasma Poster displays [CND+03] were able to show competing content items in a continuous loop, but clearly lacked the ability to schedule content using a wider set of criteria.

The only software infrastructure that we encountered in our survey of research systems that was designed to be reusable was developed by Stahl et al. [SSKB05]. However, that software infrastructure was limited to a scheduler that planned presentations using a small set of scheduling constraints, an approach that we had experienced during our WMCSA deployment as being too inflexible to support a wider range of experimental content. None of the research systems surveyed were found to provide generic support for the orchestration of content, or for atomicity and isolation.

The commercial digital signage software infrastructures that we surveyed mostly scheduled content based on timelines that were manually constructed by administrators in advance. However, as we have learned during the evaluation of our probes, timeline-based scheduling prohibits

the introduction of content of dynamic length, and is therefore unsuitable for a general-purpose software infrastructure for experimentation in public display networks. It is also clear that this timeline-based approach does not meet our requirement to support a wide range of scheduling criteria. The only commercial digital signage system that we encountered that did not schedule content based on timelines was Netpresenter [Net08] where displays subscribed to single channels, each of which represented sequences of slides. However, this model provided even less control over the play-out of content than the timeline-based model. In addition, the closed nature of the surveyed commercial products makes the use of non-standard content types and proprietary applications very difficult.

3.6 Summary

In this chapter we presented a set of requirements for designing and building a software infrastructure to control the presentation of content on medium scale public display networks. The requirements were drawn from three main sources: an initial brainstorming phase, a collection of experimental systems (probes) and an analysis of existing public display systems and applications.

Based on these requirements we will in the following chapter describe the computational model, including key abstractions and an API that enables researchers to execute experiments on public display network.

Chapter 4

Computational Model

4.1 Introduction

In the previous chapter we presented a set of requirements for designing and building a software infrastructure to control the presentation of content on medium scale public display networks. In this chapter we present a computational model and API based on these requirements. The chapter starts by providing an overview of the hardware environment constituting a typical public display network. This overview is followed by a discussion of the requirements presented in the previous chapter and their impact on the design of our computational model. We then provide a detailed description of the entities of the computational model, and the operations and key properties that these entities expose. The operations are exposed to developers in the form of a distributed API that enables developers to instantiate content on public displays and to control its visibility. We describe this API towards the end of the chapter.

4.2 Hardware Environment

As we have discovered in chapter 2, public display systems use a range of different display hardware and computational hardware. Omitting the classes of ambient displays (which typically use custom display hardware, making them unsuitable as general-purpose devices for public-display research),

purely non-visual displays (e.g. speakers), and displays without discrete output locations (e.g. using smart wallpapers or steerable projectors to create displays of arbitrary sizes in arbitrary locations) we have identified a basic set-up that is common across most public display systems. A public display typically consists of the following elements (depicted in figure 4.1):

- *one or more video sources*, i.e. a piece of equipment that is capable of producing a video signal. A video source may, for example, be a provided in the form of a workstation computer or laptop, but can also be provided by cameras or broadcast equipment, as it was for example used for the realisation of Hole-In-Space [GR80]
- *one or more video sinks*, i.e. devices that are capable of rendering a video signal so that it can be viewed by users. Typical examples for video sinks encountered in our survey in chapter 2 are computer monitors, flat-panel displays and projectors.
- *zero or more audio sources*. Often the devices that act as video sources also act as audio sources. However, configurations are possible where audio sources are separated from video sources, e.g. in the case of a portable music player that is used to enhance a public display application with background music.
- *zero or more audio sinks*, which are typically speakers.
- *zero or more interaction devices*, such as simple push-buttons, touch-sensitive overlays, mobile devices, or cameras. Interaction devices may either be associated with specific displays (or their computers), or may be shared between multiple displays. For example, in the Underpass we have deployed a single Bluetooth dongle that can be used to receive input from Bluetooth-based interaction devices. The receiver is attached to one of the computers, but interaction events received by the dongle are published to all public display machines in the Underpass.
- *zero or more context sources*. Context information provided by these sources may be used, for example, for personalising content. A location system might, for example, act as context source, providing information about the presence of users in the vicinity of displays.

In most cases the mappings between sources and sinks are statically configured and this would, for example, be the case in a public display system consisting of a single workstation computer and a flat-panel display. However as we have seen in the context of the Underpass, systems exists that

allow a dynamic mapping of video and audio sources to video and audio sinks. In these cases, a *mechanism for switching between different sources* is required. In the most trivial case, switching may be performed simply by unplugging one source, and plugging in a different source. Sinks may also provide multiple alternative inputs: computer monitors are often equipped with two video inputs and provide buttons that allow users to toggle between these inputs. Other examples include KVM switches or professional audio/video matrix switches [Sie08] that provide a number of inputs that can then be mapped onto one or more outputs.

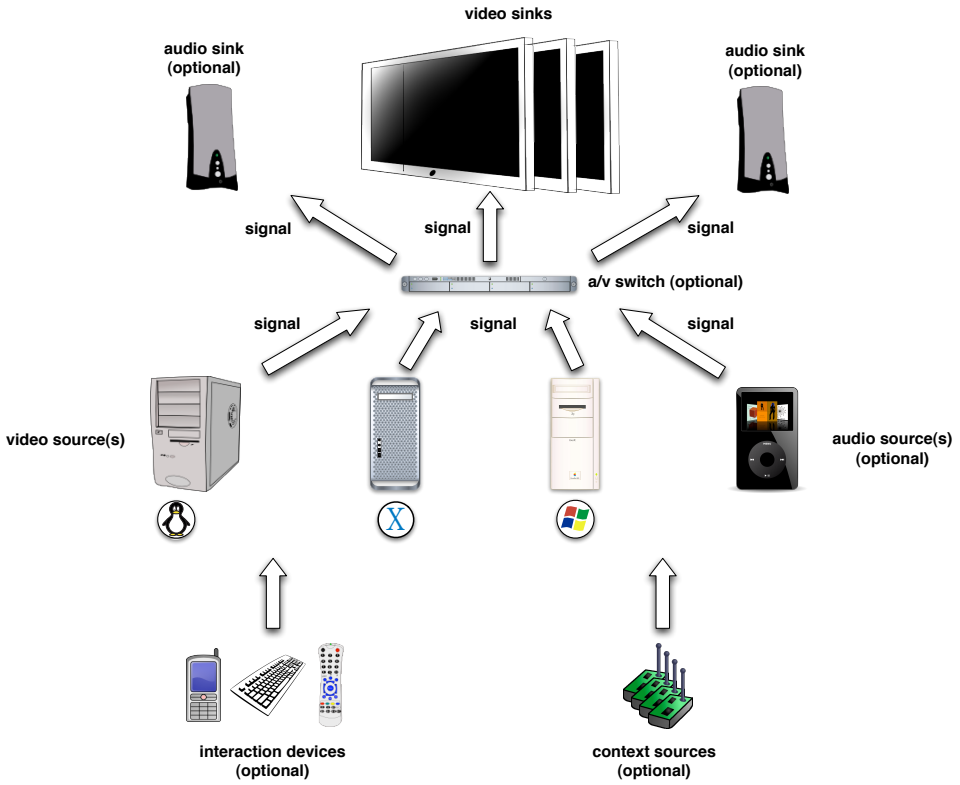


Figure 4.1: The hardware environment of a public display.

4.3 Discussion of Requirements

In this section we analyse how the requirements presented in chapter 3 influence the overall design of the software infrastructure presented in this thesis. Specifically we focus on requirements that affect the computational view of the infrastructure's architecture.

4.3.1 Content Scheduling

One of the key requirements that we obtained as a result of the analysis of use cases, existing public display systems and probes in chapter 3 was that researchers use diverse criteria to determine when pieces of experimental content should be displayed (requirement R4). Besides natively supporting a range of criteria, our software infrastructure would therefore most likely also have to provide support for adding new criteria, e.g. if a new type of sensor was to be used as input to the scheduling process as part of an experiment.

Multiple Scheduling Algorithms

The requirement for supporting a diverse set of scheduling criteria makes it difficult to employ a single scheduling algorithm that can be used for all types of experimental content. For example, the experiences we gained with the technology probes described in chapter 3 showed that a constraint-based scheduling algorithm that creates timelines was adequate for scheduling day-to-day background content. However, the same approach was unsuitable for the introduction of interactive, on-demand content (requirement R5) or content of variable length (requirement R10).

The sheer number of scheduling criteria that would have to be supported makes it unlikely that a single scheduling algorithm can be constructed that provides support for all the criteria that researchers might require support for at any point in time. Moreover, such a scheduling algorithm would have to evolve constantly as support for new scheduling criteria has to be introduced, e.g. to support new context sources or interaction devices. Since a single algorithm is not able to satisfy the requirements of experimental and non-experimental content in a public display network, we propose *to enable researchers to develop multiple experiment-specific algorithms in the form of separate schedulers that are able to operate concurrently on the same set of displays in the public display network* (design proposition DP1).

Distribution of Schedulers

Experiments may require content to be scheduled based on input from sensors or interaction devices that can only be interfaced through native APIs that are accessible on the computers that these devices are attached to (requirements R8 and R9). Moreover, these sensors and interaction devices

do not necessarily have to be attached to one of the computers driving the public displays. For example, in the case of Metamorphosis (the experimental piece of electronic art that was deployed in the Underpass) the installation's passive infrared sensors were attached to a separate machine that was responsible for scheduling content and controlling the installation, but did not produce any output itself. These requirements can be addressed by *allowing schedulers to be deployed to and executed on arbitrary computers that are attached to the display network* (design proposition DP2).

Operations Provided

The experiences we gained with the technology probe in the Brewery Arts Centre revealed a strong requirement to be able to arrange pieces of content into precisely defined sequences (requirement R12: "support for orchestrated performances"). This meant that we had to be able to precisely define when transitions between content items took place, i.e. any scheduling approach that introduced elements of randomness into the play-out schedule was not suitable. More crucially not only did content have to be orchestrated on a per-display basis, but also across multiple displays. The installation used three projected displays that were installed side-by side. To achieve an aesthetically pleasing schedule, the timings of each content item had to be synchronised with the timings of content items on the other displays.

If schedules have to be produced that are aesthetically pleasing, then the ability to "start" and "stop" the playback of content items is not enough to achieve such schedules. Depending on the size of content items and their storage locations, content players require different amounts of time to pre-fetch these content items and to transition into a state in which output on the displays can be produced. In the best case these preparation delays lead to 'dead airtime' on individual displays in between content items. If content is to be synchronised across multiple displays, then a preparation delay on one display will cause delays on all other displays as well, as the displays will not be able to start the play-out of content until all involved displays are ready to do so.

These issues can be addressed by separating the instantiation and pre-loading of content from the actual play-out, and by providing schedulers with control over when these operations are carried out. This means that schedulers are able to instantiate and pre-load content before it is scheduled to appear on the targeted displays. During the preparation phase content is kept

hidden, i.e. the preparation phase can be carried out while other content is visible on the displays. Progressing from one content item to the next is then simply a matter of making the old content item invisible, and of making the new content item visible (if continuous media is used, then playback has to be started at the same time).

In summary, we propose *to expose operations to schedulers that offer precise temporal control over the instantiation and pre-loading of content, its playback states, and its visibility* (design proposition DP3).

Provision of a Distributed Scheduling API

If our aim is to support developers of experiment-specific schedulers by providing operations that allow precise control over the life-cycle and visibility of content, then one of the resulting questions is how access to such operations can be provided.

For example, PlanetLab [BBC+04, Pla08b], one of the best known distributed platforms for research, provides researchers with direct shell access to virtualised network nodes. Researchers are able to control the execution of experiments by using shell commands to instantiate and terminate the processes that comprise these experiments.

The Globus Toolkit [Fos06, Glo08b] is another example of a distributed platform for research. Experimenters using Globus are able to control the life-cycle of experiments using GRAM, the "Grid Resource Allocation and Management" service [FFM07, Glo08a], a set of console-based tools that researchers are able to execute remotely on their own machines. Moreover, Globus provides Web-Service-based Java and C APIs to GRAM that researchers are able to use to construct their own execution management tools that they can run on their own machines.

In the context of our own software infrastructure we propose to enable researchers to develop experiment-specific schedulers by providing them with a distributed API. The provision of an API offers several benefits compared to direct shell access or the provision of console-based tools. Firstly, by providing an API we are able to offer operations that are targeted at the task at hand, i.e. at reasoning about and controlling the life-cycle and visibility of content on public displays. In comparison, direct shell access typically offers the means for managing and terminating processes, but does normally not provide the means for controlling the presentation of the output generated

by these processes.

The provision of an API can also help to reduce the complexity involved in developing schedulers. In the cases of both direct shell access and console-based tools that can be executed remotely on the experimenters' own machines, it is difficult to use complex data structures as arguments or return values of operations. The arguments of shell-based commands are text-based and return values are either passed back in the form of an integer-based result code, or as text-based output on the standard output. If complex data structures are to be used as arguments or return values, these have to be converted from their native representation into a text-based format, adding to the complexity involved in the development of schedulers. The use of a native API typically eliminates the requirement to convert arguments.

By providing a distributed API that can be accessed from arbitrary machines in the display network we are able to support the level of distribution of schedulers that we argued for (see design proposition DP2). Moreover, by offering a distributed API all aspects of communicating with the actual display machines can be made transparent to users of the API, hence facilitating the development of schedulers.

In summary, we propose to enable researchers to develop experiment-specific schedulers by *providing* them with *a distributed API* (design proposition DP4).

Sharing Model

The software infrastructure presented in this thesis has to provide "support for research-related and non-research-related content" (requirement R2). On public displays the main resources that different content items compete for are airtime and display real-estate. While airtime can be divided between different pieces of content, this is more difficult in the case of display real-estate: if multiple content items share the same display and are, for example, assigned different regions of the display, then all of these content items will obviously be visible to users at the same time. While the division of displays into multiple areas is commonly used in commercial public display systems (see chapter 2), the schedules in these systems are typically created by single administrative entities. As a result, schedules can be created more easily in which the pieces of content in the different areas of the display are compatible with each other. Compatibility in this case may, for example, mean that content items that are shown at the same time do not

deliver contradicting messages to observers of the display, or that individual content items do not distract viewers from other information on the screen. In our system, however, where content from different research experiments competes for screen real-estate, it is unlikely that there will be a comparable central administrative entity that is able to ensure that content items originating from different experiments are compatible with each other. Moreover, even if such an authority existed, it is likely that this authority would not be able to reach meaningful decisions: the compatibility of research-related content is likely to depend partly on the nature of the experiment that is being conducted and the research questions that are to be answered, However, the assessment of compatibility will also depend on the nature of the other content items that the compatibility is to be tested against. Compatibility is therefore probably best assessed by researchers conducting the experiments, and would in the worst case involve a check of their content items against all other content items in the system.

However, allowing multiple experiments to be displayed simultaneously on the same display without such compatibility checks might lead to issues with the evaluation of these experiments. It might, for example, be unclear whether the low user-uptake of a certain piece of experimental content was a result of fundamental problems with the content itself, or whether users were in fact distracted by other content that was shown on the same display at the same time.

Even if a satisfactory solution for ensuring the compatibility of content items existed, we would still expect the division of screen real-estate between different pieces of content to lead to significant additional complexity. Complexity is, for example, introduced by the need to resize content to make it fit the assigned screen segment. While it may be possible to scale the media itself relatively easily, this can have significant impact on the usefulness of the content. For example, the font size of textual information is often specifically tailored to certain display sizes to ensure that the text is readable when shown on the public displays. If content is to be displayed in screen segments of varying dimensions, then either content authors will have to author multiple versions of their content (one for each segment dimension), or mechanisms have to be employed to automatically adapt content to different sizes of screen real-estate, adding to the complexity of the overall software infrastructure.

Additional complexity can also be introduced by the desire to constrain the presentation of certain content items, for example to certain minimum or maximum sizes or to specific areas of the screen. As a result, formats are required to express this form of metadata. The software

infrastructure would be required to implement the functionality to parse these constraints.

We therefore argue that attempts at dividing display space significantly increase the complexity of the system and its use, and as a result we propose *to use time-division of the airtime on displays as model for sharing display resources between experiments* (design proposition DP5).

Transactional Semantics

If multiple schedulers are able to operate independently on the display in a public display network, there is a potential for conflicts. Conflicts occur when schedulers attempt to make pieces of content visible on displays that already show other content items. When looking at single displays in isolation this means that the content item that is currently visible remains visible, and that the new content item cannot be shown unless it is of higher priority and is therefore able to preempt the display. Conflicts are, however, more problematic when these occur during orchestrated performances that involve multiple displays. In the case of Metamorphosis in the Underpass, for example, a failure to show one of the videos that is part of the performance means that the other two videos should not be shown as well. Ideally, the three videos should appear as atomic unit, and intermittent states where a scheduler attempts to make a video visible on one of the displays should not be observable by users looking at the displays. We therefore identified the requirement to "provide support for atomicity and isolation" (requirement R13).

In an environment where multiple independent Schedulers are able to operate on the same set of displays at the same time, atomicity and isolation can be provided using transactions. In this approach API operations can be grouped into transactional blocks. Transactions in the context of database management systems typically provide atomicity, consistency, isolation and durability. The transactional semantics we propose support a subset of these guarantees: atomicity, consistency and isolation. Grouping API operations into a transactional context ensures that these operations are either all carried out successfully, or that none of the operations in the block are carried out. These semantics enable researchers to control content that spans multiple displays as an atomic unit and specifically ensure that changes to the visibility of content that are carried out in a transactional context are carried out atomically. The isolation property of our transactions guarantees that intermittent effects of operations that are carried out in a transactional context are not made visible to other entities in the system or to humans observing

the displays. Moreover, the consistency property ensures that all state changes on public displays that are the result of committing a transaction are compliant with the constraints and policies of our software infrastructure. The consistency guarantee would, for example, be violated if, as result of a transaction, two content items were visible on a display at the same time (see design proposition DP5).

While we consider atomicity, consistency and isolation to be important properties of our transactions, we propose not to require support for durability. In the context of database management systems, durability typically ensures the persistence of the effects of a transaction, particularly in the light of hardware and software failures. However, in the context of operations for modifying the state and visibility of content in public display networks, the provision of durability adds significantly to the complexity of the software infrastructure. Consider the following example: a researcher uses a transaction to make content items visible as an atomic unit on a set of displays. After a while, one of the displays fails and eventually recovers. Two difficulties exist in this scenario. Firstly, if continuous media was used as content, the recovery procedure would be required to ensure that playback is not out of sync as result of the failure. As a result, it would not be sufficient to recover display state based on persistent logs. Instead, the software infrastructure would be required to understand how playback would have progressed during the time when the display was unavailable. This may be achievable in the case of continuous media, but is significantly more complex in the case of event-driven content (e.g. requiring the content itself to maintain persistent logs, or requiring the software infrastructure to replay the sequence of events). Secondly, the display might not recover until after the scheduler has removed the content from the displays. In this case the recovery of the display's state from a log does not return the display to its intended state: as a result of the recovery procedure content will be shown on the display, despite the intentions of the experimenter to remove the content from the involved displays. Modifying the recovery procedure to achieve the correct behaviour again increases the complexity of the software infrastructure. Finally, in the context of database management systems, the loss of the results of a transaction can have permanent consequences (e.g. leading to the loss of a reservation for a hotel room). However, the failure to provide durability in the context of a public display network generally has no long-term effect, since the ability to show content on a display does typically not depend on the history of content that has previously been shown on the same display.

In summary, we propose *to provide API operations with transactional semantics* (design propo-

sition DP6).

4.3.2 Levels of Abstraction

One of the requirements established as result of our investigations in chapter 3 was that our software infrastructure would have to provide support for non-standard hardware setups (requirement R15: "support for non-standard and dynamic hardware setups"). Such setups might, for example, include multiple computers that are attached to the same physical display and are able to produce output on the display in an alternating fashion. A video switch may make it possible to dynamically switch between the two machines at run-time. The installation in the underpass, for example, provided a set-up in which a single computer was equipped with a multi-headed graphics card. By associating the first three outputs of that machine with the projectors in the Underpass, large pieces of content could be created that stretched across all three projection screens without having to be split up into three separate content items.

One of the design questions is how much of these underlying hardware details to expose to the developers of schedulers, i.e. do developers of experiment-specific schedulers directly control the mapping of computers and their graphics outputs to projection screens in the Underpass. In such a model, making a piece of content in the Underpass visible across all three projectors would require developers of schedulers to understand that achieving this requires content to be instantiated on the PC, as opposed to one of the Mac Mini machines. The PC has a total of four graphics card heads, resulting in a virtual desktop of 4096x768 pixels size. Researchers would have to know that in order to have content shown across the first three graphics card heads, content has to be of size 3072x768 pixels and should be displayed left-justified on the PC's virtual desktop. They would require knowledge about the different projectors, their inputs and how to associate each graphics head with the appropriate projector input. It becomes clear that this model quickly turns the seemingly simple operation of making a single piece of content visible into a rather complex undertaking that requires a large amount of knowledge about the underlying hardware characteristics.

If we analyse the above example we realise that instead of knowledge about the exact hardware setup, developers could instead be given the possibility to think about public displays in terms of the properties that certain hardware combinations provide. The desire to show wide-screen content

in the Underpass mandates the use of a specific hardware configuration, i.e. the use of the PC with content scaled to a specific on-screen resolution and position, and a specific mapping between graphics card heads and projectors. If content is only to be shown across two projection screens, then multiple options for combining the PC's video card heads and projectors exist. However, if a researcher's only constraint is that his content is to be displayed across two projection screens, then these combinations will all be equally valid for achieving the researcher's goals. The question of selecting appropriate hardware resources for carrying out an experiment can therefore be reduced to the selection of combinations of those hardware resources whose properties match the requirements of the experiment.

Instead of requiring researchers to manually determine combinations that match their requirements, a list of sensible combinations can be provided by the software infrastructure, together with a description of the resulting properties of these combinations. Researchers are then able to select pre-defined combinations without requiring knowledge about how these combinations are formed. For example, the software infrastructure might advertise the combination of the first three graphics heads of the PC in the Underpass with all three projectors as combination "Underpass-Left-Centre-Right" with a total screen size of 3072x768 pixels. A combination of the first two graphics heads with the left and centre projectors might be advertised as "Underpass-Left-Centre" with a screen size of 2048x768 pixels. By enabling researchers to control the life-cycle of content and its visibility on these virtual entities, they can be freed from knowing about and having to handle the switching of display inputs.

To reduce the complexity involved in developing schedulers we therefore propose *to expose combinations of hardware resources as single objects that are characterised by their properties and hide the details of the underlying hardware configurations* (design proposition DP7).

4.3.3 Modelling Content Playback

We have argued above for providing API operations that offer researchers temporal control over the steps involved in instantiating and showing content on public displays. It is apparent that these steps affect the internal state of content players. We therefore propose *to model content players as stateful entities* (design proposition DP8).

4.3.4 Summary of the Design Propositions

design proposition DP1	to enable researchers to develop multiple experiment-specific algorithms in the form of separate schedulers that are able to operate concurrently on the same set of displays in the public display network
design proposition DP2	to allow schedulers to be deployed to, and executed on, arbitrary computers that are attached to the display network
design proposition DP3	to expose operations to schedulers that offer precise temporal control over the instantiation and pre-loading of content, its playback states, and its visibility
design proposition DP4	to provide a distributed API
design proposition DP5	to use time-division of the airtime on displays as model for sharing display resources between experiments
design proposition DP6	to provide API operations with transactional semantics
design proposition DP7	to expose combinations of hardware resources as single objects that are characterised by their properties and hide the details of the underlying hardware configurations
design proposition DP8	to model content players as stateful entities

4.4 Overview of the Computational Model

At the core of the abstractions we provide is the concept of a *Display*. A Display represents a specific combination of audio & video sources and audio & video sinks in a given installation and hides the potential underlying complexity of non-standard hardware mappings from developers, allowing us to provide "support for non-standard and dynamic hardware setups" (requirement R15). Displays act as outlets for the output emitted by *Applications*, stateful entities that render the visible or audible output for content items or experimental applications. Each instance of an Application is instantiated on a single Display. Displays expose operations that allow Applications to be instantiated and terminated. They also offer operations to control the visibility of output produced by Applications. Moreover, each Display is responsible for detecting and handling contention between Applications that are instantiated on the same Display, allowing us to support

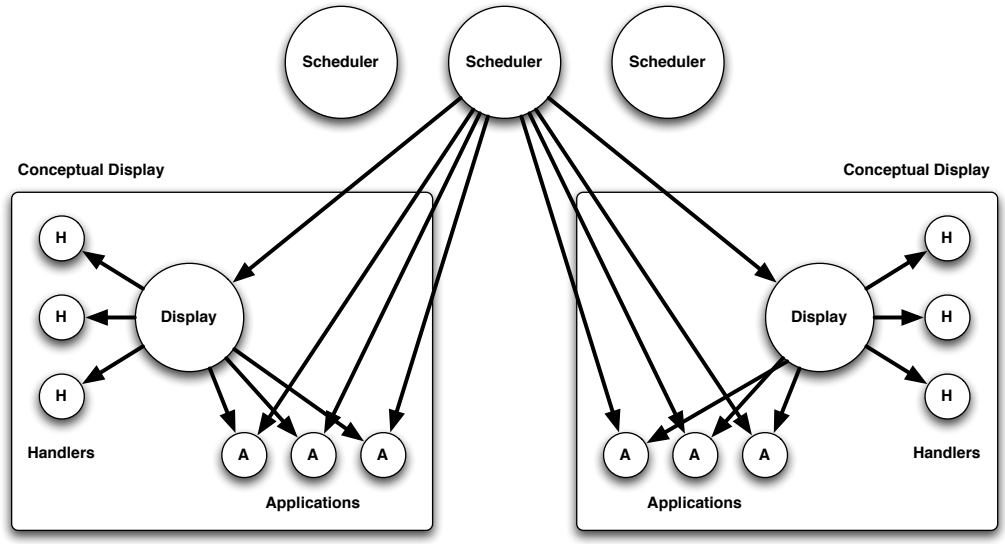


Figure 4.2: Overview of the computational model.

"arbitration between conflicting pieces of content" (requirement R3). Contention is handled based on a priority value that is associated with each Application. Policies can be employed, for example, to allow higher-priority Applications to preempt Displays from lower-priority Applications, addressing the requirement R7 ("support for priorities and preemption").

Conflicts between different Displays that occur when two or more Displays share a piece of underlying hardware that cannot be accessed concurrently are addressed by *Handlers*. Handlers are optional policy components that can be associated with Displays to extend the functionality provided by our software infrastructure. Besides detecting conflicts between Applications on different Displays, Handlers can be used, for example, to toggle the inputs on LCD displays to ensure that the correct inputs are selected as output from Applications is displayed, or to create audit trails of Display activity (requirement R14: "support for audit trails"). To achieve their aims, Handlers intercept the invocations of operations on Displays. The use of Handlers allows us to "provide support for non-standard and dynamic hardware setups" (requirement R15) without having to expose the details of these setups to users of the model.

Entities that control the life-cycle of content and its visibility on Displays are called *Schedulers*. Multiple schedulers may be active in the display network at the same time, and these schedulers may simultaneously operate on the same set of displays, allowing us to provide "support for a wide range of scheduling criteria" (requirement R4). Schedulers are able to control Applications using a distributed *API* that provides remote access to the operations exposed by Displays and

Applications. As we have outlined above, providing remote access allows Schedulers to be executed on any machine that is connected to the display network. Researchers may use this feature to enable Schedulers to interface with native APIs to sensors and interaction devices that are only available on certain computers (requirements R8 and R9). The API provides operations for instantiating and terminating Applications, and for directly controlling the visibility of their output on Displays. This level of control enables researchers to precisely control the appearance of applications on public displays and enables them to arrange content into precisely orchestrated sequences (requirement R12: "support for orchestrated performances"). API operations can be grouped into *transactional blocks* that are executed as atomic units. Transactions also provide isolation, i.e. intermittent states of transactional blocks are not made visible to other entities in the system. Moreover, they ensure that intermittent states of transactional blocks remain invisible to humans observing the displays. We call this property *visual isolation*. The consistency property of our transactions ensures that the results of transactions comply with the policies of our software infrastructure, e.g. preventing a Display from having two Applications visible at the same time as result of a transaction. By providing these transactional semantics we are able to address the requirement to provide "support for atomicity and isolation" (requirement R13). Finally, the API allows Applications to be aggregated into Application Groups. API operations can be carried out on individual Applications as well as on Application Groups. By enabling developers to invoke operations on Application Groups we provide "support for bulk operations" (requirement R6).

Sources of context information and interaction devices are both modelled as separate computational entities. Information provided by *Context Sources* and *Interaction Devices* may be accessed by Applications and Schedulers either using native APIs or using a common eventing framework that can be used as foundation for developing "support for context-sensitivity and personalisation" (requirement R8) and "support for interactivity" (requirement R9).

An overview of the relationships between the entities of the computational model is depicted in figure 4.3.

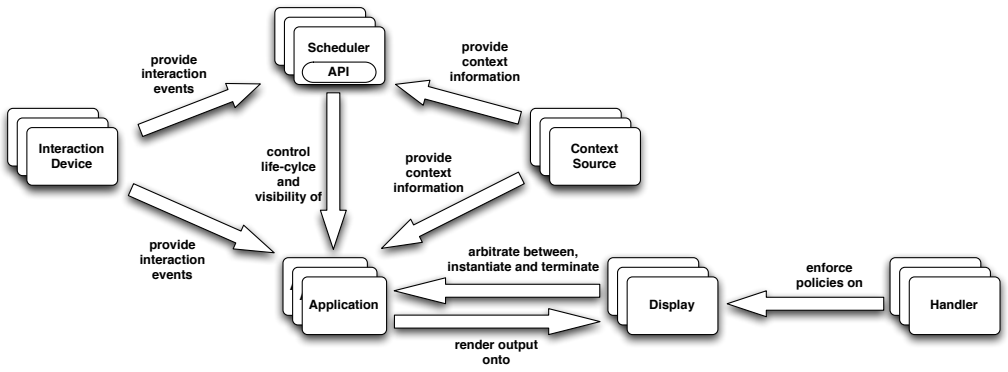


Figure 4.3: Relationships between entities in the computational model.

4.5 Computational Entities and Provided Operations

In this section we provide a detailed description of the entities in our computational model and the operations these entities provide.

4.5.1 Applications

Summary: Applications represent entities that are responsible for rendering content on Displays. They have an internal state model reflecting their life-cycle, playback state, and visibility.

Operations

- `ChangeState`: invocations of this operation trigger changes to the Application's internal state.

  Arguments

  - `target_state`: the intended state that the Application should transition to.

  Returns:

  - `result`: a result code indicating whether the operation was carried out successfully.

- `Constructor`: used to create an Application instance.

  Arguments

  - `content_url`: a URL identifying the piece of content that is to be rendered by the Application.

Attributes

- `application_state`: the Application's current state.
- `display_id`: the globally unique identifier of the Display that the Application has been instantiated on.
- `content_url`: a URL identifying the piece of content that is rendered by the Application.

Component Interaction

- Schedulers invoke `ChangeState` to instruct Applications to pre-fetch content and to get ready to produce output.
- Displays invoke `ChangeState` to control the visibility of the Application's output and the playback state of its content.

Description

Applications are stateful software entities that render content items onto Displays. Different types of Application provide support for rendering different content types. To show a particular piece of content of a certain type on a Display, Schedulers instantiate an Application that is capable of showing that particular content type and subsequently control the life-cycle and visibility of that Application instance. Examples of Applications include video players, renderers for Web-based content and custom-built interactive applications. We provide a detailed description of the engineering of Applications in chapter 5.

Researchers would normally not have to engineer and implement their own Applications, as Applications capable of rendering the most common content types (e.g. videos, images, and web-based content) will be provided for them. However, if experiments employ custom-crafted

content that cannot be rendered by one of the provided Application types, researchers are able to implement their own Applications that provide support for their experiments. This might for example be necessary if experimental content has to be able to use native APIs to interface with Interaction Devices or Context Sources and is therefore implemented in the form of a native process (requirements R9 and R10). By enabling researchers to implement their own Applications, we are able to support both standard content types and proprietary applications (requirement R11).

Application instances are created by Displays. Each Application instance is addressable using a globally unique *Application Process Identifier* and is capable of producing output on the Display it was instantiated on. The association between an Application instance and a Display cannot be changed during run-time, i.e. running Applications cannot be migrated between Displays. While multiple Applications can be active on the same Display at the same time, each Display is only able to show the output of a single Application at any single point in time (see design proposition DP5).

Applications contain state machines reflecting their readiness to produce output and the visibility of that output (see design proposition DP8). The state model we use is comparable to that of Players in the Java Media Framework [Sun08]. However, in contrast to the model used in JMF, our model additionally allows control over the visibility of content. Moreover, instead of the distinction that is made in the context of JMF between the initialisation of Application-internal components and the pre-fetching of content, our state model does not distinguish between these two actions and uses a single state (`PREPARED`) to indicate that the initialisation of internal components and the pre-fetching of content have been carried out successfully. Our state model comprises the following states:

- `IDLE`: the Application has been instantiated, but is not yet ready to produce output. This might, for example, mean that content still needs to be pre-fetched, or that software components that are necessary to produce output have not been initialised so far. Applications automatically enter this state after they have been instantiated.
- `PREPARED`: the Application is ready to produce output, but is currently not doing so, i.e. the Display does not currently show any output from this Application. The content that is to be displayed by the Application has been initialised. Depending on the type of content that is supported by the Application, this initialisation may have different effects. In the case of

continuous media, for example, the video has been pre-fetched, and the playback pointer is set to the start of the media. Playback will not start until the Application's state is change to `VISIBLE`. In the case of interactive content, the content has been loaded and initialised.

- `VISIBLE`: the Application is producing output and that output is currently visible on the Display. If continuous media is used, playback has been started. Each Display can have at most one Application in this state at any point in time.

- `NOT_VISIBLE`: the Application is not producing output, i.e. the Display does not currently show any output from this Application. If continuous media is used, playback has been paused, and a subsequent state change to `VISIBLE` will cause playback to continue at the current position. In case of interactive content, the content is left in the current interaction state, i.e. a subsequent state change to `VISIBLE` will resume the interaction at the point it was in when the Application was made `NOT_VISIBLE`.

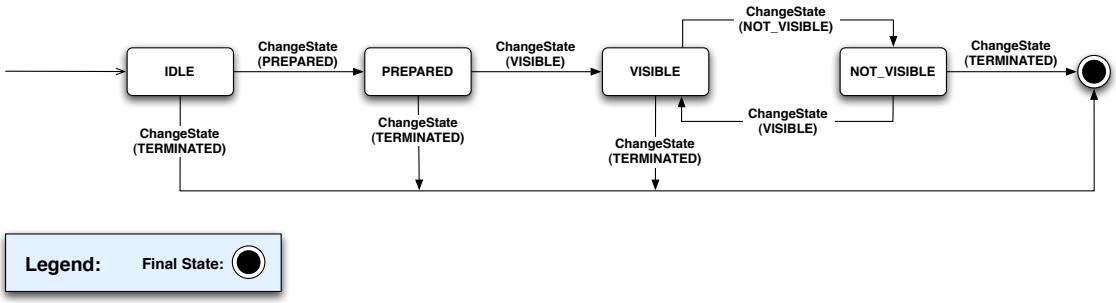


Figure 4.4: State transitions as a result of `ChangeState` operations.

In addition to the states outlined above, Applications support state transitions to the virtual state `TERMINATED`. A transition into this state causes Application instances to exit gracefully.

State transitions are not initiated by Applications themselves, but are instead requested by external entities by invoking the `ChangeState` operation. An overview of the state machine and the transitions that can be initiated using `ChangeState` operations is provided in figure 4.4. Schedulers invoke `ChangeState` to request Applications to access and pre-fetch content by requesting a state transition to `PREPARED`. A successful outcome of the invocation indicates to a Scheduler that the content has been pre-fetched and is ready to be displayed. Displays invoke `ChangeState` to request changes to the visibility of Applications. The invocation may be a result of `Transition` instructions (see section 4.5.2) received by Displays from Schedulers. Displays may also invoke `ChangeState` during conflict resolution to make conflicting pieces of content `NOT_VISIBLE`.

The `ChangeState` operation and the state model provide Schedulers with precise control over the instantiation and pre-loading of content, the Application's playback states and the visibility of its content (see design proposition DP3). Operations are carried out by Applications without introducing any artificial delays, providing Schedulers with the necessary temporal control to enable them to create precisely orchestrated performances (see requirement R12).

To summarise, Applications are stateful entities that are responsible for displaying content on Displays. They expose a single operation (`ChangeState`) that allows Schedulers and Displays to control the pre-loading of content, its playback state, and the visibility of the produced output. Multiple Applications can be active on a single Display at the same time. However, at any point in time only one Application may be visible on each Display.

4.5.2 Displays

Summary: Displays create and terminate Application instances, oversee their visibility and handle conflicts between Applications.

Operations

- `CreateApplication`: instructs the Display to instantiate a new Application instance.

  Arguments

  - `content_url`: a URL identifying the content item that is to be rendered by the Application.

  Returns

  - `pid`: the Application Process Identifier of the newly created Application.
  - `result`: a result code indicating whether the operation was carried out successfully.

- `Transition`: requests a change to the visibility state of an Application.

  Arguments

  - `pid`: the Application Process Identifier of the Application whose visibility state is to be changed.

- `target_state`: the intended visibility state, i.e. one of `VISIBLE` or `NOT_VISIBLE`.
- `priority`: the priority of the `Transition` operation (used during conflict resolution).

Returns:

- `result`: a result code indicating whether the operation was carried out successfully.

- `TerminateApplication`: instructs the Display to terminate an Application.

  Arguments

  - `display_id`: the Display's globally unique Display Identifier.

  Returns:

  - `result`: a result code indicating whether the operation was carried out successfully.

Attributes

- `running`: the Application Process Identifiers of all Applications that are currently associated with the Display.
- `visible`: if an Application is currently visible on the Display, then this attribute provides the Application Process Identifier of that Application and its priority value. Otherwise the attribute is set to `NULL`.

Component Interaction

- Schedulers invoke `CreateApplication`, `TerminateApplication` and `Transition`, to control the life-cycle and visibility of Applications;
- Displays invoke `ChangeState` on Application instances as a reaction to `Transition` and `TerminateApplication` requests from Schedulers;
- Displays enable Handlers to intercept invocations of `CreateApplication`, `TerminateApplication` and `Transition` by invoking the `HANDLE` operation exposed by Handlers.

Description

At the core of the abstractions we provide is the concept of a Display. Each Display represents a specific mapping between audio & video sources and audio & video sinks and is characterised by the properties that this mapping provides (see design proposition DP7). Moreover, each display is addressable using a globally unique *Display Identifier*. In the most basic case of a public display set-up where each computer is connected to a single physical display there will be one conceptual Display for each of these mappings. However, in the Underpass for example, several conceptual Displays can be identified, for example one for each Mac Mini, one for each addressable head of the PC workstation's video card, and one 'widescreen display' which comprises the first three heads of the PC workstation. By associating the output of Applications to a particular Display, the actual mapping of which computers content needs to be distributed to, which video outputs need to be selected etc. is hidden from the developers of Schedulers, allowing us to hide the potential complexities of non-standard and dynamic hardware setups from developers (see requirement R15).

Schedulers invoke `CreateApplication` to instruct Displays to create Application instances. Each Display manages the Application instances it has created. The URL that is passed along as part of the `CreateApplication` invocation is used by the Display to determine the content type that the Application instance is required to display, and to instantiate an appropriate Application that is capable of rendering content of that type. The URL is also passed along by the Display as an argument to the constructor of the newly created Application.

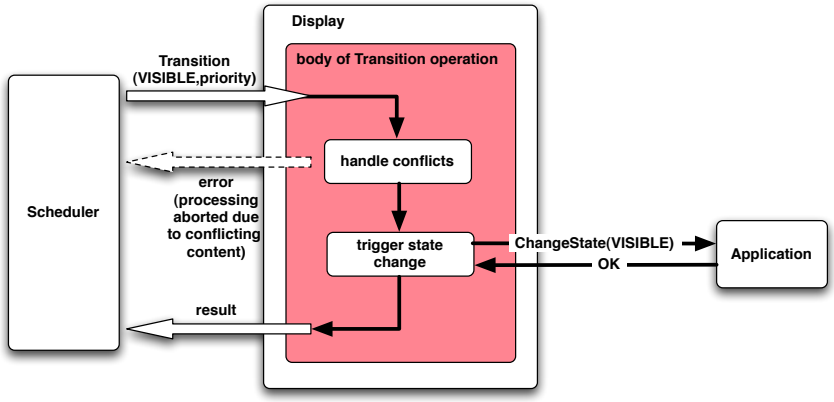


Figure 4.5: Displays as gatekeepers to visibility changes.

During the lifetime of an Application, the Display acts as point of coordination for operations affecting the visibility of the Application. This role enables Displays to enforce policies associated

with these operations. Examples include requesting state changes to other Applications in the case of priority-based preemption, or the enforcement of policies regarding the layout of content items on the display real-estate. Without Displays as points of coordination, the enforcement of these policies would have to be carried out by Applications themselves. Besides an unnecessary replication of functionality – functionality for priority-based preemption would, for example, have to be integrated into each Application – this would also increase the complexity involved in implementing Applications as these would not only have to implement the required policies, but would also be required to track the state of other Applications. Moreover, in contrast to Applications, which may be implemented by experimenters themselves, Displays represent trusted entities in the system and are therefore more suitable for the enforcement of policies.

As a result, Schedulers wishing to affect the visibility of Applications are required to invoke `Transition` operations on Displays instead of directly invoking `ChangeState` on Applications (see figure 4.5). The operation accepts the Application Process Identifier of the targeted Application, the intended visibility state and a priority value as arguments. The intended visibility state can be one of `VISIBLE` and `NOT_VISIBLE`, and these states correspond to the underlying Application states. The priority value is only specified if the target state is set to `VISIBLE` and defines the priority of the application that is to be transitioned. If the `Transition` operation completes successfully, then the priority value is associated with the Application instance, and this association is recorded by the Display. The association remains in place unless the Application is terminated or transitioned into the visibility state `NOT_VISIBLE`.

If a `Transition` operation does not conflict with other Applications on the Display (and if it complies with the policies of the Handlers associated with that Display), the Display requests changes to the Application's visibility and playback state by issuing a `ChangeState` request.

Schedulers invoke `TerminateApplication` to request an Application to be terminated. `TerminateApplication` causes the Display to instruct the targeted Display to exit gracefully by invoking its `ChangeState` operation with a target state of `TERMINATED`.

Priority-Based Conflict Resolution

The priority value that is passed along with `Transition` requests into target state `VISIBLE` is used as the determining factor during conflict resolution. Conflicts occur if an Application is to

be made visible on a public display on which another application is already visible. Depending on the underlying hardware configuration, such a conflict may involve either a single or multiple Displays.

Conflicts between Application instances that both reside on the same conceptual Display will be detected by that Display and resolved using a priority-based approach. As we have described above, if Applications are successfully `Transition`-ed into Application state `VISIBLE`, the priority value that was passed along as parameter to the `Transition` request becomes the priority value that is associated with that Application.

Using these priority values, conflicts are handled according to the following policy: let $p_{visible}$ be the priority that is associated with the Application that is currently visible on the Display. Let p_{new} be the priority value that is passed along as argument to a `Transition` operation that attempts to make a different Application visible. If $p_{new} <= p_{visible}$, then the `Transition` operation will fail and the Application that is currently visible will remain visible. If $p_{new} > p_{visible}$, then the Display will invoke `ChangeState` to cause the Application that has so far been `VISIBLE` to become `NOT_VISIBLE`. The Display will then use `ChangeState` to instruct the new Application to become `VISIBLE`, thereby allowing the new Application to preempt the Display. Figure 4.6[1] shows an example of the sequence of operations involved in priority-based conflict resolution.

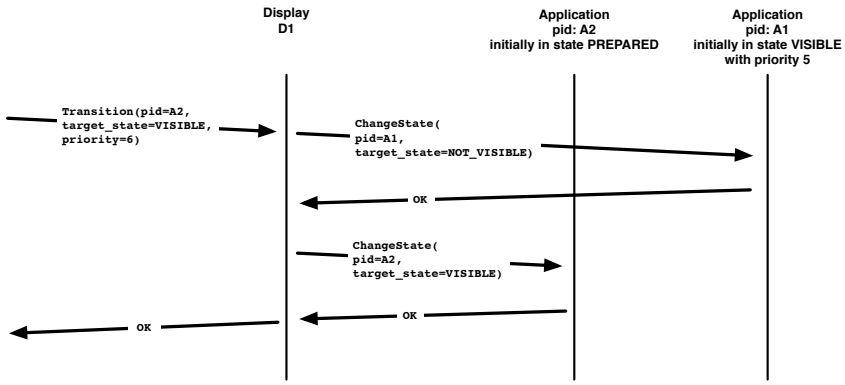


Figure 4.6: Example of priority-based preemption of a Display.

Conflicts that involve multiple Displays may be detected and resolved by specialised Handlers that use the above policies and `Transition` operations to make conflicting content on other Displays not visible (see section 4.5.4).

[1]Please note that due to the purely illustrative nature of the sequence diagrams in this thesis we have decided to base these on an informal notation. If the use of a more formal notation had been required, the diagrams could have, for example, been realised in the form of UML sequence diagrams [The07].

4.5.3 Schedulers

Summary: Schedulers control the life-cycle of Applications, their playback state and the visibility of their output.

Operations

Schedulers do not expose any operations to other computational entities.

Attributes

Schedulers do not expose any information to other computational entities.

Component Interaction

- Schedulers instruct Applications to pre-load content and to get ready to produce output by invoking the Applications' `ChangeState` operation;
- Schedulers control the life-cycle, playback-state and visibility of Applications by issuing `CreateApplication`, `TerminateApplication` and `Transition` requests to Displays.

Description

Schedulers use the operations exposed by Displays and Applications to precisely control the life-cycle of Applications and the visibility of the output these Applications produce. Schedulers are the system components most commonly programmed by experimenters using our software infrastructure. Specifically, Schedulers:-

- request new Application instances to be created by invoking `CreateApplication`.
- instruct Applications to pre-load their content and to ready themselves to produce output by invoking `ChangeState` with `PREPARED` as target state.
- request changes to the visibility of output produced by an Application and its playback state using the `Transition` operation.

- request Application instances to be removed from Displays by invoking `TerminateApplication`.

Schedulers invoke operations exposed by Applications and Displays using the distributed API that we describe in section 4.6. Different programming languages can be used to implement new Schedulers, making it possible to construct a diverse set of experiment-specific scheduling algorithms on top of the operations exposed by the API (see design proposition DP1). Whenever an algorithm or criterion is to be used that is not supported by the existing cadre of Schedulers, researchers are able to create new Schedulers. Researchers are, for example, able to create constraint-based Schedulers that construct timelines and are responsible for scheduling non-interactive, low-priority content, such as news bulletins and announcement. These Schedulers use the operations provided by Displays and Applications to instantiate Application instances and control their visibility on Displays according to the created timelines. Moreover, researchers are, for example, able to construct Schedulers that instantiate Applications and make them visible if certain interaction events occur, e.g. if a button is pressed. An example of a simple event-based Scheduler is depicted in figure 4.7. Schedulers can also be created that orchestrate content across multiple Displays by invoking the above operations at specific times in a specific sequence. Please note that all finite state machines in this thesis are represented a Mealy machines [Mea55].

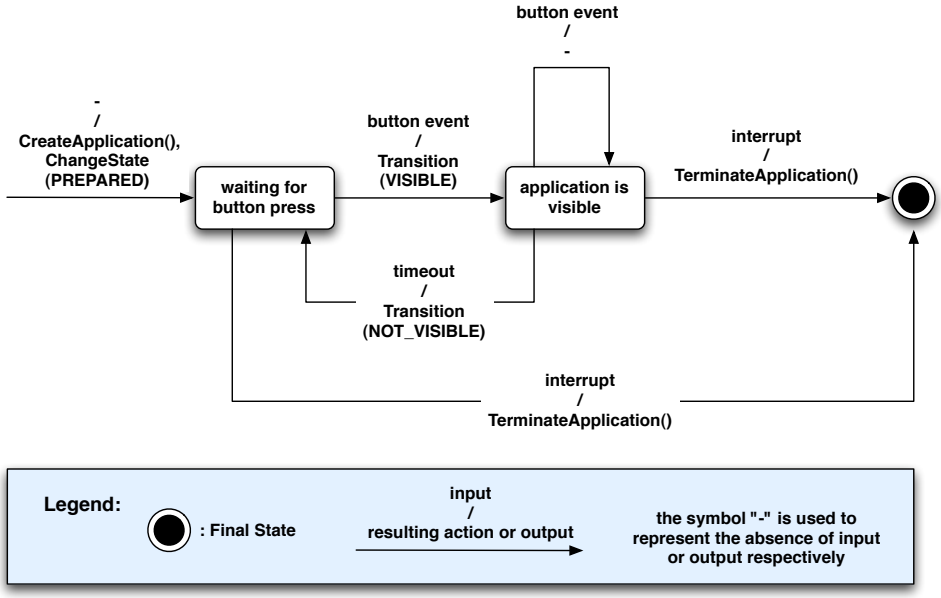


Figure 4.7: State diagram of a simple event-based Scheduler.

Conflicts that occur if different Schedulers attempt to make multiple Applications visible on the same Display or on conflicting Displays are handled by Displays or Handlers respectively. Researchers are therefore able to concurrently employ multiple Schedulers that use different scheduling algorithms and criteria and operate in the same public display network and on the same set of Displays (see design proposition DP1).

Using our distributed scheduling API, Schedulers can be executed on any computer that is connected to the public display network (see design proposition DP2), allowing Schedulers to be co-located with sensors or proprietary interaction devices (see requirements R8 and R9).

4.5.4 Handlers

Summary: Handlers are optional policy components that can be used to extend the functionality of Displays.

Operations

- `Handle`: invoked by Displays to instruct Handlers to process an invocation of `CreateApplication`, `Transition` or `TerminateApplication`.

  Arguments

  - `operation`: the operation that the Handler is instructed to process. Can be one of `CreateApplication`, `Transition` or `TerminateApplication`.
  - `phase`: indicates whether the call to `Handle` is made by the Display in the pre-processing phase or the post-processing phase of the operation, and therefore determines how the Handler is to process the operation.
  - `arguments`: the arguments that were passed along with the operation invocation.
  - `result`: if `Handle` is invoked in the post-processing phase, this argument contains the return arguments that the Display obtained by processing the `CreateApplication`, `Transition` or `TerminateApplication` operation internally.

Returns

- `stop_processing`: if set to `True` instructs the Display to stop processing the operation immediately and to return the result code contained in `result`. This field is only valid in the pre-processing phase.
- `arguments`: a potentially modified version of the arguments that were passed along to the `Handle` operation.
- `result`: a potentially modified version of the return arguments passed along when `Handle` was invoked.

Attributes

Handlers do not expose any information to other computational entities.

Component Interaction

- Displays invoke `Handle` to enable Handlers to intercept and process invocations to `CreateApplication`, `Transition`, and `TerminateApplication`. Besides being able to act on these invocations, Handlers are able to rewrite the arguments that were passed along with these invocations, and the return arguments before Displays return these to the callers. They are also able to instruct Displays to abort processing an invocation.
- Handlers may invoke operations exposed by other computational entities to achieve their aims.

Description

Handlers represent optional policy components that can be associated with Displays to extend their functionality. Handlers may be written for a variety of reasons, e.g. to provide audit trails of Display activity (see requirement R14), or to hide the complexity of the hardware set-ups that are underpinning Displays from the developers of Schedulers (see requirement R15). Handlers may, for example, be employed for:

- logging changes to the visibility of content on Displays;

- automatically switching display inputs in cases where multiple computers are attached to the same physical display device;
- detecting and handling conflicts between content on different (conceptual) Displays when these are attached to the same physical display hardware;
- implementing power-down policies for projectors to allow them to cool down from time to time;
- automatically powering on physical displays as soon as Applications are made visible.

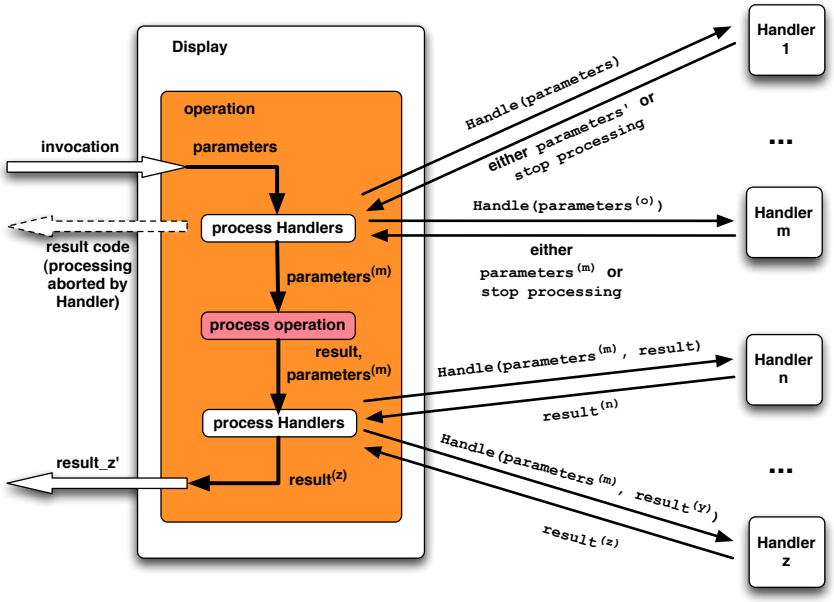


Figure 4.8: Interaction between Displays and Handlers.

Handlers are able to provide these features by being able to influence the processing of operations on Displays during two phases:

- the *pre-processing phase*, i.e. after the operation has been invoked, but before the actual processing is commenced by the Display. If multiple Handlers are associated with the Display, then Handlers are invoked by the Display in a sequence that can be specified on a per-display basis. Each Handler receives a copy of the arguments that have been passed in by the caller of the operation. Each Handler is able to use these arguments as input to its own processing logic, and Handlers may invoke operations on other computational entities as part of this processing. Moreover, handlers may modify the arguments that they have received, and

return the modified version to the Display. This modified version is then passed along by the Display as input to the next Handler in the sequence. The version of the arguments that is returned by the last Handler in the sequence is finally used by the Display for processing the operation. Handlers in this phase may also instruct the Display to send a negative result code immediately, causing the Display to abort the processing of any further Handlers or the main operation body.

Handlers in this phase may, for example, process `Transition` operations to target state `VISIBLE` to detect and potentially resolve conflicts with Applications on other Displays. We have, for example, constructed a Handler that is attached to conflicting Displays in the Underpass and intercepts `Transition` operations on these Displays. The Handler attempts to make conflicting Applications on other Displays `NOT_VISIBLE` using the priority-based preemption policies described in section 4.5.3. The Handler invokes `Transition` on these Displays to achieve the intended changes in the visibility of conflicting Applications. If the conflicting Applications cannot be transitioned, the Display is instructed to stop processing the `Transition` operation and to return a negative result code.

- the *post-processing phase*, i.e. after the operation has been processed by the Display, but before the Display sends off the return arguments. Handlers are passed a copy of the return arguments that the Display obtained by processing the operation locally. They are provided with a copy of the argument list as it was returned by the last Handler in the previous phase (this is also the argument list that was used by the Display to process the operation). As in the first phase, Handlers in this phase are processed sequentially. However, Handlers in this phase are not able to instruct Displays to abort the further processing of the operation. While Handlers in the first phase were able to modify the operation's argument list, Handlers in this phase are able to modify the return arguments of the operation, provided that these modifications do not change the overall outcome (i.e. success or failure) of the operation. As in the previous phase, the modifications applied by one Handler are passed as input to the next Handler in the sequence. The return arguments returned by the last Handler in the sequence represent the return arguments that are returned by the Display in response to the operation invocation.

  Handlers in this phase may, for example, be used to create play-out logs to provide audit trails about when, where and for how long content items were shown in the display network. Handlers are able to obtain this information by monitoring and recording instances where

`Transition` operations were processed successfully by Displays. We are therefore able to address the requirement for supporting audit trails (requirement R14) by enabling developers to create Handlers for this purpose.

Handlers can also be constructed that turn physical display hardware on (or off) as soon as an Application has been successfully transitioned to `VISIBLE` (or `NOT_VISIBLE` respectively). We have also used the same principles to construct Handlers that switch display inputs in cases where multiple display machines are attached to the same physical display.

An overview of these interactions between Displays and Handlers can be found in figure 4.8.

Handlers are normally written and configured at deployment time when an installation's hardware configuration is finalised. Handlers are also nominally the only entities in our architecture that embody knowledge about the technology of an installation and its specific mappings between audio and video sources and audio and video sinks. Once a Handler is in place for a given hardware setup, it would not normally be changed when new content or Schedulers are introduced. Each Handler is addressable by other computational entities using a globally unique *Handler Identifier* and may be associated with one or more Displays.

Please note that neither Schedulers nor Applications directly interact with Handlers, and so although they exist in our computational model, Handlers are normally not visible to the typical developers of Schedulers or Applications.

4.5.5 Interaction Devices

Summary: Interaction Devices provide information about user input to Schedulers and Applications.

Operations

Interaction Devices do not expose any operations to other computational entities.

Attributes

In general, Interaction Devices provide information about user interactions. However, different types of Interaction Devices use different conventions for the names, types and semantics of the attributes provided.

Component Interaction

- Schedulers obtain information from Interaction Devices, for example to be able to show particular pieces of content as a result of user interaction.
- Applications obtain information from Interaction Devices, for example to modify the displayed content to reflect the input users have made.

Description

As we learned during our survey of public display systems in chapter 2, our software infrastructure has to allow researchers to create interactive content (requirement R9). Our computational model does not prescribe specific types of Interaction Devices to be present in a public display network, nor do we limit Interaction Devices to a range of specific types. Instead we foresee Interaction Devices being added to computers whenever they are needed by individual experiments. As a result we do not provide any limitations regarding how Applications and Schedulers may access Interaction Devices. Information can be provided by Interaction Devices through proprietary APIs. Alternatively, Interaction Devices may disseminate information through a common eventing framework, and we expect standard content formats for certain types of Interaction Devices (e.g. pointing devices, or keyboard devices) to develop over time.

Schedulers can use the information obtained from Interaction Devices, for example, to reach decisions about when to instantiate Applications, when to make them visible, and for how long. A scheduler may, for example, react to users touching a touch-sensitive screen overlay and show a specific Application in response. Interactive Applications may obtain information from Interaction Devices and use this information to adapt their content as a reaction to user interaction.

4.5.6 Context Sources

Summary: Context Sources provide context information to Schedulers and Applications that can be used to create context-sensitive and personalised content.

Operations

Context Sources do not expose any operations to other computational entities.

Attributes

In general, Context Sources provide information (e.g. user presence or location) to Applications and Schedulers. However, different types of Context Sources employ different conventions with respect to the names, types and semantics of the attributes they provide.

Component Interaction

- Schedulers obtain information from Context Sources, for example to be able to personalise content, based on the identity of the users detected in the surroundings of a public display.
- Applications obtain information from Context Sources, for example to adapt the orientation of navigation arrows that are shown on a public displays to match the deployment location and orientation of that display.

Description

Context Sources provide additional information to Applications and Schedulers that enables them to adapt content and to reach scheduling decisions. Unlike Interaction Devices, Context Sources provide information about the overall context of the system, rather than intentional user input. For example, a Context Source may be able to identify individual users in the vicinity of a public display. A Scheduler may use this information to personalise the types of content displayed according to the preferences of those individuals. Similar to information made available by Interaction Devices, context information may either be accessible through proprietary APIs or in the form of

events that are disseminated using a common eventing framework.

As in the case of Interaction Devices we do not attempt to provide developers with a complete framework for distributing and accessing context events in public display networks. Nor do we try to specify or constrain context. Instead our computational model enables Context Sources to be added to the display network and accessed by researchers as needed to construct context-sensitive and personalised experiments on top of our software infrastructure (requirement R8).

4.6 The Application Programming Interface (API)

To facilitate the invocation of operations and the collection of results, the operations exposed by the computational entities described above are made available in the form of a distributed Application Programming Interface (API). The API provides programming interfaces to the `ChangeState`, `CreateApplication`, `Transition` and `TerminateApplication` operations. Figure 4.9 shows an example of a minimal Scheduler (shown in pseudo-code) that uses the API to display a video for a duration of 10 seconds (please note that all examples in this section use pseudo-code; specifically, the use of an object-oriented notation for API operations in the examples does not mandate implementations of the API to expose object-oriented interfaces). The Scheduler invokes `CreateApplication` to create a new Application instance on the Display represented by object `display1`. The video is pre-loaded by invoking `ChangeState` on the newly created Application instance. The video is subsequently played and made visible using a call to `Transition`. After a period of 10 seconds, the Application is transitioned into state `NOT_VISIBLE`, causing output to be made invisible and playback to be paused. A call to `TerminateApplication` finally destroys the Application instance.

```
( result, app ) = display1.CreateApplication(
                              "http://a.host/a_video.mpeg" )
app.ChangeState( PREPARED )
display1.Transition( app, VISIBLE, DEFAULT_PRIORITY )

sleep( 10 )

display1.Transition( app, NOT_VISIBLE )
display1.TerminateApplication( app )
```

Figure 4.9: A Scheduler that shows a video for a duration of 10 seconds.

The operations offered by the computational entities through the API allow Schedulers to control the instantiation, pre-loading, playback state and visibility of content on public displays. As we have seen in the example shown in figure 4.9, the scheduling API offers precise control over the sequence in which operations are executed. If operations are to be executed at specific points in time, it is the responsibility of Schedulers to implement the logic that ensures that the corresponding API calls are made at the correct points in time (see design proposition DP3). In the example shown in figure 4.9, the Scheduler uses a simple sleep statement to achieve the necessary temporal control, but the same principles allow the construction of Schedulers that, for example, make content visible at specific times during the day.

Moreover, by providing Schedulers with control over the sequencing and timing of operations, the API can be used to construct orchestrated performances of content involving multiple public displays by being able to specify the points in time when Applications are to be instantiated, when content is to be pre-loaded and when transitions between Applications are to occur. The ability to support the orchestration of content across displays was one of the requirements that we identified in chapter 3 (requirement R12).

The example in figure 4.10 illustrates how the API can be used to construct Schedulers that are capable of displaying content on-demand as result of user interaction. The ability to support on-demand content was one of the requirements that we identified in chapter 3 (requirement R5). The example outlines a simple event-based Scheduler that displays an interactive Application as soon as user interaction occurs. Once visible, the Application is kept visible for as long as interaction with the Application continues to take place. If no further interaction occurs for a specified duration of time (specified by `timeout`), the Application is hidden from the user's view by transitioning it into state `NOT_VISIBLE`. The Application is terminated if the Scheduler receives an interrupt signal. Besides support for on-demand content, the example also demonstrates how the API can be used to construct Schedulers that support content items whose duration is not known in advance (requirement R10). In case of the example, the content is to be shown as long as the interaction takes place.

API instances communicate with other entities of the software infrastructure over the network using a distributed protocol. The distributed nature of the API (design proposition DP4) enables Schedulers to be hosted on any machine that is connected to the public display network (design proposition DP2).

```
( result, app ) = display1.CreateApplication(
                            "file:///content/interactive_app" )
app.ChangeState( PREPARED )
while not interrupted:
    wait for interaction event or timeout
    if interaction event received and app is not visible:
        display1.Transition( app, VISIBLE, DEFAULT_PRIORITY )
    else if timeout received and app is visible:
        display1.Transition( app, NOT_VISIBLE )
display1.TerminateApplication( app )
```

Figure 4.10: A simple event-based Scheduler that shows an interactive Application.

Besides programming interfaces to the operations `CreateApplication`, `ChangeState`, `Transition`, and `TerminateApplication`, the API provides support for carrying out bulk operations on groups of Applications (Application Groups), and for grouping operations into transactional blocks to enable schedulers to schedule multiple Applications as atomic unit.

4.6.1 Application Groups

Application instances may be bundled into *Application Groups*. An Application Group allows an operation to be carried out simultaneously on all applications within that group by using a single invocation at the API level. Such bulk operations become more important as the display network grows in scale, and, for example, a piece of content is to be shown simultaneously on multiple public displays. Instead of invoking the same operation once for each Application instance, Schedulers are able to group these Applications into an Application Group, allowing a single operation to be used to control all Applications.

Figure 4.11 shows the conceptual model of API invocations on Application Groups using the example of a `ChangeState` operation. The API translates the API invocation into operations on individual Applications and aggregates the results returned by the involved Applications. API support for operations on Application Groups is provided for `ChangeState`, `Transition` and `TerminateApplication` operations.

Each Application Group is identified by a globally unique *Application Group Identifier*. The API provides developers of Schedulers with the ability to manage Application Groups. This includes the ability to create new Application Groups, to add Application Instances to Application

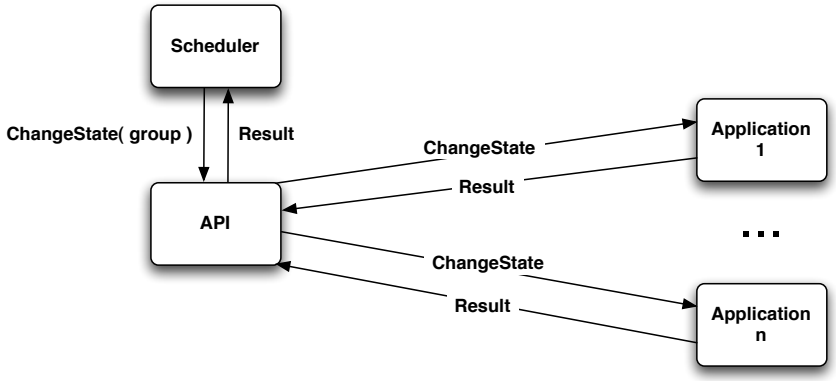


Figure 4.11: API support for Application Groups (here in the context of a `ChangeState` operation).

Groups, and to destroy Application Groups. In our current computational model, each Application Group is only accessible through the API instance that created the Application Group, i.e. Application Group Identifiers cannot be shared among different API instances (i.e. among different Schedulers). Thus Application Groups are not first-class entities in our model.

An example of the use of Application Groups is shown in Figure 4.12. The Scheduler depicted in the figure instantiates two Applications on two Displays. Once instantiated, both Application instances are added to a new Application Group. The Application Group is subsequently used to invoke `ChangeState`, `Transition` and `TerminateApplication` operations. As a result, these invocations affect both Applications.

```
( result, app1 ) = display1.CreateApplication(
                            "http://a.host/a_video.mpeg" )
( result, app2 ) = display2.CreateApplication(
                            "file:///content/experiment.html" )
group = new ApplicationGroup()
group.addApplication( app1 )
group.addApplication( app2 )
group.ChangeState( PREPARED )
group.Transition( VISIBLE, DEFAULT_PRIORITY )

sleep( 10 )

group.Transition( NOT_VISIBLE )
group.TerminateApplication()
```

Figure 4.12: Pseudo-code showing the use of API operations on an Application Group.

4.6.2 Transactional Support

The operations that are exposed by the entities in our computational model can be invoked with the transactional semantics we proposed as part of design proposition DP6. The API provides support for creating new transactions, and for grouping the operations `CreateApplication`, `ChangeState`, `Transition`, and `TerminateApplication` into the resulting transactional blocks. API operations are made available to developers for committing and aborting transactions. However, details relating to how this support is provided and the syntax of corresponding API functions are specific to the actual engineering and implementation of the API, and are therefore described in the following chapter.

Our transactions provide *atomicity*: if any single operation fails or times out, then the transaction aborts and all operations involved in the transaction roll back. This allows developers to reliably schedule content across multiple Displays as an atomic unit in the presence of potential conflicts and failures. Figure 4.13 demonstrates how this can be achieved for two content items. The Scheduler uses a transactional block to enclose the instantiation of both Application instances, the pre-loading of content, and the `Transition` operation to make the Applications visible on the displays. Any error that occurs inside the transactional block will cause all operations inside this block, including the `CreateApplication` operations, to be rolled back. Content will only be made visible if all operations inside the transactional block can be executed successfully and both Application instances can be transitioned into Application state `VISIBLE`.

```
group = new ApplicationGroup()
BEGIN TRANSACTION
    ( result, app1 ) = display1.CreateApplication(
                                "http://a.host/a_video.mpeg" )
    ( result, app2 ) = display2.CreateApplication(
                                "file:///content/experiment.html" )
    group.addApplication( app1 )
    group.addApplication( app2 )
    group.ChangeState( PREPARED )
    group.Transition( VISIBLE, DEFAULT_PRIORITY )
COMMIT TRANSACTION

if transaction was successful:
    sleep( 10 )
    group.Transition( NOT_VISIBLE )
    group.TerminateApplication()
```

Figure 4.13: Using the transactional semantics of the API to show two content items as an atomic unit.

We are also able to provide guarantees with respect to *isolation* in two distinct domains – system state and content visibility. In terms of system state, changes to an Application's or a Display's internal state that are made within the context of a transaction are visible only within that transaction until the transaction is finally committed. Crucially, we are also able to support isolation as it applies to the actual physical visibility of operations on public displays. We call this form of isolation *visual isolation*. Visual isolation guarantees that humans observing the public display system will not be able to witness any intermediate states before the transaction is committed. This means that any changes to the visibility of content on public displays resulting from operations that are part of a transaction are held back until the transaction is committed.

Moreover, our transactions provide *consistency*, guaranteeing that the results of transactions do not to violate the policies and constraints of our software infrastructure. Concrete examples for such constraints are: the limitation that each Display can have no more than one Application in state `VISIBLE` at the same time, the policy that only higher-priority content is able to preempt a Display, and the set of supported state transitions that Application processes may undergo as part of `ChangeState` operations.

4.6.3 Examples

In this section we present two examples that show how the scheduling API presented above can be used in concrete scheduling scenarios.

Figure 4.14 demonstrates the use of the API to create a Scheduler that shows a news bulletin every hour on the hour that is directly followed by the weather forecast. The Scheduler enters an infinite loop in which it waits until 5 minutes to the full hour, at which time it instantiates Applications for the news bulletin and the weather forecast using calls to `CreateApplication`. The Scheduler requests the content associated with both Applications to be pre-fetched using `ChangeState`, after which the Scheduler waits until exactly on the hour. On the hour the Scheduler requests the news bulletin to be made visible using a call to `Transition`, and waits until the end of the news bulletin. The Scheduler then replaces the news bulletin content with the weather content. The Scheduler achieves this by requesting the Application instance associated with the news bulletin to be made `NOT_VISIBLE` using the `Transition` operation, and by requesting the Application instance associated with the weather content to be made `VISIBLE`. After the weather

content has been shown, the Scheduler uses the `Transition` operation to make the weather content not visible, and subsequently terminates both Application instances using `TerminateApplication`.

```
while True:

    sleep until 5 minutes to the hour

    # instantiate and pre-fetch news content
    ( result, news_app ) = display1.CreateApplication(
                                "http://host/the_news.mpeg" )
    news_app.ChangeState( PREPARED )

    # instantiate and pre-fetch weather content
    ( result, weather_app ) = display1.CreateApplication(
                                      "http://host/the_weather.mpeg" )
    weather_app.ChangeState( PREPARED )

    sleep until exactly on the hour

    display1.Transition( news_app, VISIBLE, DEFAULT_PRIORITY )

    sleep for the duration of the news bulletin

    display1.Transition( news_app, NOT_VISIBLE )
    display1.Transition( weather_app, VISIBLE, DEFAULT_PRIORITY )

    sleep for the duration of the weather bulletin

    display1.Transition( weather_app, NOT_VISIBLE )

    # terminate both applications
    display1.TerminateApplication( news_app )
    display1.TerminateApplication( weather_app )
```

Figure 4.14: Example of a Scheduler showing a news bulletin followed by the weather forecast on the hour every hour.

Figure 4.15 demonstrates the use of the scheduling API to show Metamorphosis in the Underpass. As we described earlier, Metamorphosis consisted of three individual pieces of content that were to be shown side-by side on the three display in the Underpass. The Scheduler therefore makes use of the transactional semantics provided by our API to ensure that any part of Metamorphosis is only made visible if all three parts can be made visible. Having created Application instances for the content on each of the three displays, the Scheduler adds the Application instances to an Application Group, enabling the Scheduler to control all three content items using group operations. The Scheduler uses a call to `ChangeState` on the Application Group to request all three Application instances to pre-fetch their content, and subsequently requests all instances to be made visible using a `Transition` operation with group semantics. Since the re-

quest is carried out inside a transactional block, the changes to the visibility of the Application instances will not be carried out until the transaction is committed. Once the transaction has been committed, the Scheduler waits for a pre-determined period of time, during which Metamorphosis remains visible on the displays. After this period if time, the Scheduler removes Metamorphosis from the displays by requesting the Application instances to be made `NOT_VISIBLE`, and by using `TerminateApplication` to cause the Application instances to be terminated.

```
group = new ApplicationGroup()
BEGIN TRANSACTION
    # instantiate the Metamorphosis content on all three displays
    ( result, app_left )   = display1.CreateApplication(
                                "file:///metamorphosis_left" )
    ( result, app_centre ) = display2.CreateApplication(
                                "file:///metamorphosis_centre" )
    ( result, app_right )  = display3.CreateApplication(
                                "file:///metamorphosis_right" )

    # add Applications to an Application Group
    group.addApplication( app_left )
    group.addApplication( app_centre )
    group.addApplication( app_right )

    # pre-fetch content
    group.ChangeState( PREPARED )

    # show Metamorphosis
    group.Transition( VISIBLE, DEFAULT_PRIORITY )
COMMIT TRANSACTION

if transaction was successful:
    # leave Metamorphosis on the displays for a certain period of time
    sleep for a while

    # remove Metamorphosis from the displays
    group.Transition( NOT_VISIBLE )
    group.TerminateApplication()
```

Figure 4.15: Example of a Scheduler showing Metamorphosis in the Underpass.

4.7 Summary

In this chapter we have presented the design of a computational model and accompanying API for controlling the life-cycle and presentation of content on public display networks based on the requirements outlined in chapter 3. The computational model is built around the notions of *Displays*, *Applications*, *Schedulers*, *Handlers*, *Interaction Devices*, and *Context Sources*. It equips experi-

menters with a clear set of abstractions for developing experiments and reasoning about state of the public display system. The API provides experimenters with a small set of operations for controlling the life-cycle of content on individual displays and for controlling its visibility. Transactional semantics and operations on Application Groups allow for the creation of orchestrated experiments spanning multiple displays. The set of operations presented in this chapter provides support for writing a diverse range of scheduler types, including constraint-based schedulers that might or might not construct schedules in advance, and schedulers that directly schedule content in response to incoming events, such as input events received from Interaction Devices or context events received from Context Sources. The following chapter provides a description of the engineering of the computational model to provide a low-level software infrastructure and a low-level scheduling API.

Chapter 5

Engineering Model

5.1 Introduction

In the previous chapter we presented the design of a distributed systems infrastructure and accompanying API for controlling the presentation of content on public display networks. This chapter focuses on the engineering of the software infrastructure and specifically describes how the computational entities can be mapped to processes and how these processes can be distributed onto the machines in a public display network. Moreover, we describe the design of a distributed protocol for interconnecting these processes. We start by providing an overview of the processes and their distribution. We then discuss the design choices that underpin these mappings, followed by a detailed description of the protocol.

5.2 Overview

The *engineering model*[1] represents a particular mapping of the computational model presented in chapter 4 onto machines and processes in a public display network. An overview of the engineering model is provided in figure 5.1.

[1] our interpretation of the term "engineering model" is based on the terminology defined in the context of the ISO Reference Model of Open Distributed Processing (RM-ODP) [ISO95]

Displays, Applications, Handlers and Schedulers are all modelled as separate processes. The processes communicate using a request/response protocol engineered on top of a publish/subscribe-based event channel that allows subscriptions to be made based on the content of events.

Each conceptual Display is managed by a Display process that is hosted on the computer that is underpinning the conceptual Display. Display processes are responsible for instantiating Application processes, for detecting conflicts between the Application processes they manage, and for overseeing the visibility of the output created by Application processes.

Handler processes may in principle be hosted on any machine in the display network. However, the requirement to interface with lower-level hardware, for example to be able to power a physical display on or off, may require Handler processes to be executed on specific computers.

Application processes are co-located with Display processes on the same computer. To facilitate the development of new types of Applications, the functionality of each Application has been separated into two entities: a reusable protocol engine, and a content-type specific part that is responsible for producing the output on the Display, based on the instructions provided by Scheduler processes and Display processes. Moreover, the content-type specific part is responsible for controlling the placement and size of the produced output on the physical display.

The operations exposed by Application processes and Display processes are made available to Schedulers in the form of an RPC-style API. Like all inter-process communication in our software infrastructure, the API is engineered using a request-response protocol on top of the event channel. Schedulers can therefore be hosted on any machine as long as this machine is able to establish connectivity to the event channel (see design proposition DP2).

All processes in our engineering model periodically emit status messages in the form of events. These status messages are used to publish the attributes associated with each entity type. Processes also emit status messages whenever the value of one of their attributes changes. Any process that is capable of connecting to the event channel is able to subscribe to and observe these status messages. By using appropriate subscriptions, processes are also able to receive and observe the exchange of protocol messages. As a result, specialised tools can be developed that use these properties to monitor the activities in the display network or to create audit trails by logging status and protocol messages, addressing requirement R14.

Information from Interaction Devices and Context Sources may be accessed in two different ways. The information can be disseminated in the form of events, allowing any process in the public display network to subscribe to and receive these events and enabling schedulers that wish to receive this information to be executed on any machine in the display network (see design proposition DP2). In this case, gateway processes are used to retrieve the context or interaction information from the underlying devices using proprietary APIs and to publish this information in the form of events. Alternatively, processes may directly interface with Context Sources and Interaction Devices using proprietary APIs. Using either method, researchers are able to construct context-sensitive or interactive experiments (see requirements R8 and R9).

In the following sections we provide a detailed description of both the processes that constitute the low-level infrastructure, and the protocol that underpins the communication between these processes.

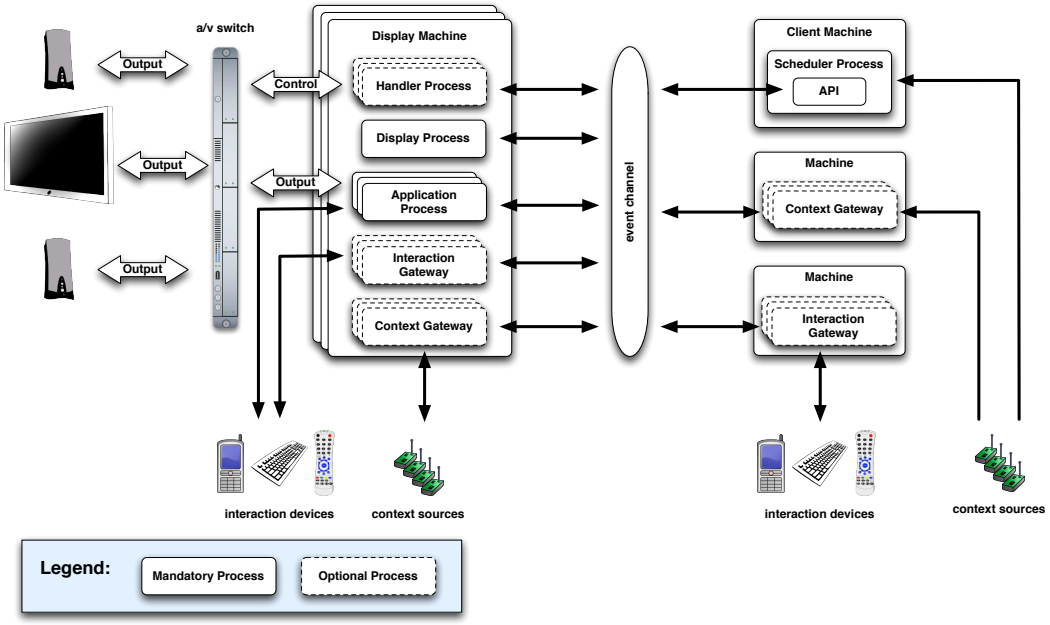


Figure 5.1: Distribution of functionality and processes.

5.3 Processes and Their Distribution

In this section we describe the mapping of computational entities to processes in our engineering model and explain the distribution of these processes onto machines in a public display network.

5.3.1 Application Processes

As described in chapter 4, Applications are responsible for rendering content onto Displays. Moreover we have argued in chapter 3 that the software infrastructure is required to provide support for proprietary content and applications that, for example, use native APIs to interface with sensor platforms and actuators (see requirement R11). These APIs may only be available for specific programming languages. We therefore cannot assume that all experimental Applications are implemented using the same programming language. As a result we propose to engineer Applications (which represent such experimental applications in our computational model) as separate processes that can be implemented using the researchers' programming languages of choice.

Distribution of Application Processes

The decision where to host Application processes is limited by the main function of Applications to produce output on public displays. Solutions exits, such as the X display protocol [SG86] or the RFB protocol [Ric07], that allow processes that produce graphical output to be hosted on remote machines. In this model, the output produced by these processes is transferred over the network to the machines where it is to be displayed. However, these solutions only provide limited frame rates and are typically not suitable for videos or animated content with a suitable quality. Moreover, the graphics hardware of most of today's computers feature accelerated graphics output based on in-silicon implementations of graphics APIs. For example, OpenGL [Ope08] hardware support often provides accelerated 3D output. However, these accelerations are not available if output is produced on a remote machine: hardware accelerated output is generated by the actual graphics hardware and can therefore not be accessed by software solutions, such as X or RFB. It is possible to redirect the graphics signal to a video capture card that converts the signal back onto a digital video stream that can be sent over the network. However, the bandwidth requirements of streaming uncompressed, high-resolution, high-frame-rate videos over the network limit the scalability of this solution as the number of public displays and Application processes increases. Compressing the videos before they are streamed places high requirements on the CPU of the compressing system and typically leads to delays between the capturing of input and the manifestation of the corresponding stream output. As a result, we propose to execute Application processes on the actual display machines.

Separation of Functionality

The functionality of an Application process can be further decomposed into two parts:

- a *rendering engine* that is specific to the content type supported by the Application. For example, there may be rendering engines that are able to display video files, and other rendering engines suitable for showing web content.
- a *protocol engine* that implements the state machine that underpins the Application process and that exposes the `ChangeState` operation to Displays and Schedulers. The protocol engine is also responsible for realising the intended transactional semantics, including resource locking, and the handling of `commit` and `abort` protocol messages (see section 5.4.2).

While support for different content types calls for the development of different rendering engines for each of these content types, the protocol engine is generic across all Applications and can therefore be re-used for the construction of different types of Applications. In our engineering of Applications we propose to follow this separation: each Application process is composed of a generic *protocol wrapper* that is common for the engineering of all Application processes that are implemented using the same programming language, and a *content-type-specific renderer*. Application processes are formed by combining one renderer with the appropriate protocol wrapper. Renderers expose an interface that is common to all renderers, which enables arbitrary renderers to be combined with the generic Application protocol wrapper without requiring any modifications to that wrapper. This interface consists of the following set of methods:

- a constructor, accepting the url of the content as argument.
- `prepare`, causing the renderer to pre-fetch the content.
- `make_visible` to make the renderer's output visible on the display, and to instruct the renderer to start playback if continuous media is used.
- `make_not_visible` to make the renderer's output not visible and to instruct the renderer to pause playback.
- `terminate`, causing the renderer to clean up its internal state and to prepare for termination.

To implement transactional semantics (see design proposition DP6), all methods accept a 'commit' flag as a parameter. This flag indicates whether the call is part of a prepare phase (i.e. whether the mere possibility of executing the action should be checked) or whether the action should be committed (i.e. the action should actually be carried out). As a result a `ChangeState` operation that is carried out in a transactional context typically triggers two invocations of the corresponding renderer method: a first time with cleared commit flag during prepare phase, and a second time when the transaction is committed, this time with the commit flag set. The act of aborting a transaction does not modify the playback state or visibility state of renderers. To simplify the engineering of renderers, our current engineering model does therefore not provide any means for notifying renderers if transactions are aborted. However, we do acknowledge that as a result renderers may reserve and hold on to resources (for example as part of state transitions into Application state `PREPARED`) and we would therefore expect future versions of the engineering model to provide means for notifying renderers if transactions are aborted.

All operations except for the constructor return a result code indicating whether the invocation was successful.

5.3.2 Display Processes

Each machine in an installation typically offers one or more conceptual Displays (as discussed in chapter 4). Since Displays are responsible for instantiating Application processes, at least those functional parts of Displays that are in charge of creating Application processes have to be hosted on those physical machines that Applications are to be instantiated on.

The other functions that are performed by Displays (i.e. the oversight of the visibility of Applications, the communication with Handlers, and the detection of conflicts between Applications on the same Display) can in principle be provided by processes that are executed, for example, on a remote server. However, the distribution of the functionality provided by Displays into separate processes would result in additional overheads incurred by the communication between these processes. We therefore propose to represent each Display entity in the computational model as a single Display process in the engineering model. Each Display process implements all of the functions of a Display entity. Moreover, Display processes are hosted on the particular display machines that are managed by them.

5.3.3 Handler Processes

The engineering of Handlers as separate processes is partly dictated by the potential need for Handlers to interface to specific hardware components, such as audio/video matrix switches (see requirement R15), whose APIs may only be available for certain programming languages. It is therefore unlikely that all Handlers could be implemented in a way that enables them to be integrated directly into Display processes, e.g. in the form of modules. Moreover, according to our computational model, single Handler instances may be associated with multiple Displays. This means that if Handlers were integrated into Display processes, then Handler instances would have to implement two different mechanisms for communicating with Display processes: a native interface for communication with the Display process they themselves are part of, and a remote interface for communicating with other Display processes in the system that the Handlers are associated with. This remote interface may, for example, be based on direct communication between remote Display processes and the Handler. Alternatively, Handler stubs may be associated with each remote Display, and a remote Display may in this case communicate with a Handler by invoking operations on its own Handler stub, which as a result interacts with the (remote) Handler.

To simplify the construction of Handlers we therefore propose to engineer Handlers as separate processes that communicate with Displays using a protocol that is layered on top of the event channel. In our engineering model, each individual Handler process is addressable using a globally unique *Handler Identifier*. The association between Handler processes and Display processes is read and parsed by Display processes on start-up. Changes to this configuration therefore require Display processes to be restarted.

By allowing Display processes and Handler processes to communicate using the event channel, Handler processes can in principle be hosted on any computer in the display network, as long as a connection to the event channel can be established. However, the need to interface with specific hardware that cannot be accessed using remote invocations may limit the ability to freely distribute specific Handlers. For example, some LCD displays can be switched on and off using commands that are issued over an RS-232 connection. A Handler can be constructed that is responsible for switching such a display on or off based on whether content is currently visible on the display or not. It makes sense to host such a Handler on the machine that the display's RS-232 interface is

connected to.

5.3.4 Scheduler Processes

Design proposition DP2 proposes to allow Schedulers to be distributed arbitrarily in the display network. As a result we propose to engineer Schedulers as separate processes. The API that Schedulers use to influence the life-cycle and visibility of Applications uses the event channel to communicate with Display processes and Application processes. As in the case of Handler processes, Scheduler processes can therefore in principle be hosted on any machine in the display network. This means that researchers can, for example, execute their experiment-specific Schedulers on their own computers, as long as connectivity to the event channel can be established from these machines. However, the need, for example, to interface specific hardware, such as sensors, that can only be accessed from certain machines may require certain Schedulers to be executed on specific computers.

5.4 Protocol

The operations exposed by processes are invoked using a request/response protocol that is layered on top of a distributed publish/subscribe-based event channel, such as Elvin [SA97] or Siena [CRW01]. A taxonomy of the features of distributed event systems has been published by Meier and Cahill [MC05].

By using an event-channel to exchange protocol messages we can allow additional processes to observe this message exchange. As a result, tools can be created that monitor the system or create audit trails by observing the exchange of protocol messages between the processes in the software infrastructure. Such tools may, for example, provide information about when and for how long specific experiments were shown on individual public displays by observing `Transition` operations. Our survey of existing public display systems and applications in chapter 2 showed a need for our software infrastructure to support the creation of audit trails (see requirement R14).

For the engineering of the protocol we expect the event channel to have the following two properties: firstly, event production and subscription are both undirected, i.e. processes wishing

to receive events do not require knowledge of the addresses of producers, and producers don't require knowledge about the addresses of subscribers. As a result neither producers nor consumers are required to maintain an overview of other (potential) communication partners. Secondly, subscriptions can be expressed based on the content of events, e.g. allowing subscribers to receive all events that contain a field named "display_id" whose value matches "display-01". Instead of content-based subscription, some event distribution platforms only allow topic-based subscriptions. In these systems, producers mark events as belonging to one or more topics. Consumers subscribe to one or more topics, causing them to receive all events belonging to topics they have subscribed to. However, we believe that topic-based subscriptions do not allow precise enough control over subscriptions to allow consumers to receive only events that they are interested in. Moreover, producers do not necessarily know what criteria event consumers are interested in. It is therefore difficult for a producer to determine how an event should be classified into topics. Even if such a classification was achievable more easily, it is unclear how the semantics of the created topics can be communicated to consumers of events. All these issues can be circumvented if subscriptions can be made based on the content of events, rather than topics.

In our software infrastructure a process wishing to call an operation exposed by another process emits a request event on the event channel. The generic structure of a request event is shown in table 5.1. All protocol events carry a field named `event_type` that corresponds to the operation that is to be invoked. Possible event types for invoking operations are `CREATE_APPLICATION`, `CHANGE_STATE`, `TRANSITION`, `TERMINATE_APPLICATION` and `HANDLE`. Responses are communicated using events of type `"RESULT"`. Each request event carries a globally unique *Request Identifier* (`request_id`) that is echoed in any response event corresponding to the request. The Request Identifier allows processes to correlate response messages to their original request message.

All request events carry addressing information specifying which processes the operation should be invoked on. `CREATE_APPLICATION`, `TRANSITION` and `TERMINATE_APPLICATION` events are targeted at Display processes and hence include a Display Identifier in the form of a `display_id` field. `CHANGE_STATE` events contain an Application Process Identifier (`pid`) specifying the Application process that the event is intended for. Finally, `HANDLE` events feature a `handler_id` field carrying the Handler Identifier of the targeted Handler.

As we will see in section 5.4.1, if `CHANGE_STATE`, `TRANSITION` and `TERMINATE_APPLICATION` requests are addressed to Application Groups, they contain a `group_id` field instead of `display_id`

Field Name	Type	Description/Value
`event_type`	string	Specifies the operation that is to be invoked. Can be one of: `"CREATE_APPLICATION"`, `"CHANGE_STATE"`, `"TRANSITION"`, `"TERMINATE_APPLICATION"`, `"HANDLE"`.
`request_id`	string	A globally unique identifier allowing the caller to correlate response messages with requests.
`display_id` or `pid` or `handler_id` or `group_id`	string	A Display Identifier, Application Process Identifier or Group Identifier specifying which process or group of processes this request is addressed to.
`trans_id`	string	If the invocation is part of a transaction, then this field contains the Transaction Identifier of that transaction. Otherwise the value of this field is set to `"0"`.
additional operation-specific arguments in the form of (name,value) pairs ...		

Table 5.1: Generic structure of a protocol request.

Field Name	Type	Description/Value
`event_type`	string	"RESULT"
`request_id`	string	The Request Identifier of the request this event is a response to.
`display_id` or `pid`	string	This field is only used if the corresponding request was targeted at an Application Group. In this case a Display Identifier or an Application Process Identifier is provided to identify the process that emitted the response.
`result`	string	The result code, indicating whether the requested operation was carried out successfully. Can be one of `"OK"` or `"FAILED"`.
`reason`	string	If the value of the `result` field is `"FAILED"`, this field indicates why the operation invocation has failed. Examples for possible values include: `"OPERATION_NOT_SUPPORTED"`, `"MALFORMED_REQUEST"`, `"CONFLICTING_CONTENT"`, `"IN_TRANSACTION"`, `"ILLEGAL_TRANSITION"`.
additional operation-specific response codes in the form of (name,value) pairs ...		

Table 5.2: Generic structure of a protocol response.

and `pid` fields. The `group_id` field contains the Application Group Identifier of the targeted Application Group.

All request events contain a `trans_id` field. If the request is not part of a transaction, the value of this field is set to `"0"`, otherwise it holds a globally unique *Transaction Identifier* that is used to identify the specific transaction in the system.

In addition, request messages contain operation-specific name-value pairs that are used to transport the arguments of operation invocations. We provide a detailed description of these arguments in sections 5.4.4, 5.4.5 and 5.4.6.

A process receiving a request message processes the operation invoked by the request and returns a response message. An abstract overview of the actions taken by the receiver of a request is shown in figure 5.2. Normally, when a request message is only directed at a single process (i.e. the operation is not to be carried out on an Application Group), each request message is answered by the targeted process using a single response message. Multiple response messages are only expected if requests are addressed to Application Groups (see section 5.4.1).

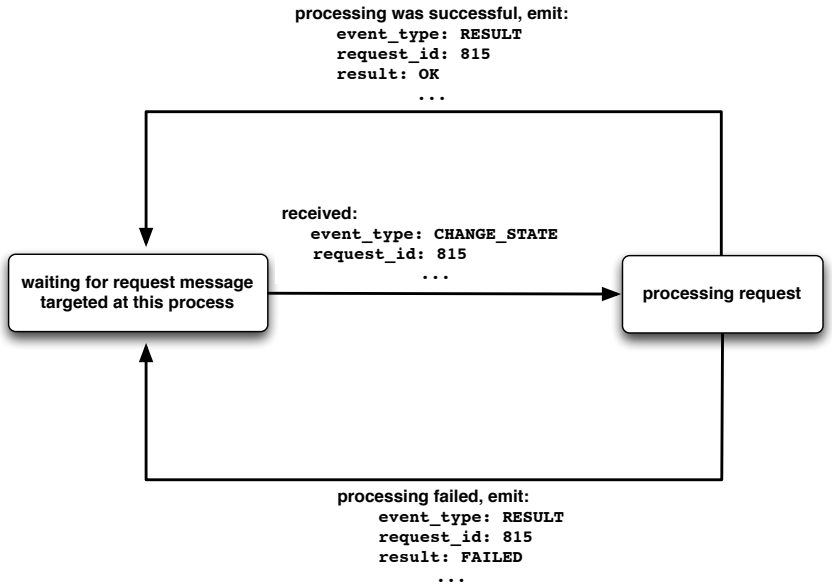


Figure 5.2: Protocol actions taken by the receiver of a request message.

The generic format of a response message is depicted in table 5.2. Each response message has a `response` field indicating whether the operation invocation has been successful or not. Possible values are `"OK"` and `"FAILED"`. In case of a failed invocation, further information about the reason for

the failure may be returned in the `reason` field of the response message. Examples for error codes include `"CONFLICTING_CONTENT"` to indicate that a `Transition` operation failed since the public display was already showing content that could not be pre-empted, or `"MALFORMED_REQUEST"` to indicate that the request message could not be parsed by the targeted process. Moreover, the response message may contain additional request-specific fields that are used to communicate the results from an operation invocation back to the requester.

If the requester fails to receive a response from the targeted process within a certain time-out period, the operation invocation is considered to have failed. The protocol actions for a sender of request message are shown in figure 5.3.

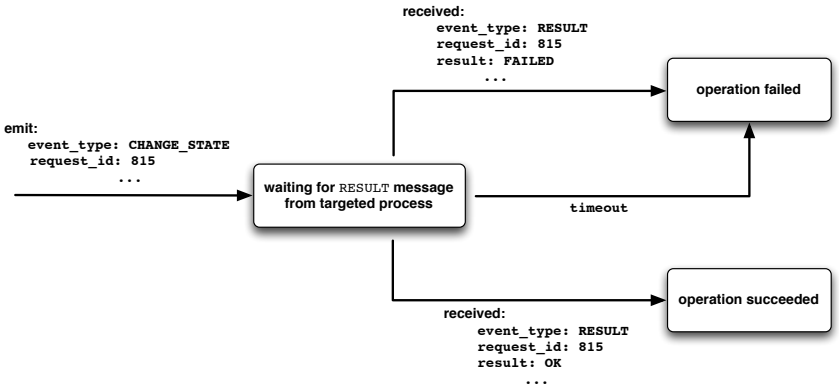


Figure 5.3: Protocol actions taken by the sender of a request message.

5.4.1 Operations on Application Groups

Support for operations on groups of Applications (see requirement R6) can theoretically be implemented by the API without requiring support from the underlying protocol. Operations involving Application Groups can be split up by the API into separate invocations on the protocol level: one for each involved process. The API collects the responses from these processes, aggregates the responses, and returns the aggregated result to the caller. The operation is considered successful if responses have been received from all members of the Application Group and if all members have responded with a positive result code (i.e. `"OK"`). Otherwise the bulk operation is considered to have failed. Figure 5.4 shows an example of this procedure being applied to an invocation of the `ChangeState` operation on an Application group consisting of three Application processes. The API generates a separate `CHANGE_STATE` protocol request for each process and collects response messages from each process before returning the result to the Scheduler.

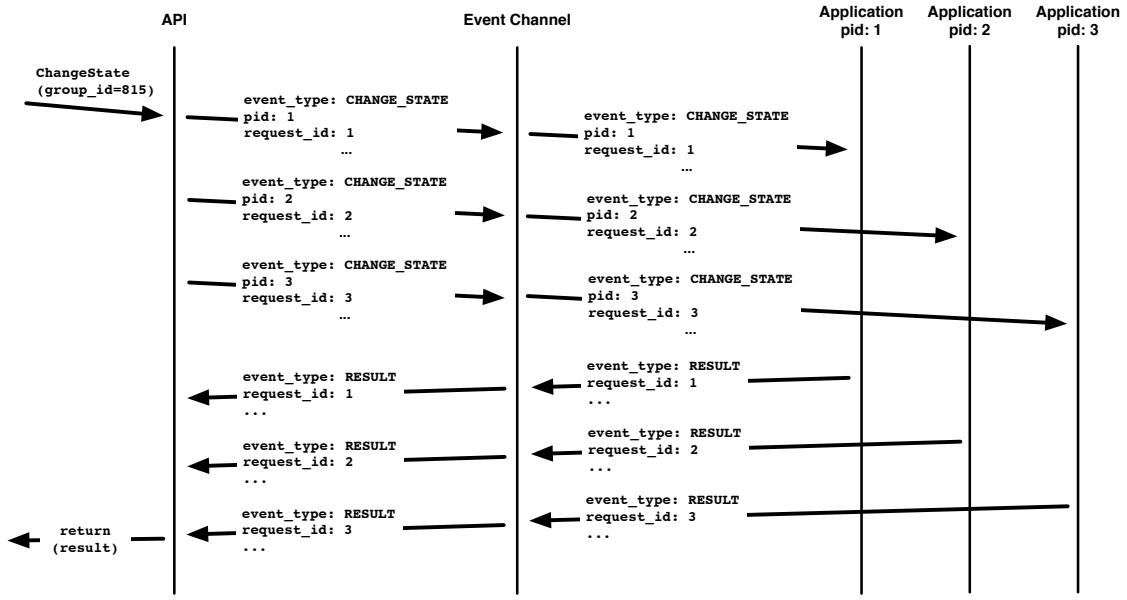


Figure 5.4: Possible engineering of operations on groups of Applications.

This procedure can be further optimised if support for group operations is provided at the protocol level and information about the membership of Applications in Application Groups is disseminated to other processes in the system. Figure 5.5 shows such an optimised exchange, again using the example of a `ChangeState` operation. In this case, the involved Applications are aware that they are part of Application Group 815. Instead of emitting one `CHANGE_STATE` event for each group member, the API instance is therefore able to emit a single event, addressed to the whole group. Applications that are part of Application Group 815 are able to subscribe for `CHANGE_STATE` events that are addressed to that group, i.e. events that carry a `group_id` field whose value is set to 815. As in the non-optimised case, each Application replies by emitting a response event (i.e. the `event_type` field is set to `RESULT`). Since only one request event is emitted by the API instance, the `request_id` can no longer be used to associate response events with individual processes. `RESULT` events that are sent as response to operations on Application Groups therefore carry an additional field that identifies the process that emitted the response. In case of a `CHANGE_STATE` request, each response message is generated by an Application process. Each response therefore features a `pid` field that holds the Application Process Identifier of that process. A response corresponding to a `CREATE_APPLICATION`, `TRANSITION` or `TERMINATE_APPLICATION` request provides a `display_id` field containing the Display Identifier of the Display process that processed the request.

Bulk operations using Application Groups are targeted at providing developers with the ability

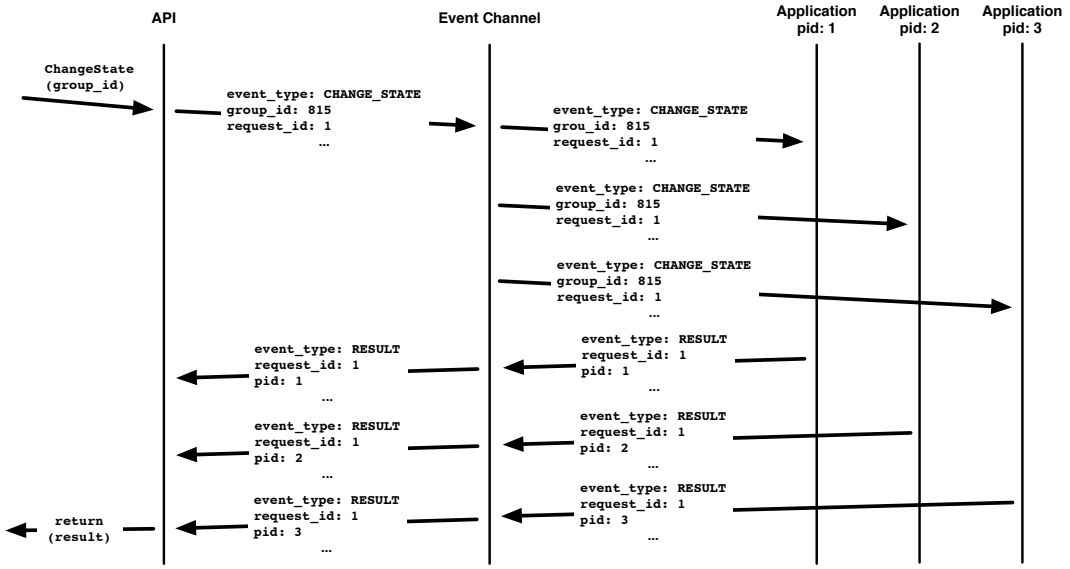


Figure 5.5: Optimised engineering of operations on groups of Applications.

to control Application processes that are hosted on different Displays, and in general we do not expect bulk operations to be used to control Application processes that reside on the same Display. We therefore do not employ the same optimisations to `CHANGE_STATE` requests that are emitted by Display processes as a result of incoming `TRANSITION` or `TERMINATE_APPLICATION` requests. Instead, if an operation that is carried out using an Application Group applies to more than one Application on a single Display, the Display will disseminate one `CHANGE_STATE` request to each of the Applications involved. An example outlining the communication between Display processes and Application processes in the context of a `TerminateApplication` operation performed on a group of four Applications is depicted in figure 5.6.

Managing Application Groups

In our engineering of the software infrastructure, Application Groups are created by Schedulers by instantiating a new `ApplicationGroup` data structure. We do not provide any separate API operations for adding Applications to Application Groups, or for removing them from them. Instead, group management is piggy-backed onto `CreateApplication` and `TerminateApplication` API calls. The `CreateApplication` operation that is exposed by the API to Schedulers accepts an optional Application Group Identifier as argument. If this argument is provided by the Scheduler, the Application that is instantiated by the `CreateApplication` operation is automatically

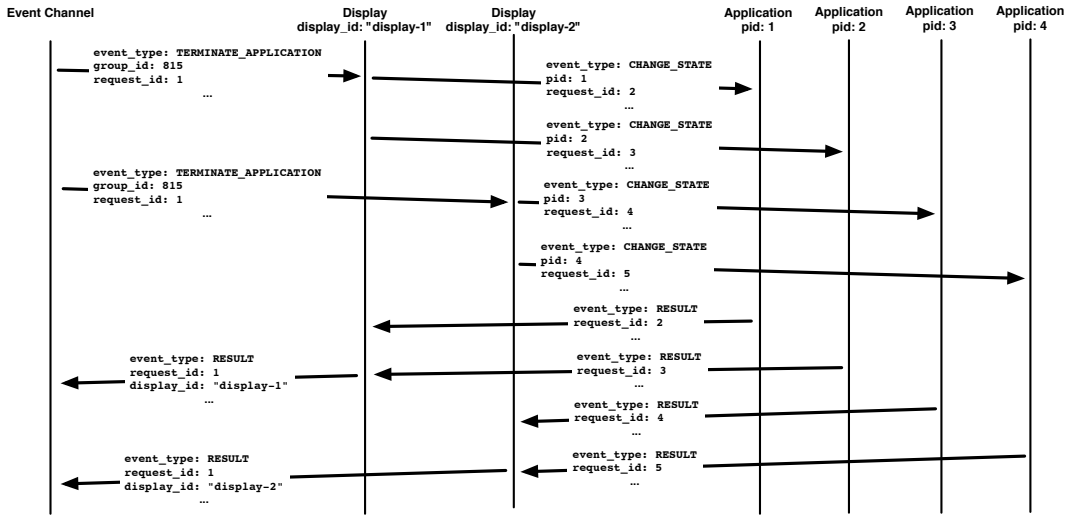


Figure 5.6: Communication between Display processes and Application processes in the context of operations using Application Groups.

added to the Application Group. Terminating an Application instance using `TerminateApplication` implicitly removes an Application that is member of an Application Group from that group. Application Groups cease to exist once all of its members have been terminated and once the data structure representing the Application Group in the API instance is no longer referenced by the Scheduler (i.e. the data structure holding the identifier is automatically garbage collected or explicitly destroyed). The example below shows an example of an Application that is added to an Application Group as part of the `CreateApplication` invocation.

```
...
group = new ApplicationGroup()
(result, app1) = display1.CreateApplication( "http://a.host/a_video.mpeg",
                                             group )
group.ChangeState( PREPARED )
...
```

If a Scheduler requests an Application to be added to an Application Group (by specifying an Application Group as argument to the `CreateApplication`), a `group_id` holding the group's identifier is added to the corresponding `CREATE_APPLICATION` request event. The receiving Display process passes the group identifier on to the Application processes during the Application's instantiation, and thereby informs the Application process about its membership in the group. This also allows the Application to subscribe for events that are addressed to that group. The Display process records the association between the newly created Application process and subscribes to `TRANSITION` and `TERMINATE_APPLICATION` request events addressed to the Application

Group. An overview of how membership information is communicated between processes as part of a `CreateApplication` operation is shown in figure 5.7.

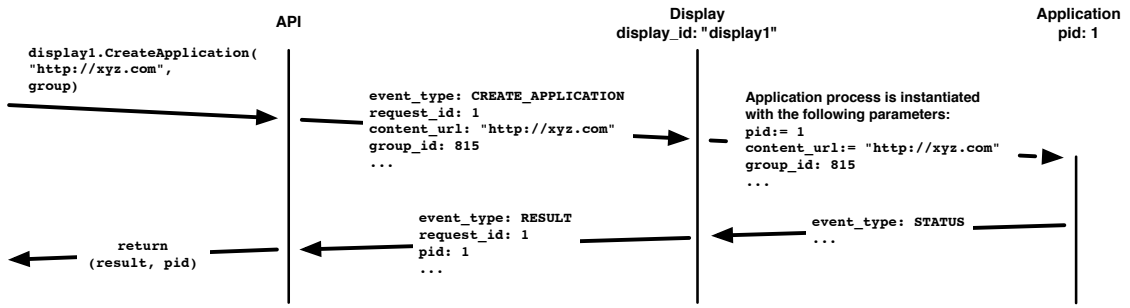


Figure 5.7: Example showing an Application instance being added to an Application Group.

Instead of this simple, macro-like engineering, of support for Application Groups, alternative forms of engineering group support are possible, e.g. based on the facilities for reliable group communication provided by the ISIS toolkit [BJ87, GBCv93].

5.4.2 Transactions

Transactions are engineered using a simplified version of the classic two-phase commit protocol [Gra78, LS79] that is commonly used for engineering distributed transactions. During the first phase (the preparation phase) of the protocol the processes that are involved in the transaction check whether the operations that are part of the transaction can be carried out successfully in the order that they were invoked in. If an operation can be carried out without errors, it is brought into a committable state. Once all operations that are part of the transaction are in a committable state, the transaction enters its second phase (the commit phase) during which the operations are committed.

As we have outlined in chapter 4, our transactions do not provide any guarantees with respect the durability of the results. Moreover, by employing a simplified two-phase commit protocol we do not guarantee atomicity in all failure cases. We discuss the simplifications and their implications later in this section.

Design Considerations

The requirement to isolate transactions from operations that are carried out by other Schedulers influences the actions that can be performed during the preparation phase. Any actions that are performed during this phase are required to remain hidden from other processes that are not part of the transaction. Additionally effects of these actions are required to remain hidden from humans observing the public displays until the transaction is committed. Specifically this means that during the preparation phase of a transaction, Displays are able to instantiate Application processes as a result of `CREATE_APPLICATION` requests, and Application processes can pre-load content if they receive a `CHANGE_STATE` request into target state `PREPARED`. Neither of these actions are visible to other processes in the system or to humans observing the public displays.

Processes that perform actions during the preparation phase are required to log these actions so that they can be rolled back if the transaction is aborted. If the actions associated with an operation cannot be fully executed during the preparation phase, these actions have to be queued up and performed once the transaction is committed. Examples include changes to the visibility of content as a result of `Transition` and `ChangeState` operations.

The requirement to provide isolation (see requirement R13) also influences how process that are involved in a transaction handle requests that are performed concurrently by other Schedulers outside the current transaction. Such requests can, for example, be non-transactional, or be part of another transaction. To isolate the transaction from these requests, processes generally have to ensure that operations that are performed outside the context of the current transaction do not affect the outcome of operations within the transaction that are already in a committable state. For example, if during the preparation phase a `Transition` operation with a target state of `VISIBLE` is brought into a committable state, then the Display process has to ensure that once the transaction is committed, the Application in question can be brought into target state `VISIBLE`. This means, for example, that until the transaction is committed the Display process is required to prohibit other Schedulers from making content visible on the Display. Moreover, the Display, for example, has to prohibit `TerminateApplication` operations that are targeted at terminating the Application process in question, since a termination of that Application process would make it impossible to make the output produced by the process visible on the Display. In traditional database management systems, isolation is typically achieved with the help of locks that are placed

on the data that is to be modified by a transaction. These locks may be implemented with different levels of granularity, ranging from prohibiting access to individual rows to locking access to whole tables or even databases. Similarly, the isolation semantics that we would like to achieve in our software infrastructure can be implemented in a number of different ways that differ in the level of granularity. The following levels are possible:

- action-level locking: actions that are to be performed as part of the processing of an operation invocation are checked against the actions taken by other operations that are already in a committable state.
- operation-level locking: certain operations are declared incompatible with other operations. This means that if an operation has been brought into a committable state as part of a transaction, invocations of incompatible operations that are performed outside the transaction will automatically fail.
- entity-level locking: if an entity is involved in an ongoing transaction, all other operation invocations are rejected by the entity. This also means that no other transactions can be started while the original transaction is still ongoing.

In our engineering model we use entity-level locking to isolate transactions from other operations or transactions, i.e. once a transaction has been started involving an entity, that entity is locked until the transaction has finished. Entity-level locking has the benefit that it allows us to keep the implementation relatively simple, since the complexity of identifying conflicts is low. However, the use of entity-level locking also means that conflicting transactions and operations potentially fail more quickly. Consider the following example: A Scheduler has started a transaction Display *display*1. As part of the transaction the Scheduler invokes `Transition` to make an Application visible on that Display. Before the Scheduler is able to commit the transaction, another Scheduler starts to invoke the following sequence of operations:

```
(result, app1) = display1.CreateApplication("http://a.host/a_video.mpeg")
app1.ChangeState(PREPARED)
display1.Transition(app1 , VISIBLE, DEFAULT_PRIORITY)
```

If entity-level locking is used, the first operation will immediately fail, since Display *display*1 is already involved in a transaction.

The use of entity-level locking also significantly reduces the complexity of providing consistency. Locking the entities involved in a transaction ensures that the state of these entities cannot be modified outside the transaction, and therefore eliminates concurrency as one of the potential sources for inconsistencies. The enforcement of consistency can therefore be delegated to the engineering of each individual operation, and no further actions for enforcing consistency are required at the transaction-level.

Transaction Protocol

Figures 5.8 and 5.9 provide an abstract overview of the state model that is used by API instances and other processes to process transactions.

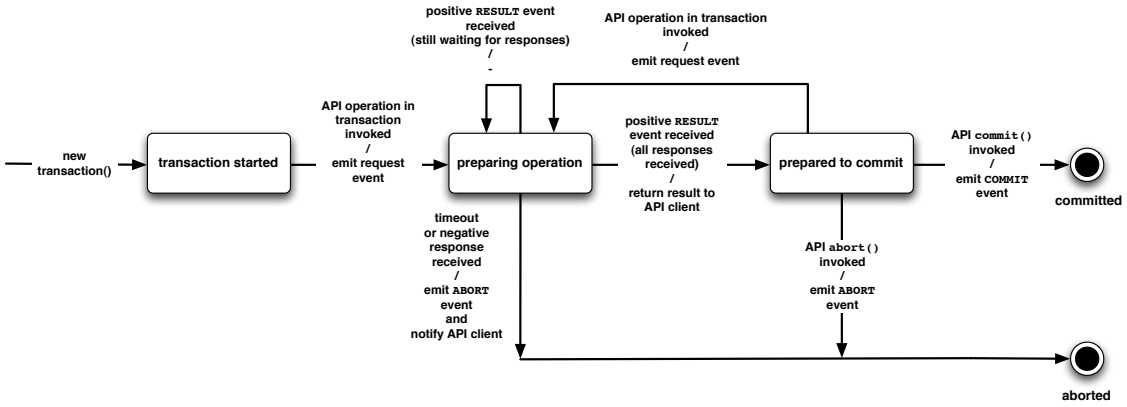


Figure 5.8: State model used by the API instance to process transactions.

Once a Scheduler has started a new transaction, the API enables operations to be invoked in the context of that transaction. When a new transaction is started at the API level, a globally unique Transaction Identifier is created. At the protocol-level, requests that are part of a transaction contain a `trans_id` field containing the Transaction Identifier of that transaction (see table 5.1). Requests that are not part of a transaction carry a `trans_id` value of "0". There is no explicit protocol message to start a new transaction. Instead, whenever a process receives a protocol message containing a `trans_id` value that the process has not seen so far, the process considers this message to mark the start of that transaction.

As in the non-transactional case, the invocation of an API operation in a transactional context causes the API instance to emit a corresponding request event. The API instance subsequently waits for response events from all involved processes. As we have described previously, in the

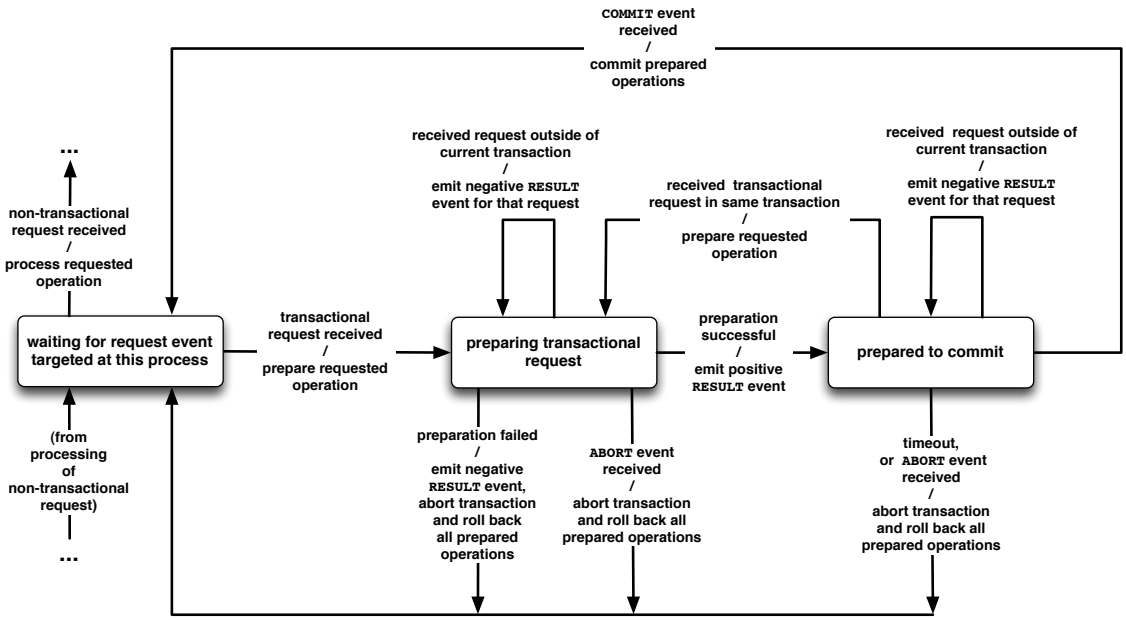


Figure 5.9: State model used by processes to handle transactions.

case of a request that is targeted at an Application Group the API instance expects a response event from each Application process that is a member of the group, or from each Display process that hosts an Application process that is part of the group. Once all response events have been received by the API instance, it returns the aggregated result of the invocation to the Scheduler. The Scheduler may subsequently invoke additional operations in the context of the transaction. If the API instance fails to receive one or more of the expected response messages within a certain time-out interval, it aborts the transaction by emitting an event of type `ABORT`. The API instance also notifies the Scheduler that the transaction has been aborted, e.g. by raising an exception. The same measures are taken if a negative response event is received from one of the involved processes. Transactions are committed or manually aborted by invoking `commit()` or `abort()` at the API level. The API instance reacts to such an invocation by sending out a `COMMIT` or an `ABORT` event that contains the Transaction Identifier of the targeted transaction. The format of `COMMIT` and `ABORT` events is shown in tables 5.3 and 5.4.

Requests that are received by a process in the context of a transaction cause the process to prepare the operation that is associated with the request using the semantics described above. If the preparation succeeds, the process emits a positive result event corresponding to the request. The process is now prepared to commit the transaction. However, if the preparation fails and the operation cannot be brought into a committable state, the process emits a negative response

event, aborts the transaction and rolls back all operations that might already be in a committable state. If the process receives an `ABORT` event during any state of the processing of a transaction, the process aborts the transaction and rolls back all prepared operations. The process also rejects all additional requests it receives that are not part of the ongoing transaction by emitting negative response events. If a time-out occurs while the process is in the "prepared to commit" state, the process aborts the transaction and rolls back all operations. The reception of a `COMMIT` event causes the process to commit the transaction and all prepared operations.

Comparison with the Classic Two-Phase Commit Protocol

As we mentioned in the introduction to this section, the protocol used to engineer our transactions is a simplified version of the classic two-phase commit protocol [Gra78, LS79]. The modifications allow for a simpler engineering of the protocol. However, as a result of these simplifications our protocol does not provide atomicity in the light of certain failure cases. We discuss the differences between the classic two-phase commit protocol and our simplified commit protocol in this section.

Provided the absence of node failures or losses of protocol messages, our protocol fully complies to the classic two-phase commit protocol. However, differences exist in the handling of node failures and the loss of protocol messages. In contrast to the classic two-phase commit protocol our commit protocol does not employ a recovery protocol. Processes that fail transiently restart without trying to recover to the state they were in before the crash. This means that if the processes were processing a transaction before the fault occurred, they do not attempt to recover the transaction.

Moreover, our simplified termination protocol allows processes to unilaterally decide to abort the transactions, if these processes are left in a state of uncertainty. This contrasts to the termination protocol employed by two-phase commit, in which processes inquire about the outcome of the transaction by contacting the coordinator or other participants.

We now describe the different failure cases and highlight the differences between our commit protocol and the classic two-phase commit protocol. We use the term *coordinator* to refer to the API instance from which the transaction originates, and the term *cohort* to refer to a Display process, Handler process or Application process participating in the transaction.

1. *time-outs in the coordinator process (i.e. the API instance) while trying to receive* `RESULT` *events from the cohorts.* In our commit protocol the time-out causes the coordinator to decide to abort the transaction and to disseminate an `ABORT` event to the cohorts. This behaviour is consistent with the behaviour of the classic two-phase commit protocol.

2. *time-out in the cohort process while trying to receive a* `COMMIT` *or* `ABORT` *event from the coordinator.* In our engineering, cohorts that find themselves in this state of uncertainty can unilaterally decide to abort the transaction. In contrast, using the termination protocol of the classic two-phase commit protocol cohorts attempt to communicate with the coordinator or other cohorts to obtain information about the outcome of the transaction. Other cohorts may respond in one of the following cases:

   - the other cohort has already sent a negative `RESULT` event in response to the transactional request. The cohort therefore knows that the transaction is going to be aborted.
   - the other cohort has already received a `COMMIT` or `ABORT` message from the coordinator, and hence knows about the outcome of the transaction.
   - the other cohort has not sent a `RESULT` event yet, and can therefore decide to cause the transaction to be aborted by sending a negative `RESULT` event.

   If neither the coordinator can be communicated with, nor any of the cohorts matching one of these conditions, the cohort remains in an uncertain state and blocks until the outcome of the transaction can be obtained.

   In practice this means that in the cases of both our simplified termination protocol and the classic two-phase commit protocol processes failing to receive an `ABORT` message will eventually abort the transaction. In case of our simplified termination protocol processes unilaterally abort after a time-out period, while in case of the classic two-phase commit protocol processes block until they have obtained information about the outcome of the transaction.

   However, since our simplified termination protocol also allows processes that fail to receive a `COMMIT` protocol message to unilaterally abort the transaction after a time-out period, these processes will as a result not implement the state changes that were requested by the transaction. Our simplified protocol does therefore not provide atomicity in this particular failure case, potentially leading to differences between the intended visibility and the actual visibility of content on the affected displays. In contrast, in the case of the classic two-phase

commit protocol processes remain blocked until they have obtained information about the outcome of the transaction.

3. *transient failure of the coordinator after sending the transactional request, but before sending a `COMMIT` or `ABORT` event.* Our commit protocol does not recover from such failures. As a result, cohorts will experience time-outs while waiting for a `COMMIT` or `ABORT` event from the coordinator and will unilaterally abort the transaction. In the classic two-phase commit protocol, the coordinator recovers the state of the transaction from its log file and subsequently decides to abort the transaction by emitting an `ABORT` event.

4. *transient failure of a cohort after receiving the transactional request, but before sending a `RESULT` event.* Since we do not implement any recovery protocol, cohort processes simply restart, but do not recover the state of the transaction. As a result these processes will never emit a `RESULT` event, causing the coordinator to experience a time-out while waiting for a `RESULT` event from that cohort. The coordinator will therefore abort the transaction and disseminate an `ABORT` event. In the context of the classic two-phase commit protocol, cohorts recover the transaction using the persistent log and subsequently emit a negative `RESULT` event, causing the transaction to be aborted.

5. *transient failure of a cohort after sending a `RESULT` event, but before receiving a `COMMIT` or `ABORT` event.* In case of the protocol we use to engineer our transactions no recovery is attempted, i.e. even if the coordinator has made the decision to `COMMIT`, this outcome won't be reflected by the cohort process. In the classic two-phase commit protocol failed cohorts recover the state of the transaction from their logs when the cohorts restart. They subsequently run the termination protocol, i.e. they attempt to obtain information about the outcome of the transaction by communicating with the coordinator or other cohorts. Recovering cohorts remain blocked until this information has been obtained.

   This means that if our simplified protocol is employed, processes recovering from this type of transient failure might not return to the state intended by the experimenter, and more critically that content on individual Displays might not be in the intended visibility state.

By omitting functionality for transaction recovery we are able to simplify the implementation of processes. As a result, our processes do not have to provide any means for recording system state to logs, recovering from logs, or garbage-collecting log entries. The absence of a distributed

termination protocol also simplifies the implementation of processes in our software infrastructure. Moreover, our unilateral and pessimistic termination protocol completely avoids the blocking of processes (one of the problems of the classic two-phase commit protocol).

The drawback of these simplifications is that our commit protocol does not guarantee atomicity in failure cases 2 and 5. More specifically, atomicity is still guaranteed if the outcome of a transaction is to abort. However, processes experiencing these failures will not implement the intended changes to content visibility if the transaction is committed.

However, we feel that these restrictions are acceptable in the light of the specific properties of public display networks. Firstly, as far as the individual displays are concerned, future states of these displays do not depend on their previous states, i.e. the showings of content items on these displays are independent from the showings of content items on these displays in the past. In other words, whether a particular content item is (and can be) shown on a public display in the future does generally not depend on whether or not a particular piece of content is currently shown on that display. Unintended outcomes of transactions are therefore only transient and have no permanent effects on the public display network.

Moreover, the classic two-phase commit protocol only guarantees that all processes that are involved in a transaction will eventually adopt the intended outcome of that transaction. However, we argue that this guarantee is not sufficient in the context of public display systems. A transaction in the context of our software infrastructure typically defines changes to the visibility of a set of content items, and users of our scheduling API expect these changes to happen not only atomically, but also roughly at the same time (i.e. once the transaction is committed). While the use of the classic two-phase commit protocol would guarantee that all involved content items implemented the intended visibility changes in failure cases 2 and 5, it does not guarantee that these visibility changes are all carried out simultaneously. Instead, those processes that have timed out or recovered from errors implement these visibility changes at a later point in time than those process that have not experienced any failures. As a result, the visible outcome of the transaction of the displays does not conform to the intended outcome of the transaction, i.e. the faults are visible as faults to human observers.

5.4.3 Access to Process Attributes

Display processes, Application processes and Handler processes provide access to their attributes in the form of status events that the processes periodically disseminate using the event channel. The structure of these events is outlined in table 5.5. In addition to fields representing the entity-type-specific attributes and their values, all `STATUS` events possess `entity_type` and `entity_id` fields that allow receivers to identify the type and identity of the processes that emitted the events. Processes emit these status events about once every minute. Additional `STATUS` events are released immediately after changes to attributes have occurred. By keeping the frequency of `STATUS` events low we can ensure that this method for providing access to process attributes is capable of scaling up to a few hundred displays (requirement R1). By releasing additional events after state changes, processes are able to inform interested processes about these state changes in a timely manner, which would, for example, not be possible if access to process attributes was provided through polling.

By subscribing to and monitoring status events, tools can be constructed that monitor the state of the processes comprising the software infrastructure. Using content-based subscriptions these tools are able to, for example, subscribe to all status events that are emitted by the Display processes in the system to construct a comprehensive overview of the state of each public display at any point in time. Since internal state changes lead to the immediate release of events, state transitions in the public display can be recorded in their temporal ordering, allowing researchers to construct detailed audit trails for their experiments (see requirement R14).

Events may also be used by Interaction Devices and Context Sources to disseminate information. In this case gateway processes are used that interface with the underlying hardware entities, for example using proprietary APIs. The gateway processes relay the interaction and context information provided by these entities onto the event channel. Examples describing how this mechanism can be used to realise mobile phone interaction using SMS text messages and to detect user presence by scanning for Bluetooth mobile phones can be found in the description of the implementation in chapter 6.

Field Name	Type	Description/Value
`event_type`	string	”COMMIT”
`trans_id`	string	The Transaction Identifier of the transaction.

Table 5.3: Structure of a `COMMIT` event.

Field Name	Type	Description/Value
`event_type`	string	”ABORT”
`trans_id`	string	The Transaction Identifier of the transaction.
`reason`	string	The reason for aborting the transaction.

Table 5.4: Structure of an `ABORT` event.

Field Name	Type	Description/Value
`event_type`	string	”STATUS”
`entity_type`	string	Specifies whether the event was generated by a process corresponding to a Display, Application or Handler entity. Possible values are: `"TYPE_DISPLAY"`, `"TYPE_APPLICATION"`, `"TYPE_HANDLER"`
`entity_id`	string	Depending on the type of process that generated the event, this field contains the Display Identifier, Application Process Identifier or Handler Identifier of that process.
entity-specific attributes in the form of (name,value) pairs ...		

Table 5.5: Structure of `STATUS` events.

5.4.4 Display Process Protocol Engine

Figure 5.10 shows an overview of the protocol engine that is used by Display processes. The generic structure of the protocol engine, including the processing of Handler chains and the engineering of transactional semantics is common to all types of requests that are handled by Display processes.

If Handlers are present in the pre-Handler chain, then the incoming request event is handed over sequentially to each Handler in the pre-Handler chain using `HANDLE` requests. If the original request was part of a transaction, then the `HANDLE` requests that are emitted by the Display process are transactional as well. In this case the original request and the `HANDLE` request are both part of the same transaction. This means that actions taken by Handlers while processing these `HANDLE` requests will not compromise the isolation requirement, i.e. the effects of these actions will not become visible to other processes or human observers until the transaction is committed. Moreover, this means that each Handler acts as a full cohort in the transaction protocol, for example causing it to react autonomously once the transaction is committed or aborted without requiring any further intervention from the Display process.

If one of the Handlers returns a `RESULT` event whose `result` field is set to `"FAILED"`, the Display emits a negative response for the original request and stops processing the request. Additionally if the request was processed in a transactional context, the transaction is aborted and all prior actions that were taken in the context of the request are rolled back. Please note that for the sake of consistency with the `RESULT` messages emitted by the other processes in our engineering model, the `RESULT` events emitted by Handler processes use the `result` field to indicate whether processing was successful or not.

Once all the Handlers in the chain have been processed successfully, i.e. they have all returned positive `RESULT` events, the Display starts processing the actual request. The arguments of the request may have been modified in the meantime by the Handlers in the pre-Handler chain. The processing of a request may, for example, involve detecting conflicts with other Application processes, or requesting state changes from Application processes using `CHANGE_STATE` requests.

How exactly a request is further processed after the pre-Handler chain has been processed also depends on whether or not the request is part of a transaction. As we have outlined before, the processing of requests in a transactional context can be split into two phases:

- a preparation phase, during which the process checks whether the actions associated with the requested operation can be performed, and during which any actions are performed that will not be observable by other processes in the system or by human observers. If the Display process emits any requests to other processes in the system, these requests will be part of the current transaction, i.e. these requests will not compromise the isolation requirement. Moreover, the receiving processes will autonomously commit these sub-requests (i.e. without requiring any additional intervention from the Display process) once the transaction is committed. Similarly, if the transaction is aborted, the sub-requests will autonomously abort without requiring intervention from the Display process.

- a commit phase, during which those actions are carried out that could not be performed during the preparation phase since they would have violated the isolation requirement. Additionally, the process' internal state is adapted during this phase to reflect the changes caused by the processing of the request.

In a transactional context, actions associated with the preparation phase are carried out after the pre-Handlers have been processed, and before the post-Handlers are processed (this corresponds to the phase labelled "processing request" in figure 5.10). Actions associated with the commit phase are performed once a `COMMIT` event has been received for the transaction.

If a request is carried out in a non-transactional context, there is no distinction between the preparation phase and the commit phase, and the actions associated with both phases are performed as a single unit after the pre-Handlers have been processed, and before the post-Handlers are processed (see "processing request" in figure 5.10).

If the actions associated with the request cannot be carried out successfully, the Display process immediately generates a negative `RESULT` event and stops processing the request. If on the other hand the actions have been completed successfully, the Display process generates a positive `RESULT` event that also contains any additional request-specific results. However, this `RESULT` event is not emitted immediately, but is passed along as input to the Handlers in the post-Handler chain. Unlike pre-Handlers, post-Handlers are not able to cause further processing of the handler chain to be aborted. Post-Handlers are also unable to modify the overall outcome (success or failure) of a request, i.e. they may not modify the `result` field of its `RESULT` event. However, post-Handlers may modify the remaining fields of a `RESULT` event that they have received as an argument.

Once all post-Handlers have been processed successfully, the Display process emits the (possibly modified) `RESULT` event as a response to the original request.

Please note that after individual Handlers have processed a request successfully, these Handlers cannot be certain that the overall processing of the request is going to be successful as well. In our current engineering model, each Handler is therefore required to monitor the outcome of a request, e.g. by subscribing to `RESULT` and `STATUS` events (alternatively, all communication between Display processes and Handler processes could be engineered with transactional semantics, allowing Display processes to inform the associated Handler processes about the overall outcome of an operation by committing or aborting the corresponding transaction).

If a transaction is aborted, all requests that are in a prepared state have to be rolled back. This means that all actions that were already performed during the preparation phase have to be reverted. Moreover, unexpected faults may occur, for example if Handlers fail to respond. In such a case the Display process should attempt to bring the system back into a meaningful state. In the transactional case this is easier, since the processing of a request does not have any noticeable effects until the transaction is committed. In the non-transactional case, this might involve emitting additional protocol requests to revert actions that were previously taken.

We describe the request-specific routines for each request in the remainder of this section, along with the structure of the request and response events that are used to invoke each operation.

CreateApplication

Format of Protocol Request

Field Name	Type	Description/Value
`event_type`	string	"CREATE_APPLICATION"
...`request_id` and `trans_id` fields...		
`display_id`	string	The identifier of the Display the Application is to be instantiated on.
`content_url`	string	A URL referencing the content item that is to be rendered by the Application instance
`group_id`	string	(Optional) If this field is present, the new Application instance is to be added to the specified Application Group.

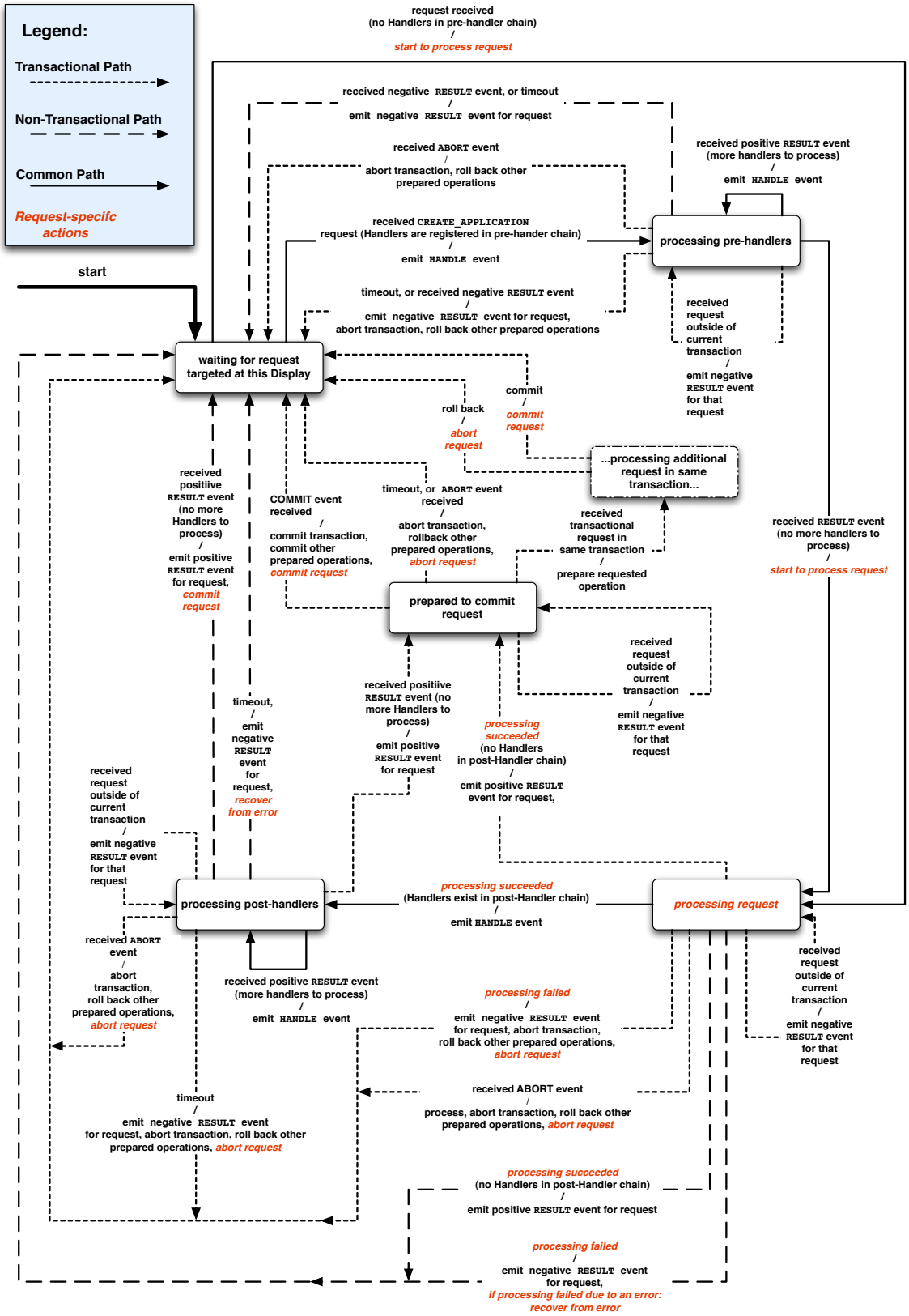


Figure 5.10: Protocol engine used by Display processes.

Format of Protocol Response

Field Name	Type	Description/Value
`event_type`	string	"RESULT"
...`request_id`, `result` and `reason` fields...		
`pid`	string	The Application Process Identifier of the newly created Application.

Processing the Request

Preparation Phase The request-specific actions that are performed by a Display process during the preparation phase in response to a `CREATE_APPLICATION` request are shown in figure 5.11. The Display process inspects the type of the content item that is to be rendered by the new Application instance (specified in the `content_url` field of the request) and selects an appropriate Application type that is capable of rendering the content item. The Display process then generates a new Application Process Identifier and subsequently creates a new Application process of the appropriate type. The new Application instance is initialised with the Application Process Identifier, the URL, the transaction identifier ("0" if the request is non-transactional) and optionally with an Application Group Identifier that indicates that the new Application instance is to be part of that Application Group. The Display process waits for a `STATUS` event from the newly created Application process that is emitted as soon as the Application process enters the Application state `IDLE`, i.e. it has successfully initialised itself. If such a `STATUS` event is received, the request-specific part of the preparation phase is considered to have ended successfully. In this case, the `RESULT` event generated by the Display process additionally contains a `pid` field that is used to communicate the Application's Application Process Identifier to the API instance and ultimately to the Scheduler (see also figure 5.7).

If the Display process fails to receive a `STATUS` event within a certain time-out period, the instantiation of the Application process is considered to have failed due to an error.

Commit Phase The Display process commits the request by adding the Application Identifier of the newly created Application instance to the Display's "running" attribute that provides information about the Application instances that are active on the Display. To disseminate the change in the Display's status, it immediately emits a `STATUS` event.

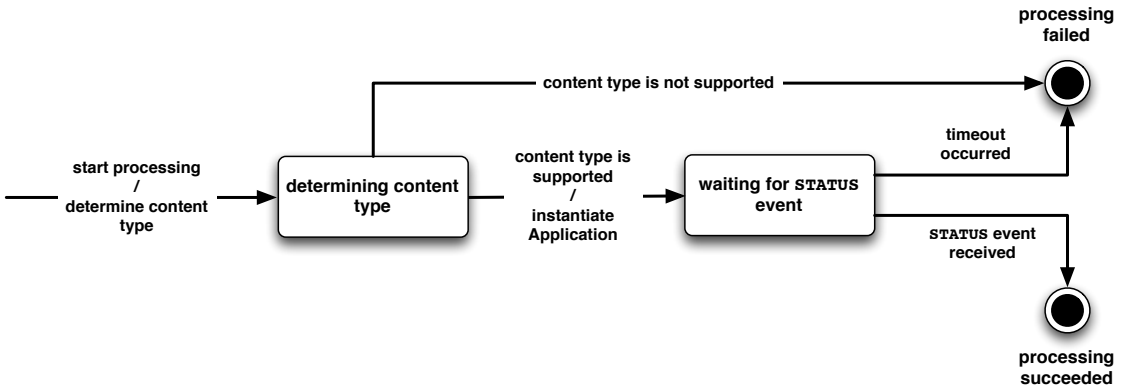


Figure 5.11: State machine representing the preparation phase of a CREATE_APPLICATION request.

Aborting the Request

The Display process is not required to take any actions to revert the request since the Application process was instantiated using transactional semantics. As a result, the Application process monitors the outcome of the transaction and terminates itself if the transaction is aborted.

Error Recovery

The Display process attempts to destroy the newly created Application instance by emitting a CHANGE_STATE request event with a target state of TERMINATED.

Transition

Format of Protocol Request

Field Name	Type	Description/Value
`event_type`	string	"TRANSITION"
...`request_id` and `trans_id` fields...		
`pid` or `group_id`	string	Application Process Identifier of the Application process that is to be transitioned or, if the request is to be carried out on a group of Applications, the Application Group Identifier of that group.
`target_state`	string	The targeted visibility state. This can be one of `"VISIBLE"` or `"NOT_VISIBLE"`.
`priority`	int32	This field is only present if the `target_state` field is set to `"VISIBLE"`. It specifies the priority of the Application process that is to be made visible. The priority value is used by Handlers and Displays for resolving conflicts between Applications.

Format of Protocol Response

Field Name	Type	Description/Value
`event_type`	string	"RESULT"
...`request_id`, `result` and `reason` fields...		
`display_id`	string	(optional) If the request was a group request, this field contains the Display Identifier of the Display process that emitted the response event.

Processing the Request

Preparation Phase The steps involved in processing a `TRANSITION` request are shown in figure 5.12. If the target state of the request is `VISIBLE`, the Display process first checks for visibility conflicts with other Applications on the same conceptual Display. The realisation of design proposition DP5 means that each Display can have at most one Application visible at the

same time. Therefore, if the content rendered by another Application process is already visible on the Display, the TRANSITION request can only succeed if the Display can be preempted from that Application. The Display process uses the priority-based conflict resolution algorithm presented in chapter 4 to determine whether an Application that is currently visible on the Display can be preempted. If this is possible, the Display process attempts to change the offending Application's state to NOT_VISIBLE by emitting a CHANGE_STATE request. If the original TRANSITION request was part of a transaction, then the emitted CHANGE_STATE request is transactional as well and part of the same transaction. If making the offending Application NOT_VISIBLE is successful, the Display process emits an additional CHANGE_STATE request, targeted at the Application that is to be made visible, to instruct that Application to change its visibility state to VISIBLE. This CHANGE_STATE request is again transactional, provided that the original TRANSITION request was part of a transaction as well. A time-out to receive RESULT events for any of these CHANGE_STATE request means that processing of the TRANSITION request is considered to have failed. Processing is also considered to have failed if the Display cannot be pre-empted from the offending Application, or if changing that Application's state to NOT_VISIBLE fails.

If the target state of the request is NOT_VISIBLE, the request has no potential to cause visibility conflicts. The conflict detection step is therefore omitted by the Display process. In this case, the Display process emits a CHANGE_STATE request to instruct the targeted Application to transition into Application state NOT_VISIBLE. This CHANGE_STATE request is emitted with transactional semantics, provided that the original TRANSITION request was part of a transaction.

Commit Phase The Display process records the targeted Application process as "visible", modifies its visible attribute accordingly, and emits a STATUS event.

Aborting the Request

The Display process is not required to take any actions since the CHANGE_STATE request that is used to instruct the Application to change its state is carried out in a transactional context as well. Moreover, this CHANGE_STATE request is part of the same transaction as the original TRANSITION request. Aborting the transaction therefore causes the involved Application process to autonomously revert any internal changes that were made as part of processing the CHANGE_STATE

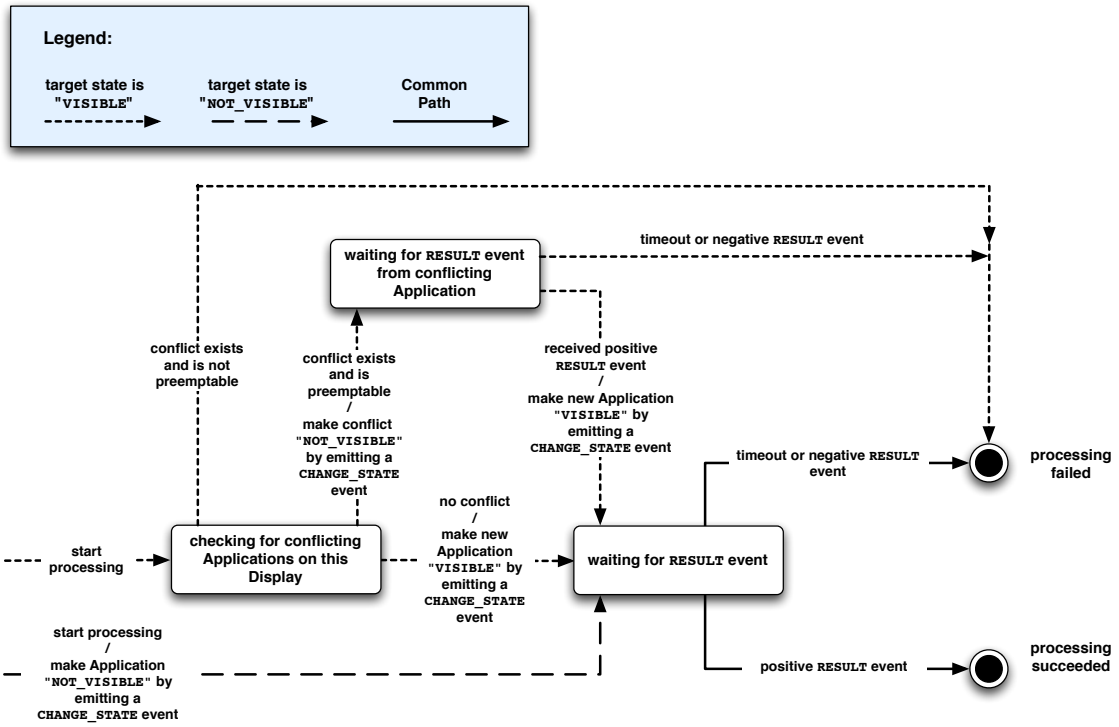


Figure 5.12: State machine representing the preparation phase of TRANSITION requests.

request.

Error Recovery

The Display process attempts to minimise the visible effects of the error by returning the Application to its original state using a CHANGE_STATE request.

TerminateApplication

Format of Protocol Request

Field Name	Type	Description/Value
`event_type`	string	"TERMINATE_APPLICATION"
...`request_id` and `trans_id` fields...		
`pid` or `group_id`	string	Application Process Identifier of the Application process that is to be terminated or, if the request is to be carried out on a group of Applications, the Application Group Identifier of that group.

Format of Protocol Response

Field Name	Type	Description/Value
`event_type`	string	"RESULT"
...`request_id`, `result` and `reason` fields...		
`display_id`	string	(optional) If the request was a group request, this field contains the Display Identifier of the Display process that emitted the response event.

Processing the Request

Preparation Phase The Display process emits a `CHANGE_STATE` event to request the Application process to enter the state `TERMINATED`, causing the Application process to terminate itself, and subsequently waits for a `RESULT` event from the Application process (see figure 5.13). If a positive response is received, processing is considered to have ended successfully. Otherwise the original request is answered with a negative `RESULT` event.

Commit Phase The Display process removes the Application process from the `running` attribute, and, if the Application process was previously visible on the Display, from the `visible` attribute. To disseminate the changes, the Display process emits a `STATUS` event.

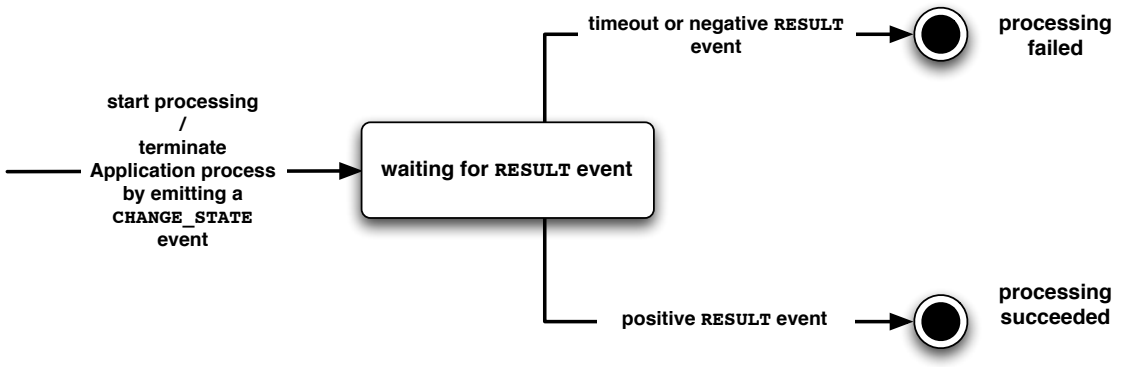


Figure 5.13: State machine outlining the protocol actions taken by Display processes for TERMINATE_APPLICATION requests.

Aborting the Request

The Display process is not required to take any actions to revert the request since the actual changes to the visibility of content are performed by Applications as a result of CHANGE_STATE requests. Since these requests are carried out in the same transactional context as the original TERMINATE_APPLICATION request, aborting the transaction will cause the involved Application processes to revert their internal changes.

Error Recovery

Since the effects of TERMINATE_APPLICATION requests cannot be reverted, the Display process does not take any further actions to recover from time-out errors, and leaves the handling of these faults to Schedulers.

5.4.5 Application Process Protocol Engine

An overview of the protocol engine implemented by Application processes (and Handler processes) is provided in figure 5.14. As in the case of Display processes, the transactional processing of requests is performed in two phases: a preparation phase and a commit phase. Similarly, if a request is processed in non-transactional context, there is no distinction between the preparation and commit phases, and the actions associated with both phases are carried out as a single unit as part of processing the request. For a more detailed discussion of the differences between the

preparation phase and the commit phase please refer to the description of the Display process protocol engine in section 5.4.4.

ChangeState

Format of Protocol Request

Field Name	Type	Description/Value
`event_type`	string	"CHANGE_STATE"
...`request_id` and `trans_id` fields...		
`pid` or `group_id`	string	Specifies which process(es) this request is addressed to. Can be an Application Process Identifier or an Application Group Identifier.
`target_state`	string	The Application state that the Application should transition to.

Format of Protocol Response

Field Name	Type	Description/Value
`event_type`	string	"RESULT"
...`request_id`, `result` and `reason` fields...		
`pid`	string	(Optional) If the request was directed at an Application Group, this field contains the identifier of the Application process that generated the response.

Processing the Request

Preparation Phase If the requested target state can be reached via a legal state transition, the Applications process reacts by calling the appropriate renderer function (`prepare`, `make_visible`, `make_not_visible` or `terminate`) that corresponds to the intended target state (`PREPARED`, `VISIBLE`, `NOT_VISIBLE` or `TERMINATED`). The calls are made with the commit flag cleared, i.e. any actions that influence the visibility of the Application process' content are held back until the request is committed.

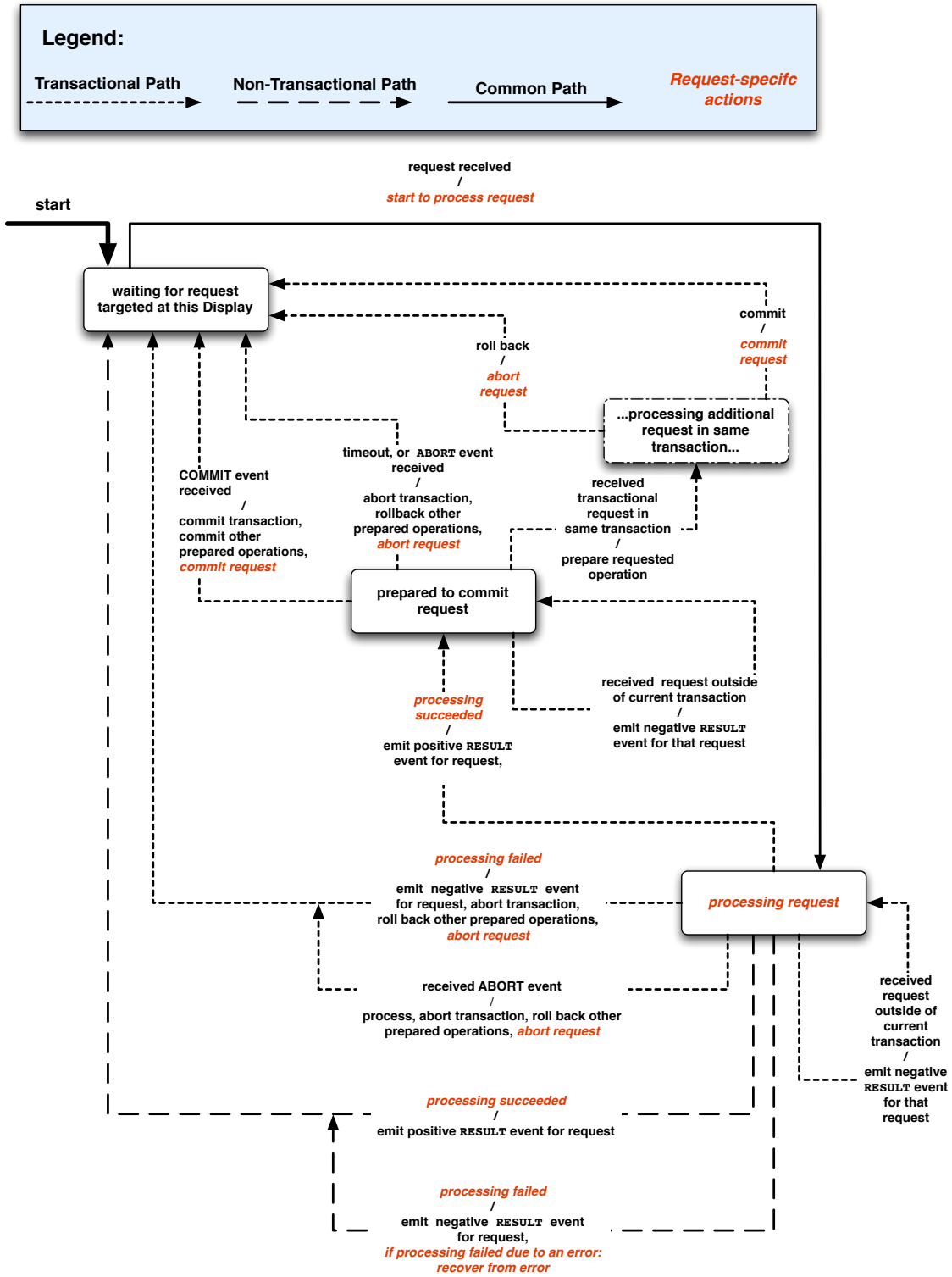


Figure 5.14: State machine outlining the protocol actions taken by Application and Handler processes.

Commit Phase The Application process initiates committing the request by invoking the renderer function that corresponds to the original request once again, this time with the commit flag set. If the request involved changes to the visibility of the Application's content, these changes will be made as part of this second invocation.

The Application process changes its externally visible attributes to reflect the process' new state, and disseminates this information using by emitting a `STATUS` event.

Aborting the Request

Since all calls to renderer functions are carried out in a transactional context, the Application process is not required to take any actions to revert request.

Error Recovery

If unexpected faults occur, the Application process attempts to minimise the visible effects of these faults by trying to make its content not visible and by stopping content playback.

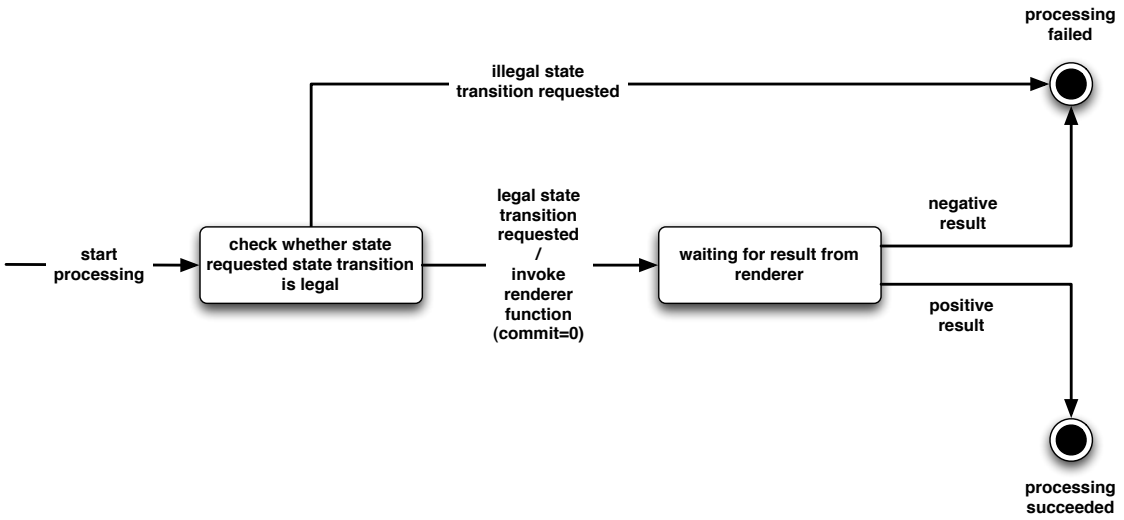


Figure 5.15: State machine representing the preparation phase of `CHANGE_STATE` requests.

5.4.6 Handler Process Protocol Engine

Each type of Handler serves a different purpose and will therefore perform different actions in response to a `HANDLE` request. However, similar to Display processes and Application processes, a common structure of protocol actions can be identified that Handlers use to process `HANDLE` requests in transactional or non-transactional contexts. An overview of the abstract structure of the protocol engine employed by Handler processes is shown in figure 5.14. As in the case of Display and Application processes, the transactional processing of requests is performed in two phases: a preparation phase and a commit phase. If requests are processed in a non-transactional context, there is no distinction between the preparation and commit phases, and the actions associated with both phases are carried out as a single unit as part of processing the request. For a more detailed discussion of the differences between the preparation phase and the commit phase please refer to the description of the Display process protocol engine in section 5.4.4.

Handle

Format of Protocol Request

Field Name	Type	Description/Value
`event_type`	string	”HANDLE”
...`request_id` and `trans_id` fields...		
`handler_id`	string	The Handler Identifier of the Handler that this event is addressed to.
`display_id`	string	The Display Identifier of the Display process that emitted the event.
`handler_chain`	string	The processing phase. Can be `"PRE_HANDLER_CHAIN"` or `"POST_HANDLER_CHAIN"`.
`original_request`	string	A marshalled version of the request event that the Handler should handle.
`original_result`	string	This field is only present in the post-Handler phase. It contains a marshalled version of the response event that the Display process is going to send to the originator of the original request.

Format of Protocol Response

Field Name	Type	Description/Value
`event_type`	string	"RESULT"
...`request_id`, `result` and `reason` fields...		
`modified_request`	string	This field is only present in the pre-Handler phase. It contains a marshalled request event that is to be used by the Display to further process the request. The request event is a potentially modified version of the original request event that was contained in the `original_request` field of the `HANDLE` request.
`modified_result`	string	This field is only present in the post-Handler phase. It contains a marshalled response event that is to be used by the Display to respond to the request. The response event is a potentially modified version of the response event that was contained in the `original_result` field of the `HANDLE` request.

Processing the Request

Preparation Phase Handlers are free to perform any forms of processing they desire during this phase, as long as processing does not exceed a certain maximum duration, causing the Display to consider the `HANDLE` request as having failed. For example a Handler that is responsible for switching on a physical display if content is brought into state `VISIBLE` typically subscribes to `TRANSITION` events. As reaction to an incoming `HANDLE` request carrying a `TRANSITION` event, the Handler might, for example, check the state of the display hardware and subsequently emit commands to that display hardware to cause it to power on.

Actions performed during this phase are required to observe the isolation requirement. This means that during the preparation phase only those actions may be carried out whose results are not visible to other processes or humans observing the displays.

As part of its preparation activities, the Handler may also modify the arguments of the request that is to be handled, or the response that is going to be returned by the Display process.

Commit Phase During this phase, Handlers will generally modify their internal state to match the actions taken during the "request-specific processing" phase. Moreover, Handlers will perform any actions that have been queued up during the preparation phase, such as actions modifying the visibility of Applications.

Aborting the Request

The actions that are used to revert the effects of `HANDLE` requests are specific to the type and purpose of each individual Handler.

Error Recovery

The actions that are used to recover from errors are specific to the type and purpose of each individual Handler.

5.5 Summary

In this chapter we have described the engineering of the software infrastructure. We started by discussing the options for mapping the computational entities presented in chapter 4 to processes, and for distributing these processes onto the machines in a public display network. We have described an engineering model for Application processes, Display processes, Handler processes, and Scheduler processes. Moreover, we have provided a detailed description of the protocol that is used to interconnect the processes in our engineering model. We have provided detailed descriptions of the engineering of transactions and operations on Application Groups. We have shown how the processes in our model use the properties of the event-channel to provide access to their attribute values, and how the publish/subscribe nature of the event channel can be used to observe the exchange of protocol messages, e.g. with the aim of creating audit trails or of monitoring system activity. Moreover, we have provided descriptions of the protocol engines used by the processes in our model to process the various protocol messages.

The next chapter will describe the implementation status of our software infrastructure and provide a detailed evaluation – both qualitative and quantitative – of the implemented system.

Chapter 6

Implementation and Evaluation

6.1 Introduction

In the previous chapter we described the engineering of low-level APIs, infrastructure and protocols, and the mapping of the computational entities described in chapter 4 to the actual hardware deployed as part of the e-Campus public display network. In this chapter we describe the implementation status of our software infrastructure and API and provide a detailed evaluation – both qualitative and quantitative – of the implemented system. The qualitative evaluation is founded on the experiences gained on a day-to-day basis from using the infrastructure and API as part of the e-Campus public display network, and specifically on a number of experimental applications and content that were supported using the infrastructure and API. Performance measurements and an analysis of the scalability of our software infrastructure provide quantitative data about our implementation. We analyse both qualitative and quantitative results with respect to the requirements outlined in chapter 3.

6.2 Implementation Status

6.2.1 Overview

We have created a reference implementation of the software infrastructure described in chapter 5. Display processes, Application processes and Handler processes have been implemented in Python [Pyt08] and use Elvin [SA97] notifications to communicate. The processes are hosted on Apple Mac Mini machines that are running the Mac OS X operating system [App08]. Python has also been used to implement the scheduling API that is accompanying our software infrastructure. Moreover, we have implemented examples for Context Sources and Interaction Devices. In the following sections we provide a description of these implementations.

6.2.2 The Low-Level Infrastructure

The software infrastructure described in chapter 5 has been implemented in the form of a set of Python processes that are executed on Mac Mini machines that are running the Mac OS X [App08] operating system. Python [Pyt08] is an interpreted programming language, for which interpreters are available for a range of different operating systems. While the software infrastructure has mainly been developed for and tested on OS X based systems, the use of Python enables us to port the developed code to other platforms with zero or minimal changes to the code base.

The processes in our software infrastructure communicate using Elvin notifications. Elvin [SA97] is an asynchronous notification infrastructure that uses a central server to multicast notifications consisting of sets of $(name, value)$ pairs to subscribers. Subscriptions are content-based and may, for example, be based on the presence of field names in notifications or on the values of individual fields. Clients register subscriptions with the central server. Publishers and subscribers maintain persistent TCP connections with the central server that is used to communicate both control information and data (notifications). The Elvin server is a commercial product, and language bindings for the Elvin API are available for Python, C, and Java.

To be able to support off-the-shelf content we have created Application types for videos and for web-based content. The latter Application type is capable of displaying flash animations and images, even if they are not embedded into web pages.

In our current version of the implementation, the functionality provided by Displays is for historic reasons mapped to two separate processes. However, we expect the functionality provided by these processes to be merged into a single Display process in the near future.

To hide the complexities of the underlying hardware from users of our software infrastructure, we have created specialised Handlers for each display type. These Handlers are responsible for powering displays on as soon as content is made visible on the respective Displays, and for powering displays off again once a display no longer has any content visible. Moreover, the Handlers ensure that the correct display inputs are selected. Commands are typically interchanged with LCD displays and projectors using model-specific RS-232 interfaces provided by the actual display hardware. The installation in the Underpass represents an exception to this rule: the RS-232 control interfaces exposed by the projectors are connected to an AMX NetLinx controller [AMX08]. The AMX controller exposes a user interface on a small tablet PC that can, for example, be used to manually power projectors on or off. Handlers in the Underpass are therefore required to communicate with the AMX controller using a TCP-based protocol.

As a result of using these handlers, powering displays on and off and selecting display inputs is completely transparent to users of our software infrastructure.

6.2.3 The Low-Level API

The scheduling API provided by our software infrastructure is exposed to developers of Schedulers in the form of a Python-based API. We would therefore normally expect Schedulers to be implemented in Python, although Python code can also be embedded and called from within C or C++ programs.

The Scheduler-facing part of the API is provided in the form of two classes: `api` and `transaction`. The `api` class exposes non-transactional versions of the operations provided by Display processes and Application processes, while the `transaction` class exposes access to the same operations with transactional semantics. Calls to the methods exposed by both classes are blocking, i.e. calling threads will be blocked until either a response is received from the underlying Display or Application processes that are serving these operations, or until a time-out occurs.

An overview of the methods exposed by the `api` and `transaction` classes is provided in figures

```
class api
    __init__( self, elvin_connection ):
        ...

    CreateApplication( self, display_id, url, group_id = None ):
        ...
        return ( result, pid )

    TerminateApplication( self, pid ):
        ...
        return result

    ChangeState( self, pid, target_state ):
        ...
        return result

    Transition( self, pid, visibility,
                priority = DEFAULT_PRIORITY ):
        ...
        return result
```

Figure 6.1: Non-transactional low-level API operations.

6.1 and 6.2.

A transaction is initiated by creating an object instance of the `transaction` class. All operations that are invoked on a transaction object belong to the same transactional block. Transactions are committed by invoking the `commit()` method. Transactions can be aborted explicitly by the Scheduler by calling `abort()`. If a transaction is aborted by the API instance, e.g. due to the failure of one of the operations within the transactional context, the transaction object throws an exception that should be caught by the calling thread.

Examples

Figure 6.3 shows an example of a Scheduler using the operations exposed by the `api` class to make a piece of content visible on a single Display ("display-01"). The constructor of the `api` class expects an initialised Elvin connection object as argument. Having created an API instance, the Scheduler instructs the Display to instantiate an Application process capable of rendering the content identified by the URL "http://xyz.co.uk/content.html". The `ChangeState` operation returns a result code and, if successful, the Application Process Identifier (`pid`) of the newly created Application instance. This Application Process Identifier is used in subsequent API calls to identify the target Application instance for the calls. The Scheduler requests the content to

```
class transaction
    __init__( self, api_instance )

    CreateApplication( self, display_id, url, group_id = None )
        returns ( result, pid )

    TerminateApplication( self, pid )
        returns result

    ChangeState( self, pid, target_state )
        returns result

    Transition( self, pid, visibility,
                priority = DEFAULT_PRIORITY )
        returns result

    commit( self )

    abort( self )
```

Figure 6.2: Transactional low-level API operations.

be made visible, sleeps for 60 seconds, requests the content to be hidden and finally instructs the Display process to terminate the Application.

```
...
a = api( connection )
( worked, pid ) = a.CreateApplication( "display-01",
                                       "http://xyz.co.uk/content.html" )
a.ChangeState( pid, APPLICATION_STATE_PREPARED )
a.Transition( pid, APPLICATION_STATE_VISIBLE )
time.sleep( 60 )
a.Transition( pid, APPLICATION_STATE_NOT_VISIBLE )
a.TerminateApplication( pid )
...
```

Figure 6.3: Example of a Scheduler using non-transactional API operations.

Figure 6.4 shows an example of a Scheduler creating and displaying two pieces of content on two public displays using transactional semantics. The calls to `CreateApplication` cause two Application instances to be created on Displays “display-01” and “display-02”. The Application instances are added to an Application Group. The Scheduler then instructs the Application instances to pre-load their content and make that content visible using `ChangeState` and `Transition` operations on the Application Group.

```
...
a = api( connection )
group = ApplicationGroup()
t = transaction( a )
try:
    ( worked, pid1 ) = t.CreateApplication( "display-01",
                                            "file:///content.html",
                                            group )
    ( worked, pid2 ) = t.CreateApplication( "display-02",
                                            "file:///video.mpeg",
                                            group )
    t.ChangeState( group, APPLICATION_STATE_PREPARED )
    t.Transition( group, APPLICATION_STATE_VISIBLE )
    t.commit()
except:
    # Transaction was aborted
    pass
...
```

Figure 6.4: Example of a Scheduler invoking API operations with transactional semantics.

6.2.4 Context Sources and Interaction Devices

We have so far implemented one example of a Context Source (Bluetooth scanners providing presence information) and one example of an Interaction Device (SMS-based interaction using mobile phones).

As we have described in chapter 4, Schedulers and Applications may choose to interface with Context Sources and Interaction Devices through out-of-band means. However, in our software infrastructure we also support the distribution of context information using Elvin notifications. As a result Schedulers and Applications are able to use Elvin's rich subscription language, for example, to selectively subscribe to specific types of context or interaction information, or to subscribe to context updates or interaction events carrying specific values. Both our implemented examples of Context Sources and Interaction Devices use Elvin events to distribute information to Schedulers and Applications.

Bluetooth Presence Information

In our implementation of the software infrastructure we have deployed Bluetooth scanners as Context Sources on each public display. The scanners are processes that search the vicinity of the displays for Bluetooth devices roughly once every 30 seconds and are targeted at discovering

Bluetooth-enabled mobile phones that are carried by users. Every Bluetooth device is equipped with a unique hardware address, and this address is retrievable during the device discovery process. As a result, the hardware addresses of Bluetooth-enabled mobile phones can be used to identify individual users where they are in the vicinity of public displays. Using information about the presence of Bluetooth devices in the surroundings of public displays therefore, for example, provides researchers with the ability to personalise content to individual users. In the past researchers have, for example, investigated solutions that use Bluetooth scans for personalising adverts that are shown on public displays [KHS+08, KPD07].

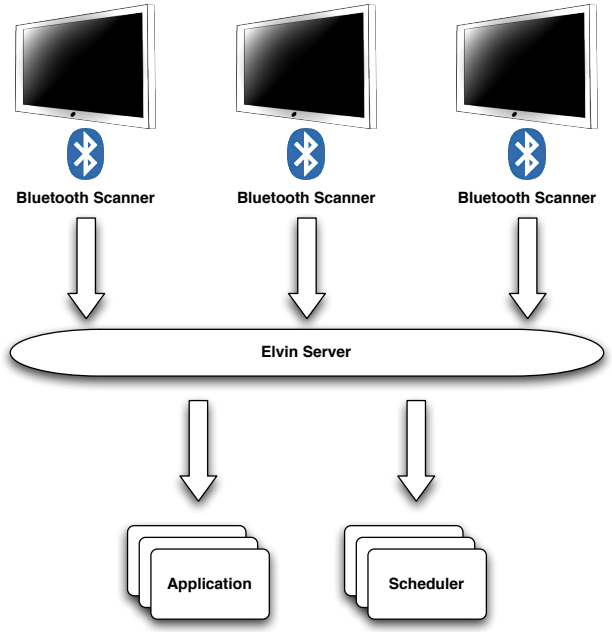


Figure 6.5: Dissemination of Bluetooth-based presence information.

Information about the discovered devices is disseminated using Elvin events that can, for example, be received by Schedulers or Applications (figure 6.5). This information includes, for example, the unique Display identifier of the Display the scanner is associated with, the number of devices that were discovered in the vicinity of the Display, the hardware addresses of the discovered devices (alternatively the scanners can be configured to return non-reversible hash codes of the devices' hardware addresses), and the order that these devices were discovered in.

We have successfully used a predecessor of the Bluetooth presence infrastructure to support an experimental capture-the-flag-style alternate reality game that used different displays of the e-Campus system as home bases (see section 6.6.3). The Applications involved employed Bluetooth presence events to detect and identify players in the vicinity of displays. We use the current version

of the Bluetooth presence infrastructure, for example, to provide researchers using the high-level scheduling API that we will present in chapter 7 with the ability to schedule content based on the presence of users.

6.2.5 SMS-based Interaction

In our implementation of the low-level software infrastructure we provide support for interaction based on SMS (Short Message Service) text messages. Using this infrastructure, researchers are able to construct content that users are able to interact with using text messages sent via their mobile phones. The text messages are required to comply to a specific format: '`<app_code> <display_number> <optional_input>`'. `app_code` and `display_number` identify the application and the public display that this interaction message is targeted at. If the actual input (`optional_input`) is omitted, then the selected content should simply be made visible on the targeted display. If input is specified, then in addition to making the content visible, the input should also be directed to that content.

```
event_type:  "remote_input"
display_id:  "ecampus-800"
application: "Map"
input_data:  "023"
remote_src:  "001122334455"
```

Figure 6.6: An SMS-based interaction event corresponding to the text message 'Map 800 023'.

A GSM gateway is used to receive these text messages. Software on the gateway is responsible for parsing the messages and for distributing them in the form of Elvin notifications in a generic format, enabling them to be received by the targeted Schedulers and Applications. An example of such an Elvin notification is depicted in figure 6.6. The depicted notification corresponds to a text message that was sent from a mobile phone with the phone number '001122334455' and that had the message body 'Map 800 023'. In this case the input is targeted at the Map application, one of the experimental interactive applications that we have constructed on top of our infrastructure for SMS-based interaction. The Map application, which is described in detail in section 6.6.1, allows users to request an interactive map of the campus of Lancaster University to be displayed on one of the displays, and to have a target location of their choice highlighted on the map. In case of the event in figure 6.6, the user has requested the map to be displayed on display 'ecampus-800'. Additionally, the location of the 'Bowland North' lecture theatre (corresponding to input

'023') should be highlighted on the map. The Map application consists of an Application and a dedicated Scheduler that subscribes to Elvin notifications that carry an `event_type` field of value 'remote_input' and whose `appplication` field has the value 'Map'. Once the Application has been made visible by the Scheduler, it communicates information about the intended input using out-of-band means. An overview of this process is provided in figure 6.7.

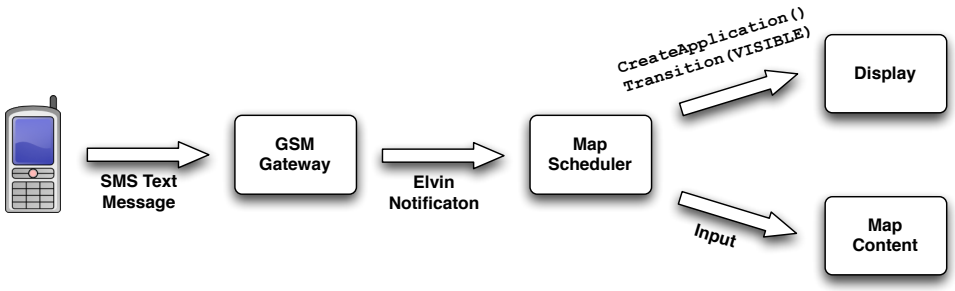


Figure 6.7: SMS-based interaction with the Map application.

Alternatively, researchers are able to implement their own parsing and handling software that is deployed on the gateway side-by-side with the standard parsing and handling software described above. This enables researchers, for example, to employ text message formats that do not correspond to the generic format described above, or to directly process messages on receipt without having to disseminate them over the Elvin event channel first.

6.3 Deployments

So far the software infrastructure described has been deployed on a total of 13 e-Campus displays. Initial deployments took place on one 40 inch LCD display and a projected display that were installed in the foyer of Lancaster University's campus theatre. A deployment of the software infrastructure on a set of three 40 inch LCD displays that were installed side-by-side outside one of the University's main lecture theatres followed. Subsequent deployments took place in the Underpass (featuring three projected displays side-by-side), on one LCD display in the foyer of InfoLab21 (the home of the Computing Department), one LCD display in the foyer of one of the colleges, and two LCD displays in the foyer of the Engineering Department. The software infrastructure is also active on one additional display that is installed in our lab for development and debugging purposes.

Within the next few months we will deploy the software infrastructure on a further 8 LCD displays, after which we plan to create a GNU/Linux port of the infrastructure that will enable us to deploy the software infrastructure on the electronic door displays that are installed in the Computing Department, and on the multi-headed PC in the Underpass.

Most of our deployment locations feature network cameras pointing at the displays, enabling us to monitor the output on the displays as it is perceived by our users [SFD+06b].

Moreover, all regular e-Campus displays (excluding the electronic door displays) are by default equipped with an off-the-shelf digital signage solution that enables standard content, such as videos and images, to be scheduled using a web-based scheduling interface that is based on constructing timelines for each display. The software infrastructure presented in this thesis currently co-exists on the individual display machines with this digital signage solution and is able to preempt the displays whenever required.

Day-to-day content was initially mainly scheduled using the digital signage solution, while we used our software infrastructure predominantly for showing experimental content that could not be scheduled using the digital signage software. This included, for example, interactive content that had to be displayed on demand, and content that was scheduled based on context events, such as the presence of certain Bluetooth devices within the vicinity of displays.

However, we have recently started to phase out the digital signage software and have begun to use our software infrastructure (in combination with the extensions that will be presented in chapter 7) to schedule not only experimental interactive and context-sensitive content, but also non-experimental, day-to-day content that was previously handled by the digital signage software.

6.4 Performance Overview

In this section we present an overview of the typical processing and communication delays that users of our low-level scheduling API will encounter when attempting to use the API to show content on public displays.

6.4.1 Methodology

To obtain performance measurements we instrumented the Display, Application and Handler processes on a total of three machines. The machines selected were all part of an installation that comprised three public displays mounted side-by-side outside one of our lecture theatres on campus. The machines were Apple Mac Mini computers running Mac OS X 10.4.11. Each machine was equipped with 1.83 GHz Intel Core Duo processors and 1 GB of 677 MHz DDR2 SDRAM memory.

To obtain measurements for different scenarios we created a small set of Schedulers (one per scenario) that were based on our scheduling API. The measurements presented for each scenario are based on 100 repetitions of each scenario. Where a scenario involved multiple runs with different numbers of displays, each run comprised 100 repetitions.

The Schedulers were hosted on a separate Apple Mac Mini machine that was located in the author's office, which was roughly 600 meters away from the deployment location of the displays. The machine was running Mac OS X 10.4.11 and was equipped with a 2GHz Intel Core 2 Duo processor and 1 GB of 677 MHz DDR2 SDRAM memory. The API on the machine was instrumented to allow us to obtain measurements of processing delays occurring in the API. The Elvin server was hosted on a separate Dell Optiplex GX620 server that was located in the author's office. The machine was running a GNU/Linux variant and was equipped with a 3 GHz Intel Pentium D 930 processor and 1 GB of 533 MHz DDR2 SDRAM memory. All machines were interconnected using Lancaster University's campus network.

Measurements were based on timestamps obtained by invoking Python's `time()` function (located in module `time`). The function `time()` returns a floating point number representing the system time in seconds since the epoch. On the machines used during the measurements, `time()` reported timestamps with a resolution of 1 microsecond. Immediately after the timestamps were taken, they were written to a log file on the computer's hard disk. A separate log file was used for each individual process.

The content used during the experiments was distributed to each computer in advance and subsequently retrieved by Application processes from the computers' hard disks. To minimise the effects that hard disk performance may have on the results (especially when comparing values

on different machines) we used a minimal MPEG-1 video as content. The video consisted of approximately 10 frames, a duration of less than 1 second and a file size of 224 KB. Moreover, the renderer was configured not to use any additional transition effects when making content visible, i.e. content windows were simply "deminiaturized" from the Mac OS X task-bar (called "Dock").

Where the term "average" is used without further qualification in the following sections, it refers to the mean average. Where error bars are shown, these represent $\pm 2 \cdot stderr$ around the sample mean. Similarly, mean values are quoted as sample means together with $\pm 2 \cdot stderr$. Moreover, all figures presented were rounded to the nearest millisecond after the calculations had been performed.

6.4.2 Experiment 1: Non-Transactional Operations on a Single Display

This experiment was designed to provide us with base-line figures about the processing delays encountered when using our API to show content on a single public display. The following code-snippet illustrates the structure of the Scheduler that was used during the experiment:

```
...
for i in range( 0, 100 ):
    # measurement point 1
    (worked, pid ) = api.CreateApplication( 'display-01',
                                            'file:/minimal_video.mpeg' )
    # measurement point 2
    api.ChangeState( pid, APPLICATION_STATE_PREPARED )
    # measurement point 3
    api.Transition( pid, APPLICATION_STATE_VISIBLE )
    # measurement point 4

    # followed by operations to remove the content from the display
    api.Transition( pid, APPLICATION_STATE_NOT_VISIBLE )
    api.TerminateApplication( pid )
...
```

All operations were performed without transactional semantics and without group semantics. Measurements were taken in the Scheduler in four different locations, i.e. before and after each operation invocation. The total processing and communication delay of an operation from the Scheduler's point of view was calculated as the difference between the timestamp taken immediately before the invocation of that operation and the timestamp obtained right after the call had returned. Moreover, the processing delays caused by the API instance and the involved Dis-

play, Handler and Application processes were measured by recording timestamps whenever the thread of activity associated with the processing of an operation entered or left a process. Figure 6.8 illustrates such a sequence of activities in the various processes using the example of a `CreateApplication()` request. If a thread of activity entered a process multiple times during the processing of an operation, these individual delays (represented by the black bars in the figure) were added to obtain the overall processing delay caused by the process. The communication delay was calculated as the difference between the sum of the processing delays incurred by Display, Handler and Application processes and the difference between the point in time when the request event was emitted by the API instance and the point in time when the API instance received the `RESULT` event.

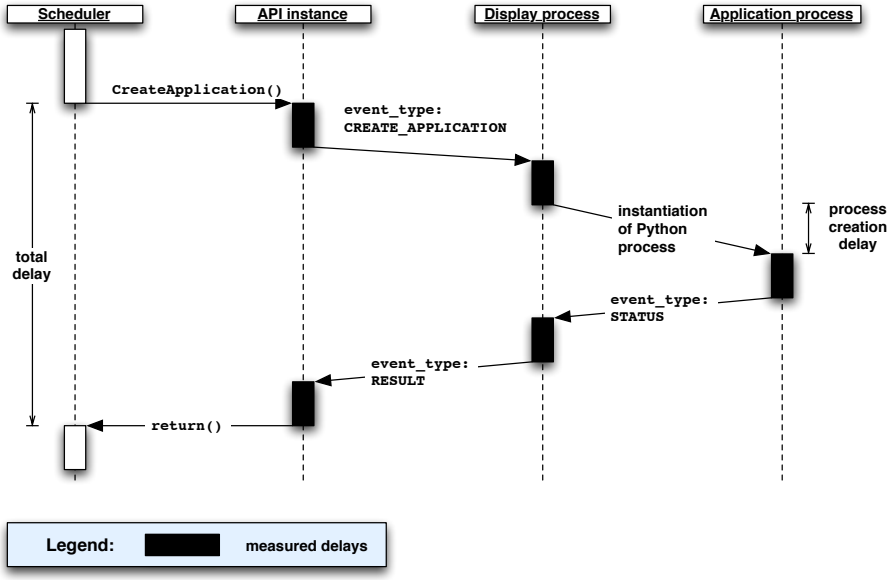


Figure 6.8: Activity of processes as result of a `CreateApplication()` invocation.

Table 6.1 provides an overview of the total delays experienced by the Scheduler in the context of the various operations. The mean total durations were: 2.681 ± 0.033 seconds for `CreateApplication()`, 0.057 ± 0.005 seconds for `ChangeState(PREPARED)` and 0.500 ± 0.012 seconds for `Transition(VISIBLE)`.

As the visual breakdown in figure 6.9(a) shows, the delay incurred when invoking `CreateApplication()` is mainly determined by two factors:

- the instantiation of the Python interpreter (labelled "process creation" in figure 6.9(a)). Since Application processes are implemented in Python, instantiating a new Application process

(a) `CreateApplication()`

	mean	SD
API instance	0.022 ± 0.003 s	0.014 s
Display process	0.141 ± 0.002 s	0.010 s
Application process	1.504 ± 0.029 s	0.146 s
communication	0.023 ± 0.002 s	0.009 s
process creation	0.990 ± 0.014 s	0.072 s
total	2.681 ± 0.033 s	0.163 s

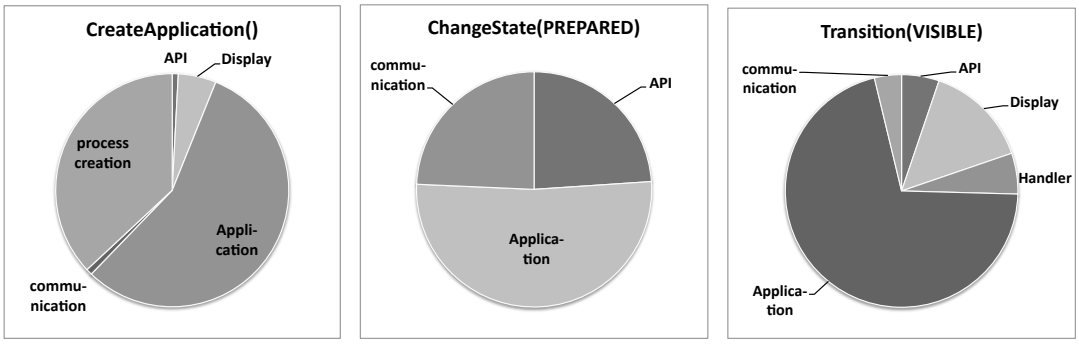
(b) `ChangeState(PREPARED)`

	mean	SD
API instance	0.014 ± 0.002 s	0.009 s
Application process	0.030 ± 0.004 s	0.018 s
communication	0.014 ± 0.000 s	0.000 s
total	0.057 ± 0.005 s	0.026 s

(c) `Transition(VISIBLE)`

	mean	SD
API instance	0.026 ± 0.003 s	0.016 s
Display process	0.073 ± 0.003 s	0.017 s
Handler process	0.028 ± 0.003 s	0.014 s
Application process	0.354 ± 0.011 s	0.053 s
communication	0.019 ± 0.002 s	0.010 s
total	0.500 ± 0.012 s	0.059 s

Table 6.1: Detailed overview of mean averages and standard deviations of communication and processing delays in the case of non-transactional operations performed on a single display.



(a) CreateApplication() (b) ChangeState(PREPARED) (c) Transition(VISIBLE)

Figure 6.9: Visual breakdown of the average communication and processing delays encountered for non-transactional operations performed on a single display.

involves creating a new instance of the Python interpreter. On average 0.990 ± 0.014 seconds of the total of 2.681 ± 0.033 seconds that are spent processing the `CreateApplication()` invocation are lost to the instantiation of this new Python interpreter instance.

- internal processing in the Application process (labelled "Application" in figure 6.9(a)). This processing time includes accessing the content to determine its MIME type and the instantiation of an appropriate renderer. Moreover, in our current implementation renderers are directly initialised with the content, i.e. instead of pre-loading content as result of a `ChangeState(PREPARED)` invocation, renderers already pre-load content as part of the processing of `CreateApplication()` invocations.

As we can see in figure 6.9(b), more than half of the delay experienced when invoking `ChangeState(PREPARED)` is caused by processing inside the Application process, with the other half being almost equally divided between processing in the API and communication delays.

The total delay in the context of `Transition(VISIBLE)` is mainly dominated by processing by the Application process. Of the 0.354 ± 0.011 seconds that were on average spent by Application processes to handle the corresponding `CHANGE_STATE` request, on average 0.327 ± 0.010 seconds were required to de-miniaturise the content window and to start playback of the video.

6.4.3 Experiment 2: Non-Transactional Group Operations on up to Three Displays

This experiment was conducted to provide an overview of the impact that operations on Application Groups have on the experienced processing and communication delays. The Scheduler used for conducting this experiment had the following overall structure:

```
...
all_displays = [ 'display-01', 'display-02', 'display-03' ]
displays = displays[ : number_of_displays ]
for i in range( 0, 100 ):
    permutated_list_of_displays = permutate( displays, i )
    group = ApplicationGroup()
    for disp in permutated_list_of_displays:
        # measurement point 1
        (worked, pid ) = api.CreateApplication( 'display-01',
                                                'file:/minimal_video.mpeg',
                                                 group )
        # measurement point 2
    # measurement point 3
    api.ChangeState( group, APPLICATION_STATE_PREPARED )
    # measurement point 4
    api.Transition( group, APPLICATION_STATE_VISIBLE )
    # measurement point 5

    # followed by operations to remove the content from the display
    api.Transition( group, APPLICATION_STATE_NOT_VISIBLE )
    api.TerminateApplication( group )
...
```

A total of three runs were conducted using a different number of displays (`number_of_displays`) in each run. As in the case of experiment 1, each run consisted of 100 iterations. During each iteration, we permutated the order in which Application processes were created on the different public

displays in order to minimise any effects that this order might have on the measured delays. A separate Application Group was created for each iteration, and all Application processes that were instantiated within an iteration were added to that group. The operations `ChangeState(PREPARED)` and `Transition(VISIBLE)` (and the clean-up operations) were carried out using group semantics. Apart from the measurement points in the Scheduler, the instrumentation of the API instance and the Display, Handler and Application processes remained unchanged when compared to experiment 1.

As can be seen in figure 6.10, the activity of processing an operation involves multiple parallel threads of activity as the group members process the resulting request in parallel. Moreover, additional processing is performed by the API instance to aggregate the `RESULT` events from the various group members.

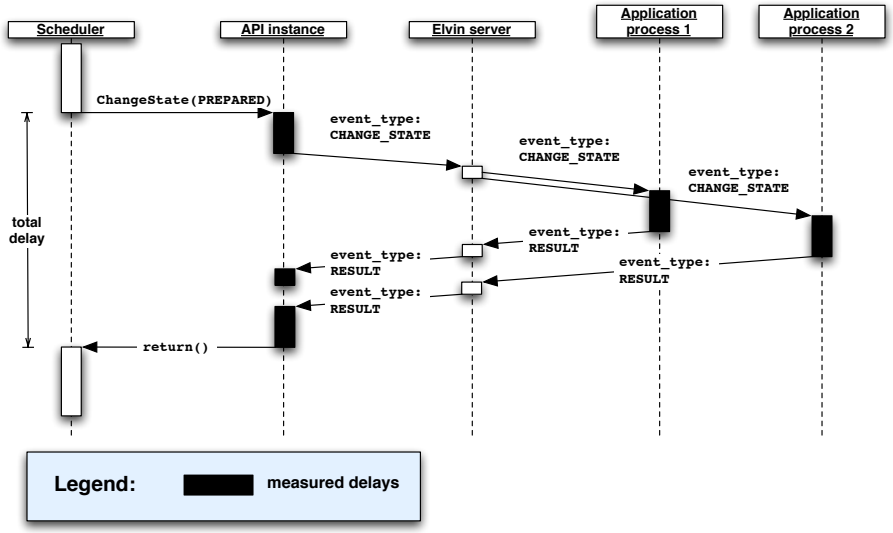


Figure 6.10: Activity of processes as result of a `ChangeState(PREPARED)` invocation with group semantics.

Table 6.2 lists the average measured delays for the `ChangeState(PREPARED)` and `Transition(VISIBLE)` operations for different numbers of displays. A graphical breakdown of these delays is presented in figure 6.11.

As we expected there are no significant impacts of the use of group operations on the processing delays incurred by Application, Display and Handler processes compared to the results of experiment 1. `HANDLE` requests that are emitted by Displays are individually addressed to each Handler. Handlers are therefore not impacted by the use of Application Groups. Application processes do

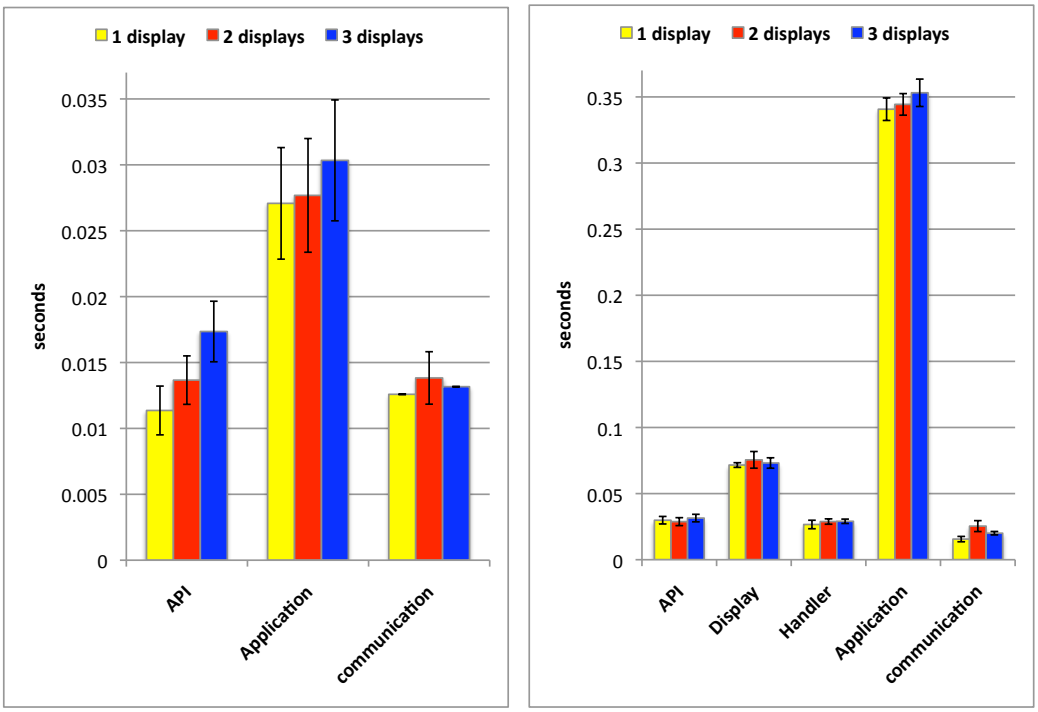
(a) `ChangeState(PREPARED)`

	1 display	2 displays	3 displays
API instance	0.011 ± 0.002 s	0.014 ± 0.002 s	0.017 ± 0.002 s
Application process	0.027 ± 0.004 s	0.028 ± 0.004 s	0.030 ± 0.005 s
communication	0.013 ± 0.000 s	0.014 ± 0.002 s	0.013 ± 0.000 s
experienced by Scheduler	0.051 ± 0.005 s	0.066 ± 0.010 s	0.078 ± 0.014 s

(b) `Transition(VISIBLE)`

	1 display	2 displays	3 displays
API instance	0.030 ± 0.003 s	0.029 ± 0.003 s	0.031 ± 0.003 s
Display process	0.072 ± 0.002 s	0.076 ± 0.006 s	0.073 ± 0.004 s
Handler process	0.027 ± 0.003 s	0.029 ± 0.002 s	0.029 ± 0.002 s
Application process	0.341 ± 0.009 s	0.344 ± 0.008 s	0.353 ± 0.010 s
communication	0.016 ± 0.002 s	0.025 ± 0.004 s	0.020 ± 0.001 s
experienced by Scheduler	0.484 ± 0.009 s	0.535 ± 0.024 s	0.577 ± 0.027 s

Table 6.2: Detailed overview of mean averages of communication and processing delays in the case of non-transactional group operations performed on one, two and three displays.



(a) ChangeState(PREPARED) (b) Transition(VISIBLE)

Figure 6.11: Graphical breakdown of the mean processing and communication delays in the case of non-transactional operations performed on one, two and three displays using group operations.

not internally distinguish between `CHANGE_STATE` protocol requests that are addressed at Application Groups and requests without group semantics. The use of group semantics therefore has no noticeable impact on the processing delays caused by Application processes. Display processes may incur additional processing delays if an Application Group contains multiple Applications that are hosted on the same Display. In this case the respective Display processes will have to spend additional time aggregating the results from the involved Application processes. However, in our scenario each Application Group consists of exactly one Application process per conceptual Display, and as a result no additional delays are incurred by the Display processes as result of the use of group operations.

Group operations do, however, have an effect on the API instance, which is tasked with aggregating the results received from the various processes. However, as we can see in figure 6.11 the overhead introduced by the addition of additional Application processes to an Application Group does not exceed a few milliseconds.

If a request is emitted with group semantics, the involved processes on the different conceptual displays process the request in parallel. We therefore measured the communication delays on a per-display basis. Let $p_{o,i,d}$ be the sum of the individual processing times of all the processes involved in the processing of operation o on Display d during iteration i. Moreover, let $\delta_{o,d,i}$ be the time difference between the point in time during iteration i at which the API emitted the request event corresponding to operation o and the point in time at which the API received the response from the targeted process on Display d. The mean communication delay for operation o over all iterations of an experiment run was calculated as follows:

$$\frac{1}{100N_d}\sum_{i=1}^{100}\sum_{d=1}^{N_d}\delta_{o,d,i} - p_{o,i,d} \tag{6.1}$$

where N_d described the total number of displays used during this particular run of the experiment. As we can see in figures 6.11(a) and 6.11(b) the addition of one or two conceptual Displays did not have any significant impact on the communication delays that we observed.

As requests are processed in parallel, the API instance is tasked with aggregating and evaluating the results received from the targeted processes in the Application Group. Specifically, the API operation does not return control to the Scheduler until one of the following conditions holds true:

a negative result has been received from one of the targeted processes, or positive results from all targeted processes have been received, or the maximum time the API instance is prepared to wait for results has been exceeded. For the API instance (and therefore also for the Scheduler), the duration of each API invocation in the absence of errors is therefore determined by that member process of the Application Group that is the slowest to respond. It is this effect that significantly contributes to the increases in the processing delays experienced by the Scheduler for `ChangeState(PREPARED)` and `Transition(VISIBLE)` as the number of displays increases (see figure 6.11).

Finally, the additional overhead of adding each newly created Application to an Application Group did not add any significant overhead to the average processing time of `CreateApplication()` invocations. In the run that was carried out using three displays, the average processing delays in the API instance for `CreateApplication()` invocations was recorded as 0.024 ± 0.002 seconds, compared to 0.022 ± 0.003 seconds in the case of experiment 1.

6.4.4 Experiment 3: Transactional Group Operations on up to Three Displays

A third experiment was conducted to provide an overview of the processing delays introduced by transactional semantics. The Scheduler used for experiment 2 was modified by replacing all non-transactional API operations with their transactional counter-parts. The sequence of creating Application instances on the different conceptual Displays, changing their state to `PREPARED` and making them visible was bundled into a single transaction:

```
...
all_displays = [ 'display-01', 'display-02', 'display-03' ]
displays = displays[ : number_of_displays ]
for i in range( 0, 100 ):
    permutated_list_of_displays = permutate( displays, i )
    group = ApplicationGroup()
    trans = transaction( api )
    for disp in permutated_list_of_displays:
        # measurement point 1
        (worked, pid ) = trans.CreateApplication( 'display-01',
                                                  'file:/minimal_video.mpeg',
                                                   group )
        # measurement point 2
    # measurement point 3
    trans.ChangeState( group, APPLICATION_STATE_PREPARED )
```

```
    # measurement point 4
    trans.Transition( group, APPLICATION_STATE_VISIBLE )
    # measurement point 5
    trans.commit()

    # followed by operations to remove the content from the display
    ...
...
```

(a) `CreateApplication()`

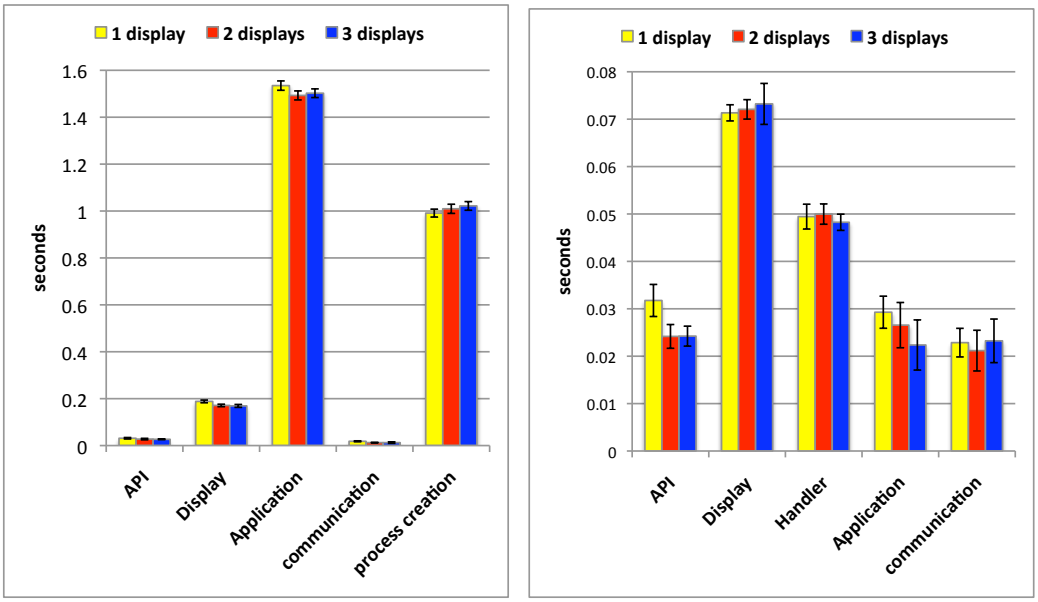
	1 display	2 displays	3 displays
API instance	0.031 ± 0.003 s	0.028 ± 0.003 s	0.027 ± 0.002 s
Display process	0.188 ± 0.006 s	0.171 ± 0.005 s	0.169 ± 0.006 s
Application process	1.535 ± 0.020 s	1.493 ± 0.019 s	1.502 ± 0.018 s
communication	0.018 ± 0.002 s	0.013 ± 0.001 s	0.013 ± 0.003 s
process creation	0.991 ± 0.017 s	1.009 ± 0.020 s	1.022 ± 0.018 s
experienced by Scheduler	2.764 ± 0.030 s	2.714 ± 0.040 s	2.732 ± 0.046 s

(b) `Transition(VISIBLE)`

	1 display	2 displays	3 displays
API instance	0.032 ± 0.003 s	0.024 ± 0.003 s	0.024 ± 0.002 s
Display process	0.071 ± 0.002 s	0.072 ± 0.002 s	0.073 ± 0.004 s
Handler process	0.049 ± 0.003 s	0.050 ± 0.002 s	0.048 ± 0.002 s
Application process	0.029 ± 0.003 s	0.027 ± 0.005 s	0.022 ± 0.005 s
communication	0.023 ± 0.003 s	0.021 ± 0.004 s	0.023 ± 0.005 s
experienced by Scheduler	0.205 ± 0.006 s	0.209 ± 0.015 s	0.216 ± 0.023 s

Table 6.3: Detailed overview of mean averages of communication and processing delays in the case of transactional group operations performed on one, two and three displays.

Figure 6.3 shows the resulting mean processing and communication delays for `CreateApplcation()` and `Transition()` operations. A graphical breakdown is shown in figure 6.12. In the case of `CreateApplication()` we witnessed an increase of the mean processing times by Display processes of around 46 milliseconds (when compared to the non-transactional case) that we attributed to additional processing in the Display processes due to the introduction of transactional semantics. Otherwise, no significant changes were observable in the case of `CreateApplication()` for the API instance or any of the other involved processes. In the context of `Transition(VISIBLE)`, we witnessed an increase of the processing time required by Handlers by an average of 74% (when compared to the non-transactional case) that we again attributed to additional processing overhead due to the introduction of transactional semantics. However, as can be seen in figure 6.12(b) the mean processing times spent by Handler processes remained more or less constant as the number of involved Displays increased.



(a) `CreateApplication()` (b) `Transition(VISIBLE)`

Figure 6.12: Graphical breakdown of processing and communication delays in the case of transactional operations performed on one, two and three displays using group operations.

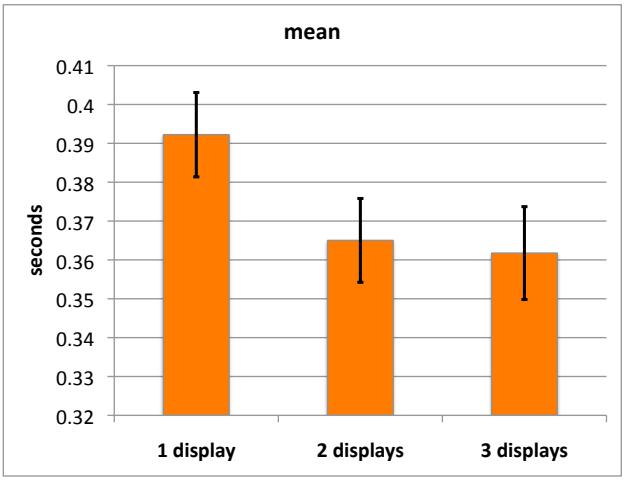


Figure 6.13: Delays between the invocation of `commit()` and the receipt of the first status event showing the Application to be in state `VISIBLE`.

We also witnessed a decrease in the processing time expended by Application processes as part of `Transition(VISIBLE)` operations from an average of 346 milliseconds in the non-transactional case to an average of 26 milliseconds in the transactional case. This decrease is caused by the fact that in a transactional context, the visibility and playback state is not changed until the transaction is committed. As a result, the GUI and playback operations that are carried out in the non-transactional case as part of processing a `Transition(VISIBLE)` operation are performed as part of processing the `commit()` operation in the transactional case.

To provide figures about the amount of time that typically passes from the time a call to `commit()` is made until the content becomes visible on the targeted public displays, we measured the average delays between the invocation of `commit()` and the receipt of the first status event showing the Application to be in state `VISIBLE`. As is shown in figure 6.13, the overall average delay is about 370 milliseconds. The average delays did not show any increases as additional displays were added: during the individual runs we recorded average delays of 0.392 ± 0.011 s (standard deviation 0.054 s) in the case of one display, 0.365 ± 0.011 s (standard deviation 0.054 s) in the case of two displays, and 0.362 ± 0.012 s (standard deviation 0.060 s) in the case of three displays.

6.5 Scalability Analysis

The scalability of our software infrastructure is primarily bounded by the maximum number of concurrent subscriptions supported by the Elvin server and by the server's maximum throughput in messages per second. In this section we investigate the scalability of our software infrastructure by calculating the requirements placed onto the Elvin event server by increasing the numbers of Displays and Schedulers.

For our calculation we assume that the public display network consists of N_d computers, each of which represents a conceptual display and hosts one Display process and at least one Handler process. Additionally we assume that each Display hosts one Application process that is currently visible.

Moreover, we assume that N_s individual Scheduler processes exist in the system and that each Scheduler attempts to replace the content on one of the Displays with a different content item.

To achieve this Schedulers perform the following operations (shown in pseudo-notation):

```
new_app = CreateApplication( display_n, new_content )
ChangeState( new_app, PREPARED )
Transition( old_app, NOT_VISIBLE )
Transition( new_app, VISIBLE )
TerminateApplication( old_app )
```

To obtain upper bounds for the number of events we assume that these operations are performed without transactional semantics. While the use of transactional semantics would result in one additional protocol message (a `COMMIT` or `ABORT` message) being emitted by each Scheduler when compared to the non-transactional case, the use of non-transactional operations leads to an overall greater number of events: using transactional semantics, processes only announce their final state once the transaction has been committed, while in the case of non-transactional semantics every state change of each process is fully observable, i.e. a `STATUS` event is emitted by each process immediately after the state change. If we assume that different Schedulers operate on different Displays, then carrying out the operations presented above using non-transactional semantics results in a minimum of eight `STATUS` messages being emitted:

- `CreateApplication()` triggers a minimum of two `STATUS` messages: one message from the involved Display process and one message from the newly created Application process. Moreover, it may trigger additional `STATUS` events from Handler processes.
- `ChangeState()` triggers one `STATUS` message from the Application process.
- `Transition()` triggers a minimum of two `STATUS` messages: one message from the Display process and one message from the Application process. It may trigger additional `STATUS` messages from Handler processes. Please note that each Scheduler invokes `Transition()` twice, leading to a total of at least four `STATUS` messages being emitted as result of `Transition()` invocations.
- `TerminateApplication()` triggers at least one `STATUS` message, which is emitted by the Display process. It may trigger additional messages from any Handlers that are involved.

Let:-

- $h_{create,i}$ represent the number of handlers that are associated with Display i and configured to intercept `CREATE_APPLICATION` protocol messages.
- $h_{transition,i}$ represent the number of handlers that are associated with Display i and configured to intercept `TRANSITION` protocol messages.
- $h_{terminate,i}$ represent the number of handlers that are associated with Display i and configured to intercept `TERMINATE_APPLICATION` protocol messages.
- $d(s)$ be a function that, given the identity of a Scheduler s, returns the identifier of the conceptual display that s attempts to replace content on.

We are thus able to describe the number of `STATUS` events triggered by the scheduling activities of Scheduler s as:

$$n_{status,s} = 8 + h_{create,d(s)} + 2 \cdot h_{transition,d(s)} + h_{terminate,d(s)} \tag{6.2}$$

The invocation of API operations also results in the exchange of protocol requests and responses:

- a minimum of two messages for `CreateApplication()`: a `CREATE_APPLICATION` request and an accompanying `RESULT` event (any `STATUS` events emitted by the newly created Application process are already included in the total number of `STATUS` events presented above). In addition, if Handler processes are configured to process `CREATE_APPLICATION` events, an additional pair of request and response events per Handler process is issued for the communication between the Display process and that Handler process.
- a request event and a response event for the `ChangeState()` operation.
- a minimum of four events for each `Transition()` operation, i.e. one `TRANSITION` request that causes the Display process to issue a `CHANGE_STATE` request, and the corresponding responses to each of these requests. In addition, if Handler processes are configured to process `TRANSITION` events, an additional pair of request and response events per Handler process is issued for the communication between the Display process and that Handler process.

- a minimum of four events for the `TerminateApplication()` operation, i.e. one `TERMINATE_APPLICATION` request that causes the Display process to issue a `CHANGE_STATE` request, and the corresponding responses to each of these requests. In addition, if Handler processes are configured to process `TERMINATE_APPLICATION` events, an additional pair of request and response events per Handler process is issued for the communication between the Display process and that Handler process.

The total number of request and response events emitted by a Scheduler s is therefore:-

$$n_{rq,s} = 16 + 2 \cdot h_{create,d(s)} + 4 \cdot h_{transition,d(s)} + 2 \cdot h_{terminate,d(s)} \tag{6.3}$$

All processes periodically emit `STATUS` events. We assume that each process emits a `STATUS` event every t seconds. If a `STATUS` message has been triggered by a state change, the process-internal timer is reset, i.e. the process will not emit another `STATUS` event until either t seconds have passed, or another state change occurs. Moreover, we assume that the duration t is much longer than the duration required by one of our Schedulers to carry out all its operations.

Therefore in the worst case Display processes, Handler processes and existing Application processes all emit one additional `STATUS` message during the period that a Scheduler is carrying out its operations, leading to

$$n_{ps,i} = 2 + h_{create,i} + h_{transition,i} + h_{terminate,i} \tag{6.4}$$

additional `STATUS` messages on display i. The total number of events directly triggered by all N_s Schedulers is:

$$\begin{aligned} n_{sched} &= \sum_{j=1}^{N_s} n_{status,j} + \sum_{j=1}^{N_s} n_{rq,j} \\ &= \sum_{j=1}^{N_s} (8 + h_{create,d(j)} + 2 \cdot h_{transition,d(j)} + h_{terminate,d(j)}) \end{aligned} \tag{6.5}$$

$$
\begin{aligned}
& + \sum_{j=1}^{N_s} (16 + 2 \cdot h_{create,d(j)} + 4 \cdot h_{transition,d(j)} + 2 \cdot h_{terminate,d(j)}) \\
= & \sum_{j=1}^{N_s} (24 + 3 \cdot h_{create,d(j)} + 6 \cdot h_{transition,d(j)} + 3 \cdot h_{terminate,d(j)})
\end{aligned}
$$

Moreover, the total number of unsolicited `STATUS` messages on all N_d displays is:

$$
\begin{aligned}
n_{ps} &= \sum_{i=1}^{N_d} n_{ps,i} \\
&= \sum_{i=1}^{N_d} (2 + h_{create,i} + h_{transition,i} + h_{terminate,i})
\end{aligned} \tag{6.6}
$$

From our measurements we have learned that a sequence of `CreateApplication()`, `ChangeState(PREPARED)` and `Transition(VISIBLE)` operations takes approximately 3 seconds to complete. To obtain theoretical worst-case figures for the number of events per second that are to be supported in our scenario by the Elvin server, we assume that our software infrastructure enables each Scheduler to perform all of its API operations within a single second, yielding a total of

$$
\begin{aligned}
n_{events} &= n_{sched} + n_{ps} \\
&= \sum_{j=1}^{N_s} (24 + 3 \cdot h_{create,d(j)} + 6 \cdot h_{transition,d(j)} + 3 \cdot h_{terminate,d(j)}) \\
&\quad + \sum_{i=1}^{N_d} (2 + h_{create,i} + h_{transition,i} + h_{terminate,i})
\end{aligned} \tag{6.7}
$$

events within that second.

For each conceptual display, individual subscriptions exist for each Display process, for all Hander processes, and for each of the $n_{app,i}$ Application processes hosted on the display. The total number of subscribers can therefore be expressed as follows:

$$
n_{sub} = N_s + \sum_{i=1}^{N_d} (1 + h_{create,i} + h_{transition,i} + h_{terminate,i} + n_{app,i}) \tag{6.8}
$$

Figure 6.14 shows the throughput requirements in events per seconds based on equation 6.7 for various numbers of Displays, Handlers and Schedulers. To obtain worst-case figures we have assumed that all Handlers are configured to handle `TRANSITION` events. Please note that in general we would expect the number of Schedulers in a typical public display network to be significantly smaller than the number of Displays. For example, we would expect a deployment of 500 public displays to have around 50 Schedulers operating concurrently in the display network. However, cases are imaginable in which each public display is managed by its own Scheduler.

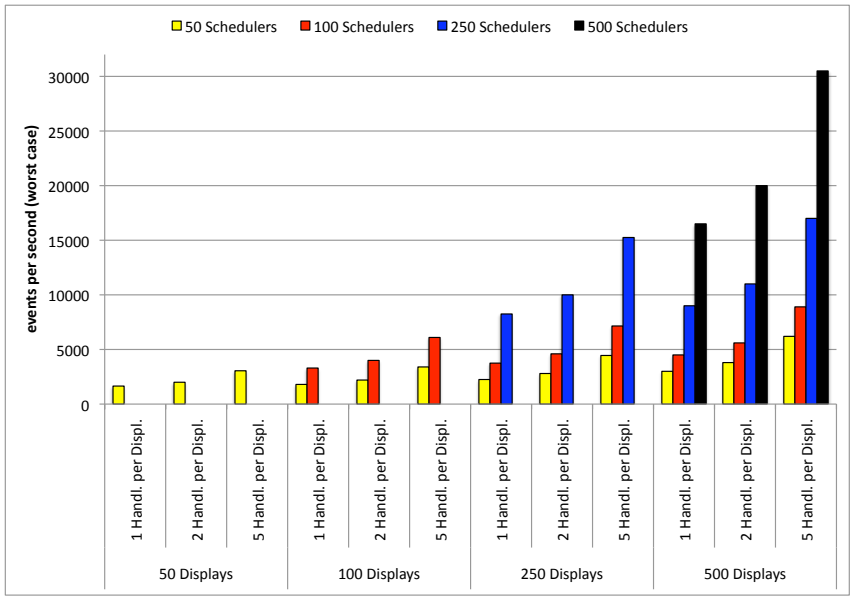


Figure 6.14: Worst-case throughput requirements in events per second for various numbers of Schedulers, Displays and Handlers, based on our scenario.

The throughputs shown in figure 6.14 are based on the assumption that every Scheduler is capable of completing all its scheduling operations within a total of one second. However, based on our measurements we can assume that in reality Schedulers require at least 3 seconds to process the five operations that form the basis of these calculations. To obtain more realistic figures we further assume that in reality not all Scheduler operate in total synchrony, and that therefore the events generated as a result of these Schedulers' operations are more or less evenly distributed within these three seconds, and that the same is true for the unsolicited `STATUS` events emitted by the various processes. The resulting throughput requirements are depicted in figure 6.15.

Finally, figure 6.16 shows the expected number of subscribers for the various numbers of Schedulers, Displays and Handlers.

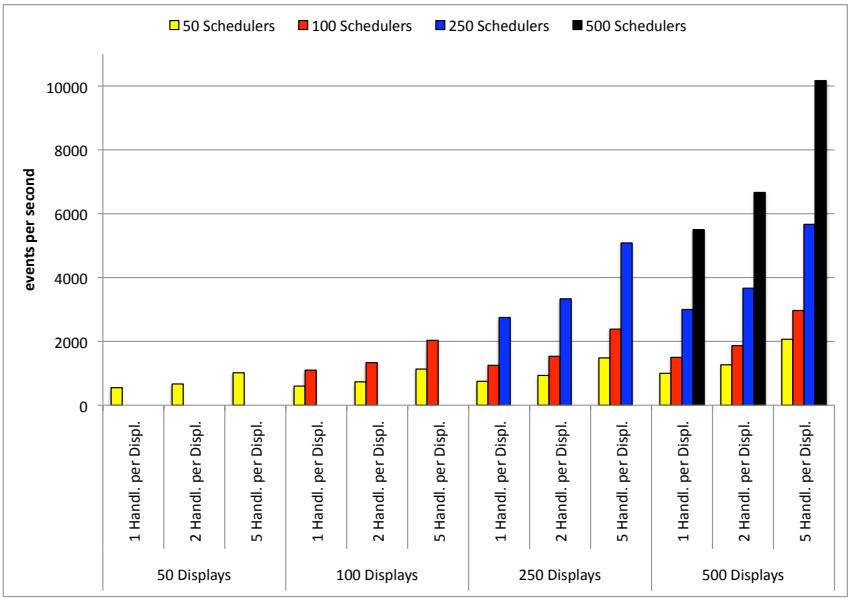


Figure 6.15: More realistic throughput requirements based on the measurements presented in section 6.4.

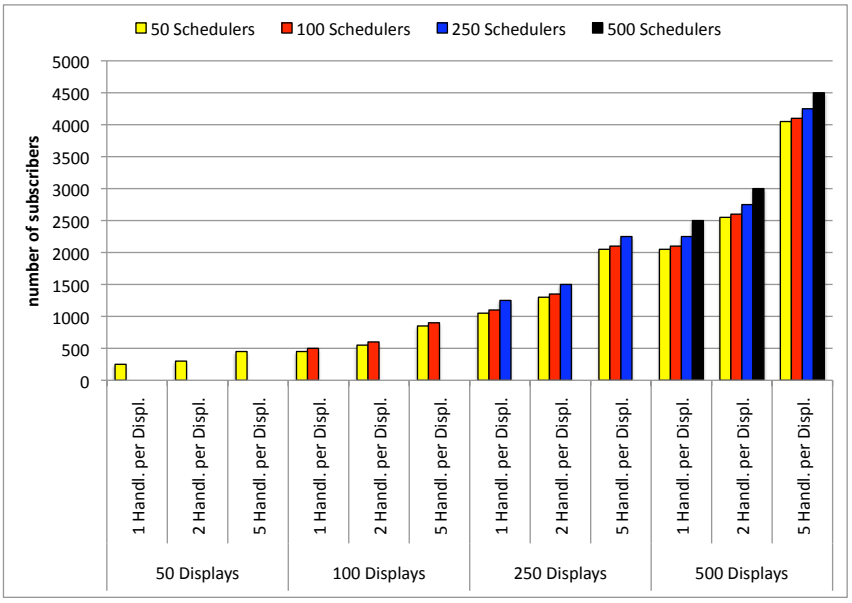


Figure 6.16: Number of subscribers for various numbers of Scheduler, Displays and Handlers.

We were unable to obtain any up-to-date performance figures from the literature for the current implementation (version 4.2) of the Elvin server. The most recent concrete statements regarding the performance of Elvin were published in 1999 by Arnold et al. and Fitzpatrick et al.:

> "An Elvin3 server on an AlphaStation 4/255 workstation can perform approximately 200,000 attribute matches per second, and sustain a throughput of 20,000 messages per second (with 50 active subscriptions and a 10% success rate in subscription evaluation)" [ASB+99]

> "A single Elvin server can effectively service thousands of clients (producers or consumers) and evaluate hundreds of thousands of notifications per second on moderate hardware platforms" [FMK+99]

The performance figures quoted by Arnold et al. are based on version 3 of the Elvin server. Moreover, in the same publication the authors state about version 4 of the Elvin server (the version used for our work) that they "expect significant performance gains from this latest implementation" [ASB+99].

Provided the advances in computer hardware since 1999 and the statements made by Fitzpatrick et al. in 1999, we therefore assume that the current version of the Elvin server is capable of supporting the required throughput and number subscribers to enable our software infrastructure to scale up to a few hundred displays. We acknowledge that we cannot make this claim with absolute certainty until the performance of the current Elvin implementation has been measured on current computer hardware, and that such detailed performance measurements are outside the scope of this thesis. However, our software infrastructure is based on relatively simple subscription expressions. We are therefore confident that using current computational hardware an event channel with the desired properties can be implemented that provides the necessary performance to underpin our software infrastructure in the context of medium-size public display networks.

6.6 Trials

The software infrastructure presented in this thesis has been used for displaying a range of experimental and day-to-day content. As we explained above, the majority of the day-to-day content

has until recently been handled by the digital signage software that is installed on the displays. We therefore used our own software infrastructure mainly for showing interactive, on-demand, or context-sensitive content, since these types of content could not be displayed using the digital signage software. In the following sections we describe our experiences of using the software infrastructure.

6.6.1 The Map Application

The Map application was the first interactive application that we developed using our software infrastructure. The application was created to help freshers at the start of the new academic year to find lecture theatres and other university facilities. The application provided students with an interactive map of campus that allowed students to highlight their current position and a target destination of their own choice.

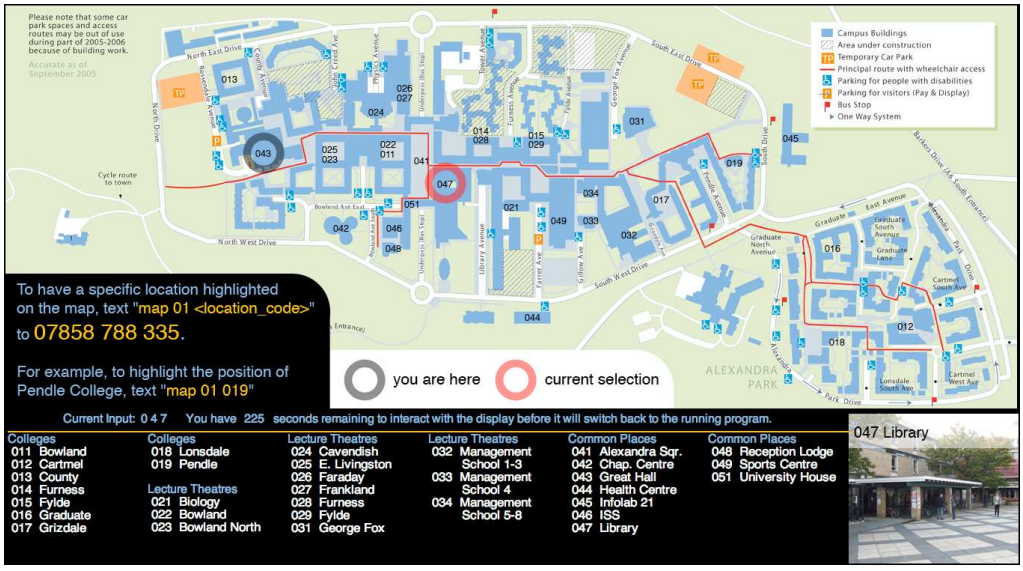


Figure 6.17: Screenshot of the Map application.

A screenshot of the resulting application is depicted in figure 6.17. The content itself was implemented as an HMTL page with embedded JavaScript code for handling interactions. The top part of the page showed a graphical map of campus, while the lower part displayed a list of possible destinations, each of which was identified using a unique number. The display also used coloured circles to highlight the user's (i.e. the display's) current position and, if specified, the location of a target destination.

The Map application was designed to be made visible on-demand. To request the map to appear on one of the displays, users were able to send SMS text messages that complied with a certain formatting schema to an SMS gateway on campus. For example, sending `map 01` caused the map to appear on the display with the number `01`. Once the map was visible, users were able to further interact with the application using SMS text messages. For example, texting `map 01 013` caused the location corresponding to location code `013` (County College) to be highlighted on the map (see figure 6.18). Once visible on a public display, the Map application stayed visible for a duration of 5 minutes, after which it was automatically removed from the display and the display returned to its regular programme. Additional interactions within the 5 minute period caused the time-out to be reset.

Figure 6.18: The Map application in use.

The Map application used the facilities for SMS-based interaction that we have described in section 6.2.5. A custom scheduler was deployed on each display machine capable of showing the Map. This scheduler was responsible for handling Elvin events representing Map interaction (received from the SMS gateway) and for using the low-level API to instantiate and control the visibility of the Map application accordingly. The scheduler was also responsible for translating location identifiers passed along in SMS text messages into keystroke events and for injecting these events into the operating system. These keystroke events were in turn received by a JavaScript

event handler embedded into the Map HTML page, and used by the logic of the event handler to determine which location on the map to highlight.

Discussion

The Map application represented our first piece of interactive content that we built using our software infrastructure (see requirement R9: "support for interactivity"). The experiment-specific Scheduler that we developed successfully used the scheduling API provided by the infrastructure to show the map content on-demand (see requirement R5: "support for on-demand content") whenever the map was requested by users. The design of the Scheduler followed the basic blueprint for event-based schedulers that we discussed in chapter 4, section 4.5.3.

The Map Scheduler was successfully able to interrupt any background content that was currently active on the public display at the time the request was made. We achieved this by assigning a higher priority to the Map content and by using the software infrastructure's ability to arbitrate between conflicting pieces of content (see requirement R7: "support for priorities and preemption", and requirement R3: "arbitration between conflicting pieces of content").

Moreover, the Map content was designed to stay on the display for five minutes after the last interaction had occurred. At the time the Map application was made visible on a display, it was therefore not foreseeable, how long the content would have to be shown for. The experiment-specific Scheduler of the Map application was successfully able to demonstrate the ability of the software infrastructure to provide support for such content items of dynamic length (see requirement R10: "support for content of dynamic length").

The Map application also demonstrated the benefits of not having to execute Schedulers on dedicated machines in the public display network (see design proposition DP2). If we had not been able to deploy Schedulers onto the actual display machines, we would not have been able to use them to inject keyboard events into those systems.

The Scheduler used the arrival of SMS-based requests as criteria for showing the Map content on public displays. The Map Scheduler therefore showed how experiment-specific scheduling criteria could be used to schedule content using our scheduling API (see R4: "support for a wide range of scheduling criteria").

6.6.2 The Bus Timetable Application

The Bus Timetable application was designed to allow students, staff and visitors on campus to obtain information about the departure times of buses leaving campus. An overview of the architecture of the Bus Timetable application is shown in figure 6.19. The application consisted of three main parts: an experiment-specific parsing and handling script that was installed on the SMS gateway, an experiment-specific Scheduler, and a PHP application responsible for generating the actual content.

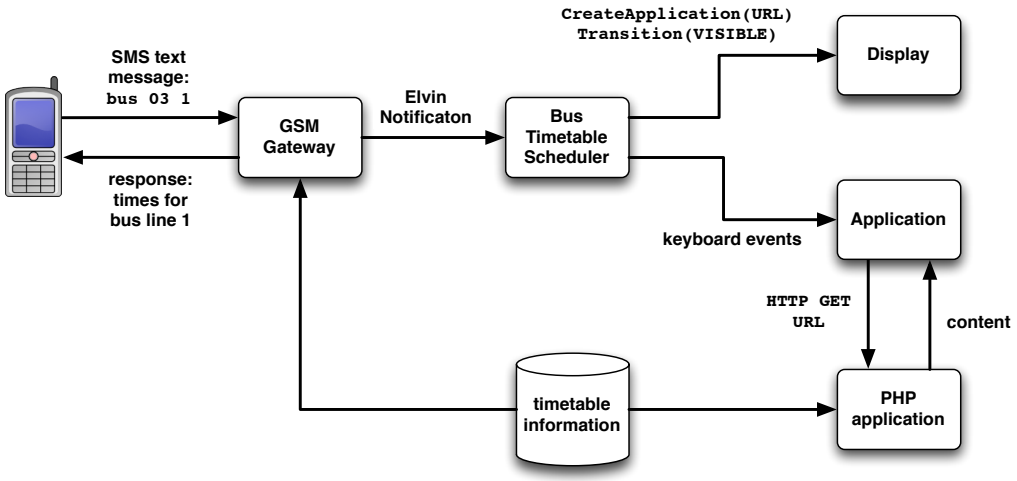


Figure 6.19: Architecture of the Bus Timetable application.

Similar to the Map application, users of the Bus Timetable application could use their mobile phones to request an overview of the departure times of buses leaving from a bus stop close by to a specific public display. For example, users were able to send a text message containing the text `bus 03` to a dedicated number to have timetable information displayed on the public display identified by Display Identifier "ecampus-03". A screenshot of the resulting output is shown in figure 6.20. Users were subsequently able to send additional text messages to the gateway. A text message containing `bus 03 3` could, for example, be used to request more detailed timetable information about bus line number 3 to be displayed, including a list of stops and the times the bus was scheduled to arrive at those stops. In addition, a copy of the timetable was sent back to the user's phone using an SMS text message, essentially enabling the user to take the information shown on the display away with him.

The experiment-specific parsing and handling script on the SMS gateway was responsible for processing incoming text messages that were related to the Bus Timetable application. Information

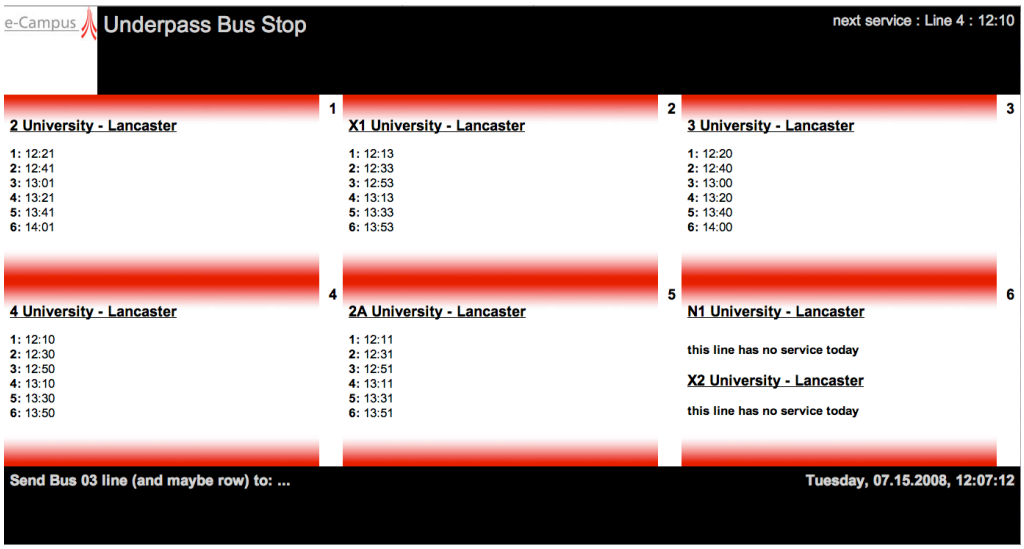


Figure 6.20: Screenshot of the Bus Timetable application.

about incoming requests were communicated to experiment-specific Schedulers, each of which was responsible for showing the Bus Timetable on one conceptual Display. The content was generated by a PHP back-end that produced HTML output that was customised for each individual display location. We used our standard Application type capable of rendering Web-based content to display the output produced by the PHP back-end. Once the Application was visible, keyboard events were used for additional interaction with the application, e.g. if a user requested additional information about a particular bus line. The task of injecting keyboard events into the operating system of the respective display machine was performed by the Schedulers.

Timetable information was held in a centralised database that was available to the PHP back-end application and the parsing and handling script running on the SMS gateway. The parsing and handling script used this information to generate responses to requests for detailed information about a particular bus line in the form of text messages.

The Bus Timetable application was designed and developed by Sabine Nowak during her internship with our research group.

Discussion

In terms of the demands imposed on our software infrastructure, the Bus Timetable application was very similar to the Map application. Both applications were scheduled on-demand by an

experiment-specific Scheduler as a result of user interaction. Interaction in both cases was based on SMS text messages. Like in the case of the Map application, the content associated with the Bus Timetable application stayed on the display for a certain period after an interaction had occurred. In both cases, content was Web-based. While the Map application used static content, content in the case of the Bus Timetable application was generated dynamically by a PHP application. This, however, remained transparent to our software infrastructure.

6.6.3 Capture the Campus

"Capture the Campus" (CTC) [SFD07] was a location-based game that used Bluetooth presence information to determine the location of game participants. The game was a public-display-based "capture the flag" derivative developed by Benjamin Sherratt as part of a student research project. The project was aimed at investigating research questions in three main areas:

- the feasibility of using Bluetooth technology as a source for location information, and the impact of its use on user acceptance,
- the level of user acceptance of augmented reality games that involve public displays,
- the willingness of users to download additional applications onto their mobile phones in order to further augment the game experience.

During the game e-Campus displays served two distinct purposes: they provided information and statistics about the ongoing game, but also represented the locations that had to be captured. Bluetooth scanners were used to detect the presence of game participants in the vicinity of displays. Participants were therefore required to carry discoverable Bluetooth devices, such as mobile phones or PDAs. Since CTC was deployed and trialed before our common Bluetooth scanning infrastructure was deployed, custom-crafted Bluetooth scanners were employed by CTC. However, just like the scanners in our current Bluetooth scanning infrastructure, the CTC scanners disseminated presence information using Elvin events.

The game logic in CTC was distributed to three main types of processes that all communicated using Elvin events: a CTC game server, CTC Applications, and a CTC marshal console. An overview of the architecture of Capture the Campus is shown in figure 6.21. At the heart of

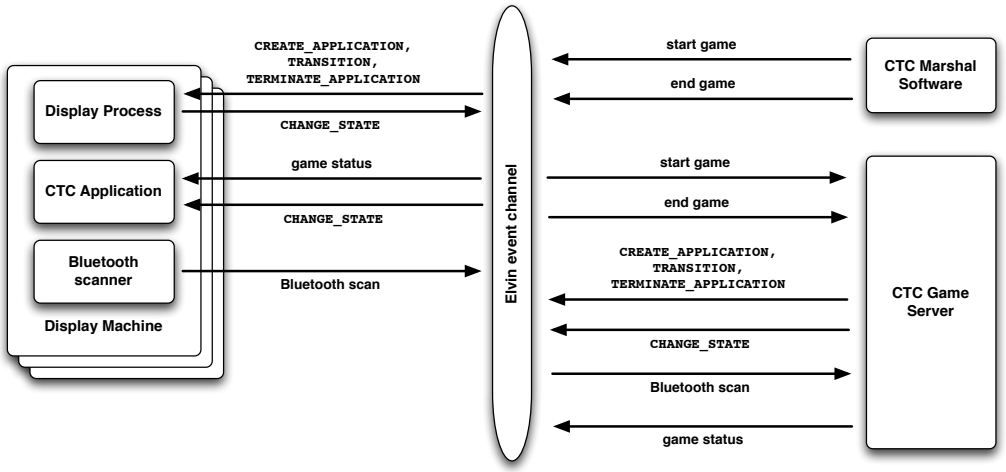


Figure 6.21: Overview of the architecture of Capture the Campus.

the architecture was the CTC game server that implemented the bulk of the game logic. Once the server had received the signal from the CTC marshal console to start a game, the server used our software infrastructure and API to create instances of the CTC Application on each participating Display, and to make these Application instances visible. If instructed by the marshal console to end the game, the API was used again to remove the Application instances from the displays. The CTC game server therefore represented a custom-crafted Scheduler that was specifically developed for this experiment. Besides controlling the life-cycle and visibility of the CTC Applications, the CTC game server was also responsible for receiving Elvin events containing Bluetooth presence information, for calculating the game's status and current scores based on this presence information, and for disseminating status and score information to the CTC Application processes using Elvin events. The game server also logged all game-related Elvin events. Based on these logs the server allowed previous games to be re-played to allow the experimenters to study participant behaviour during the game.

The CTC Application is an example for an experiment-specific Application type. One Application instance was displayed on each participating public display. Each CTC Application process showed an overview of the current scores, a map of campus that highlighted the locations that were to be captured, and whether these locations were currently captured. A screenshot of the CTC Application is shown in figure 6.22. CTC Application processes received status and score updates from the CTC game server in the form of Elvin events and used these updates to modify their output accordingly. The CTC Application was implemented in Java.

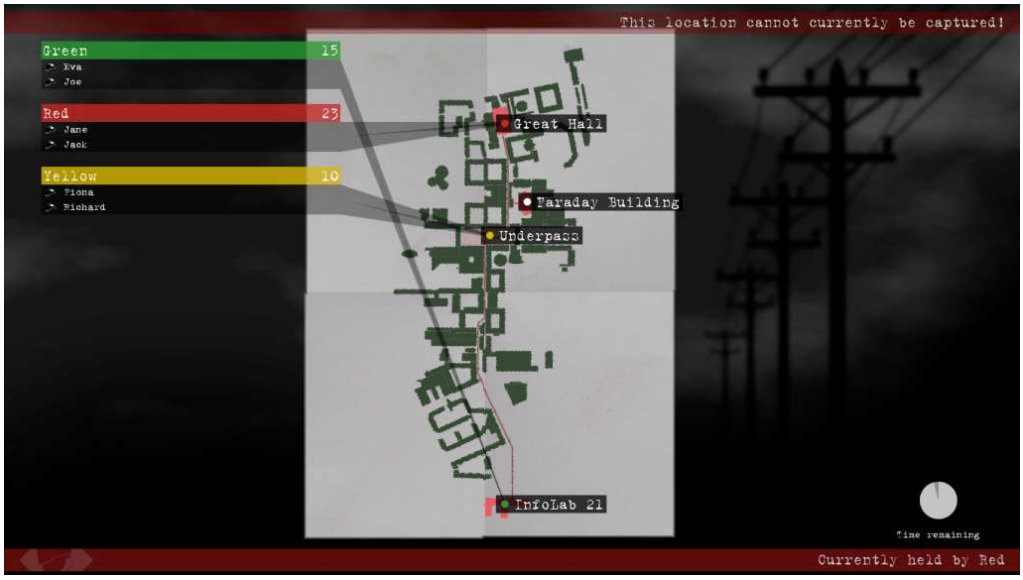


Figure 6.22: Screenshot of Capture the Campus.

Besides providing the means for starting and stopping game rounds, the CTC marshal console allowed the game marshal (or administrator) to manage the participants of each game round. This involved scanning for Bluetooth devices to obtain the hardware addresses of the Bluetooth devices used by the participants, and mapping these hardware addresses to player aliases and teams.

A small user trial was conducted, involving a total of six participants. Four e-Campus screens that were distributed across the campus of Lancaster University were used during the trial, including one of the projected displays in the Underpass. In addition to the re-play logs gathered by the game server, data was collected using observations and questionnaires.

Discussion

Capture the Campus successfully demonstrated the use of our scheduling API to support a piece of experimental content. The content was shown on demand (see requirement R5: "support for on-demand content") when the game was started by a marshal. The content remained visible for a dynamic period of time (see requirement R10: "support for content of dynamic length") until the marshal gave the signal to end the game.

It only made sense to start the game if the CTC Application could be shown on all of the involved displays (see requirement R13: "support for atomicity and isolation"). The CTC game

server therefore used the transactional features of our scheduling API to create the instances of the CTC Application on each involved display, and to make these visible, thereby treating all Application instances as atomic unit.

Once instantiated, the CTC game server controlled the life-cycle and visibility of CTC Application processes using bulk operations. This means that instead of controlling each Application process on its own, single operations were used to control all participating Applications (see requirement R6: "support for bulk operations").

The Capture the Campus game was context-sensitive (see requirement R8: "support for context-sensitivity and personalisation"). The game used Bluetooth scanners to obtain presence information and used this form of context information as input to the game logic. The CTC game employed a centralised game server that obtained this context information and relayed the resulting updates to the game's status to the CTC Application processes using Elvin events.

While Capture the Campus did not monitor the exchange of protocol requests and responses to create audit trails, it demonstrated nevertheless how the monitoring of Elvin events (in the case of CTC, events that were generated by the game-internal protocol) can be used to observe experiments and to create audit trails (see requirement R14: "support for audit trails").

Despite the fact that game output was rendered on a mixture of LCD displays and a projected displays, the differences in display hardware and how this hardware was controlled (e.g. powered on) remained transparent to the CTC game server. The CTC game server was able to simply use the operations exposed by our scheduling API (`CreateApplication`, `ChangeState`, `Transition`, `TerminateApplication`) to control the life-cycle and visibility of content, while Handler processes took care of configuring the display hardware to reflect the intended visibility. Capture the Campus therefore exemplified the ability of our software infrastructure to support "non-standard and dynamic hardware setups" (see requirement R15).

6.6.4 e-Campus Monitoring

The monitoring system was developed by John Hardy as part of a student research project. The aim of the project was to investigate means for monitoring and debugging public display networks.

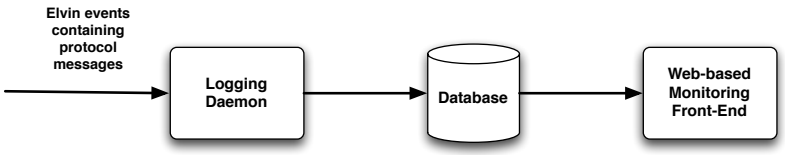


Figure 6.23: Abstract overview of the architecture of the monitoring tool.

Figure 6.23 shows an abstract overview of the architecture of the monitoring system. The system consisted of a logging daemon that subscribed to the `STATUS` events that are emitted by the various processes of our software infrastructure. The monitoring system stored these events in a relational database. A web-application was used to analyse the events in the database and to generate the user interface of the monitoring system. The monitoring system, for example, provided information about the health of individual system processes and was able to detect if faults in these processes had occurred. Figure 6.24 shows a screenshot of the web-based user interface.

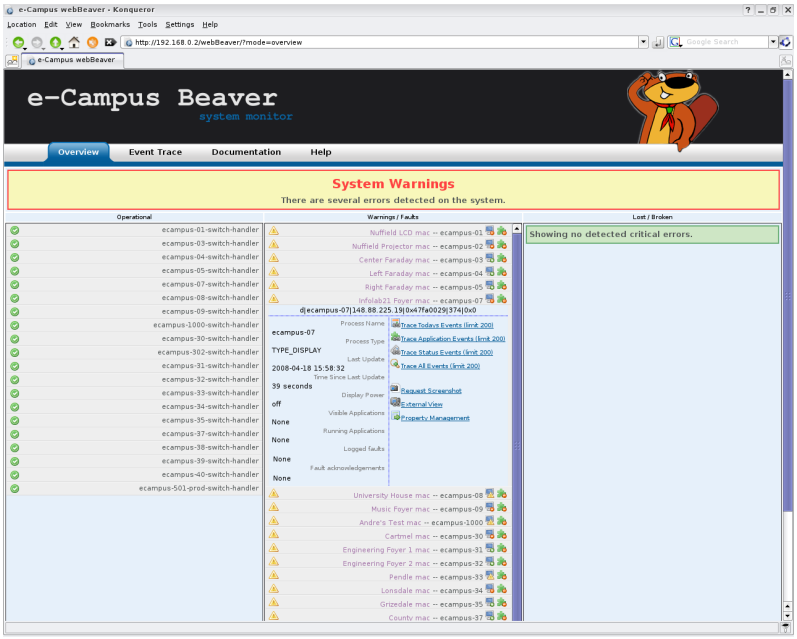


Figure 6.24: Screenshot of the monitoring tool. (Screenshot: John Hardy)

Discussion

The monitoring system is another example showing how the properties of the event channel can be used to construct tools for monitoring and auditing the activities in the public display network.

While the monitoring system was targeted at recording and monitoring the state of our software infrastructure, the same mechanisms (i.e. the monitoring of events) can be used to construct tools that generate audit trails for individual experiments. Using these mechanisms, our software infrastructure is therefore able to support the construction of audit trails, which was one of the requirements (see requirement R14: support for audit trails).

6.6.5 The Crossword Application

The Crossword application was aimed at investigating the construction of content for public displays that actively engaged users instead of treating them as passive consumers. The application was created by Aaron Gregory in the context of a student research project. The investigations were based on an interactive crossword puzzle that was shown on the public displays and that users could solve collaboratively using their mobile phones by sending SMS text messages containing proposed solutions.

An overview of the architecture of the Crossword application is shown in figure 6.25. The text messages containing proposed solutions were received by SMS gateway software that parsed the text messages, extracted the proposed solutions and compared these with the actual solutions that were stored in a central database. If the proposed solution matched the correct solution, the state of the crossword puzzle, which was maintained in the database, was modified accordingly to reflect that a solution had been found for a particular clue. Moreover, both correct and incorrect guesses were recorded in the database and reflected on the public displays in order to provide feedback to users. Users were also able to submit a nickname along with their guesses that was displayed along with the feedback. The actual output that was visible on the public displays was generated by a PHP script that produced an auto-refreshing HTML page reflecting the current state of the crossword puzzle.

To evaluate the Crossword application, user trials were conducted on three non-consecutive days involving a total of 27 participants. On the first day a single public display was used to show the Crossword application. On the other two days a second display that was installed side-by-side with the first display was added. This second display was used to show a scoreboard listing the number of correct and incorrect guesses for each participant.

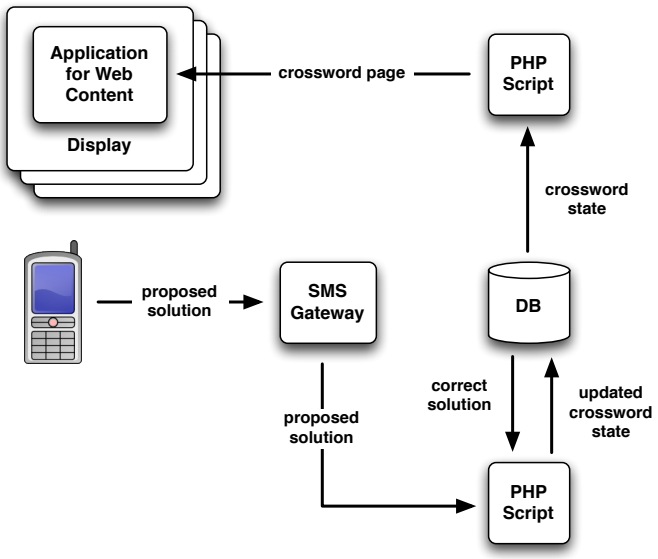


Figure 6.25: Overview of the architecture of the Crossword application.

The Crossword application was scheduled using a simple, command-line-based Scheduler that allowed the experimenter to display the Crossword content at the start of the experiment, and to remove it again at the end. In a permanent, non-experimental deployment of the application, the Crossword application could either be scheduled on-demand or for fixed periods during certain times of the day. In the former case, the mechanisms for scheduling the content are similar to those used in the context of the Map application and the Bus Timetable application, i.e. the Crossword content is shown as a reaction to incoming text messages targeted at the Crossword application, and is kept on the display for a certain period after an interaction has taken place. In the latter case we envision that there may be certain times of the day when the Crossword application is shown as default content. The displays in the Underpass might, for example, show the Crossword application in the late afternoon when large crowds of students are waiting for buses that take them back into town.

Discussion

Being Web-based content, the Crossword content could be displayed using an off-the-shelf Application type capable of rendering Web content, i.e. the experimenters did not have to create a custom Application type to support the experiment. As part of our requirements capture process we argued that while some research-related content would require the use of proprietary content

types and applications, in other cases content for experiments would be provided in the form of standard content types, such as web pages or images (see requirement R11: "support for standard content types and proprietary applications"). The Crossword application illustrates an example for an experiment that is based on standard content types, and shows that our software infrastructure is capable of supporting such content.

The Crossword application is also an example for interactive content that is using the facilities for SMS-based interaction that we provide. Since the interface with the database back-end was written in PHP, and was therefore unable to receive Elvin events, a custom Python script for parsing and handling incoming SMS messages was deployed onto the SMS gateway. On receipt of a Crossword-related text message, the script directly called the corresponding PHP script to process and record the proposed solution. In summary, the Crossword application illustrates the ability of our software infrastructure to support experiments that involve interactivity (see requirement R9).

The introduction of the additional scoreboard that was shown on a second display resulted in the need to show both content items (crossword content and scoreboard) as an atomic unit (see requirement R13). Our simple Scheduler was able to meet this requirement by wrapping operations into transactional blocks.

6.6.6 Requesting Content Using Bluetooth Friendly Names

Our group is investigating mechanisms for enabling users to interact with public displays using their mobile phones, but without having to download and install additional software onto their mobile phones. One particular approach is to allow content to be requested by encoding these requests into the device names of users' mobile phones. These device names, which are often called "friendly names", are user-settable and are usually retrieved as part of the process of discovering Bluetooth devices. As such, they are also retrieved by our Bluetooth scanning infrastructure. In our particular approach users change the friendly names of their mobile devices into a specific format that identifies the content they are requesting. For example, the friendly name `ec flickr oranges` represents a request for photos that are tagged with the keyword "oranges" from the photo sharing website Flickr [Fli08a]. The generic format for requests is `ec <content_type> <search phrase>`, where `content_type` determines the type of content that is to be displayed,

and `search_phrase` represents one or more words that further qualify the content that is to be displayed. Besides "flickr", other supported content types are:

- `map`, showing the specified location on Google Maps [Goo08c];
- `youtube`, displaying first YouTube [You08a] video matching the provided `search_phrase`;
- `google`, showing the first Google [Goo08a] match for the given search phrase;
- `music`, playing one of our royalty-free audio tracks that matches the search phrase;
- `wikipedia`, showing a Wikipedia [Wik08] article corresponding to the search phrase.

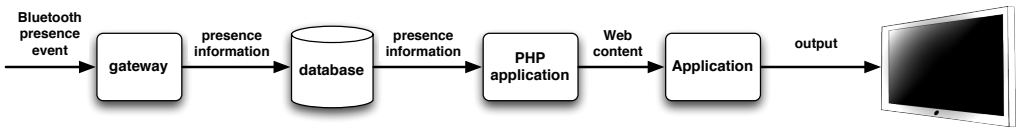


Figure 6.26: User interaction using Bluetooth friendly names.

The architecture of our prototype is depicted in figure 6.26. The prototype uses a gateway to receive Elvin events containing Bluetooth presence information. These events are generated by the Bluetooth scanners that we have deployed with each public display installation, and include the Bluetooth hardware information and friendly names (potentially representing encoded requests) of each Bluetooth device that is in range of the scanners. The gateway stores these events in a relational database. At the heart of our prototype is a PHP-based web application that, when provided with a Display identifier, queries the database and retrieves the requests that were issued recently in the vicinity of that Display. The PHP application queues these requests and generates a self-refreshing web page serving the received requests in a first-come-first-served order. Figure 6.27 shows a screenshot of the output generated by the PHP application.

At least two options exist for scheduling the application. The first option is to schedule the application on-demand if Bluetooth devices containing valid requests are in the vicinity of displays. In this case, a custom Scheduler is used that directly subscribes for Bluetooth presence events. If devices with valid requests are present, the content produced by the PHP application is shown. The Scheduler communicates with the PHP application to determine when the application has finished serving all requests, at which time the Scheduler removes the application from the display in question. The second option for showing the application is to simply display the application at

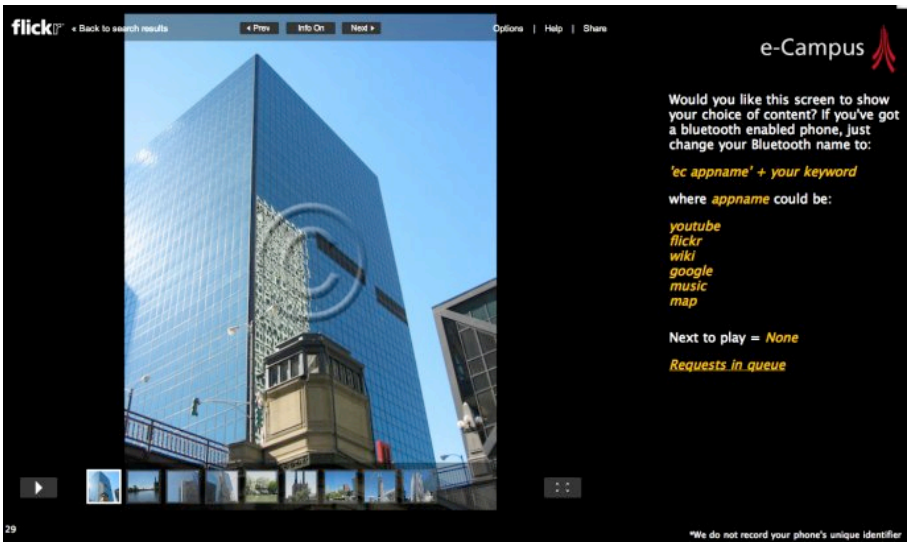


Figure 6.27: Screenshot of a prototype of the Bluetooth friendly name application. (Photos contributed by the author.)

certain times during the day for per-determined periods of time using a time-based Scheduler. We have already implemented and trialed an initial prototype of an on-demand Scheduler and continue will explore both options for scheduling the Bluetooth-based content as part of our investigations.

Discussion

Our investigations into requesting content using Bluetooth friendly names illustrate the use of our software infrastructure and the e-Campus public display network for experimentation with novel mechanisms for interacting with content on public displays (see requirement R9: "support for interactivity"). They also illustrate the use of the Elvin event channel to disseminate information obtained from interaction devices to other processes in the public display system. In this case the Bluetooth scanning framework (that is in other contexts used as a source of context information) acts as a source of interaction information.

The output produced by the PHP application is Web-based. Researchers therefore do not have to create any specialised Application types to be able to show the output. Instead, our off-the-shelf renderer for Web-based content can be used (see requirement R11: "support for standard content types and proprietary applications").

If the content is shown whenever a Bluetooth device whose name represents a valid request

is in the vicinity of a display, then this content is essentially shown on-demand without prior planning, illustrating the ability of our infrastructure to "provide support for on-demand content" (requirement R5). Moreover, in the case of Youtube videos or music songs, the content has to be shown for varying durations that depend on the lengths of the individual videos or songs that are requested by the users. The example of requesting content using Bluetooth friendly names therefore also illustrates the support provided by our software infrastructure for content of dynamic length (see requirement R10: "support for content of dynamic length"). Moreover, this example shows how Schedulers can be constructed that use experiment-specific scheduling criteria, in this case the presence of a Bluetooth device with a name conforming to a pre-defined schema, to display content (see requirement R4: "support for a wide range of scheduling criteria").

6.6.7 Day-to-Day Content

While the majority of the day-to-day content has until recently been handled almost exclusively by the digital signage software, we have used the software infrastructure and its scheduling API to support a limited number of day-to-day content items that were not related to research experiments.

Examples for day-to-day content items that were scheduled using our scheduling API include:

- a web-based photo slideshow that showed the candidate photos in a competition conducted by the Lancaster University Photographic Society,
- videos or electronic posters advertising cultural events, such as "Star City", a production by the Lancaster University Theatre Group, or the annual University of Lancaster Music Society Proms,
- advertisements for college balls,
- electronic posters or videos aimed at raising awareness for special topics, such as discrimination against disabled individuals, and discrimination against minorities in sport,
- announcements for guest lectures and workshops.

The content items were typically shown at specific times of the day for pre-defined durations on a subset of the public displays on campus. The content items were mostly scheduled using

simple Schedulers that requested an Application capable of rendering the content in question to be instantiated on the targeted display(s), and subsequently requested the instantiated content to be made visible. The Schedulers then typically waited for a pre-defined period of time, after which they removed the content from the display(s). An example of such a Scheduler is shown in figure 6.3.

Some of the schedulers implemented their own logic that allowed them to become active at specific times of the day. Other schedulers were simply executed by the cron daemon [The97] at the specified times.

Discussion

Our experiences of scheduling day-to-day content using our scheduling API demonstrated that our software experience is indeed capable of supporting not only experimental, research-related content, but also day-to-day content (see R2: "support for research-related and non-research-related content").

However, the need to author and execute separate Schedulers for each piece of content was found to be inconvenient, especially since the majority of these content items had similar scheduling requirements: the content was to be shown based on a small number of constraints, such as times of the day, durations, and the set of displays that the content was to be shown on. We revisit this observation in the next section.

6.7 Analysis

In this section we summarise the findings we made in the context of using our software infrastructure to support the trial applications described above and in the context of our performance overview and the scalability analysis, and compare these findings with our initial requirements presented in chapter 3.

Moreover, we review the feedback that we received from experimenters, as well as our own experiences with the development of Schedulers.

6.7.1 Summary of Match to Requirements

R1: Scalability up to a few hundred displays. We evaluated the scalability of our software infrastructure in section 6.5 using a series of calculations. The overall scalability of our software infrastructure is bounded by the maximum throughput and the maximum number of subscribers supported by the event channel, i.e. Elvin in the case of our implementation. Based on historic performance figures for Elvin and the advances in computer hardware that occurred since these figures we published, we concluded that the current Elvin implementation should be able to support the number of events expected in display networks of up to a few hundred displays. However, we also concluded that definitive claims could not be made until up-to-date performance figures for Elvin were obtained, and that obtaining these figures was outside the scope of this thesis.

R2: Support for research-related and non-research-related content. Our software infrastructure demonstrated its ability to support research-related content in the context of the trial applications described in section 6.6. Some of these experiments (Capture the Campus, e-Campus Monitoring and the Crossword application) were conducted in the context of student research projects, while experiments (The Map and Bus applications, and the use of Bluetooth friendly names for requesting content) were conducted as part of our own research.

As described in section 6.6, we also successfully used our software infrastructure to show various pieces of non-research-related, day-to-day content.

R3: Arbitration between conflicting pieces of content. The ability of our software infrastructure to arbitrate between conflicting pieces of content was demonstrated by the Map and Bus Timetable applications, where content was made visible as a result of user interaction. This interactive content preempted content that was currently shown on the displays using the infrastructure's priority-based arbitration and preemption facilities.

In general, the software infrastructure also ensures that once content with a certain priority level has been made visible on a display, the display can only be preempted by content of higher priority. This property ensures that experiments, once they have been made visible successfully, are allowed to run to completion unless higher-priority content requests access to the public display.

R4: Support for a wide range of scheduling criteria. The Schedulers constructed for day-

to-day content demonstrated the ability of our software infrastructure to support the construction of Schedulers that show content based on trivial constraints, such as dates, times of the day and the duration that content should be shown for.

In the context of research-related content Schedulers we used our software infrastructure, for example, to construct Schedulers that showed content based on external events that were a result of user interaction using custom, non-standard interaction techniques: SMS text messages in the case of the Map and Bus Timetable applications, and Bluetooth scans in the case of scheduling content based on Bluetooth friendly names. The same principles can be used to include other scheduling criteria into Schedulers.

R5: Support for on-demand content. The Schedulers constructed to support our interactive trial applications (the Map Scheduler, the Bus Timetable Scheduler and the Bluetooth-friendly-name-based Scheduler) have successfully demonstrated the use of our software infrastructure to show content on-demand as a result of user interaction.

R6: Support for bulk operations. In the case of "Capture the Campus", content on multiple displays had to be controlled simultaneously by the Scheduler. The use of group operations allowed the Scheduler to trigger state changes in all involved Applications without having to issue the same command repeatedly. The Scheduler supporting Capture the Campus therefore demonstrated the ability of our scheduling API to control multiple pieces of content on different displays using bulk operations.

R7: Support for priorities and preemption. Our interactive trial applications (i.e. the Map application, the Bus Timetable application and potentially the application for using Bluetooth friendly names to select and show content) were designed to make content visible on-demand by preempting the public displays from background content that might be visible on the displays at the time of interaction. The Schedulers associated with these experiments achieved this by making their content visible with a priority value that was higher than the typical priority value of non-interactive background content. The interactive content was therefore able to preempt the displays according to the policies embedded into Display processes.

R8: Support for context-sensitivity and personalisation. The Capture the Campus game represented a context-sensitive experiment that was supported by our software infrastructure. Context information gathered by our Bluetooth scanning framework was received and

processed by the central game server. The results of this processing were distributed by the game server to the game-specific Application processes using an experiment-specific protocol.

R9: Support for interactivity. The Map application, the Bus Timetable application, and the experiment into displaying content based on Bluetooth friendly names all demonstrated the use of our software infrastructure to show interactive content. In the case of the Map and Bus Timetable applications, interaction events representing SMS text messages were received by the respective Schedulers that were deployed on each display machine. The Schedulers affected the content as a result of these interaction events by injecting keyboard events into the computers' operating systems, causing the content to be adapted accordingly. The Schedulers demonstrated the ability of our software infrastructure to support Schedulers that require access to native APIs (e.g. for generating keyboard events) of the display computers.

In the case of the ongoing Bluetooth-based experiment, Bluetooth friendly names are used by users to request content to appear on public displays. Presence information obtained by Bluetooth scanners is temporarily stored in a database, from where it can be retrieved by a PHP back-end and used to adapt the content that is shown on the displays.

The Bluetooth-based and the SMS-based experiments demonstrate the ability of our software infrastructure to support interactive content that is based on non-standard, experiment-specific interaction techniques.

R10: Support for content of dynamic length. Dynamic-length content was, for example, successfully used in the following trial applications: the Map application, the Bus Timetable application and Capture the Campus. In the case of the Map application and the Bus Timetable application, the duration for which a particular piece of content was to be displayed was determined by the duration of the users' interactions with that content and could therefore not be determined in advance. While Capture the Campus was based on fixed-duration game rounds, it was up to the game marshal to decide how many consecutive game rounds to play in one session. Moreover, game rounds typically involved a debriefing phase where participants gathered in one location, discussed the game round and inspected the final scores. It was therefore difficult to predict in advance for how long the game-related content would have to be shown on the displays.

R11: Support for standard content types and proprietary applications. While most of the content used in our trials was web-based, proprietary Java-based content was used in

the case of Capture the Campus, demonstrating the ability of our software infrastructure to support both standard content types and proprietary applications.

R12: Support for orchestrated performances. None of our trial content so far required content to be arranged into precisely defined sequences, and we were therefore not able to evaluate experimentally whether our software infrastructure and API are capable of fulfilling this requirement. However, our scheduling API offers precise temporal control over the instantiation and pre-loading of content, its playback states, and its visibility. We are therefore confident that our scheduling API is capable of supporting orchestrated performances.

R13: Support for atomicity and isolation. The ability of our software infrastructure to treat content items on different displays as atomic unit and to hide intermediate system states or failure states was demonstrated in the context of the Capture the Campus experiment. In this case, content was made visible in multiple locations on campus as atomic unit. Transactional semantics offered by our scheduling API were used to achieve the desired behaviour.

R14: Support for audit trails. Audit trails were generated in the context of Capture the Campus and the e-Campus Monitoring project. In both cases, the features of the event channel were used to monitor the exchange of protocol messages. The e-Campus Monitoring project observed the protocol messages generated by the processes in our software infrastructure to generate an overview of the public display system's state. Capture the Campus logged messages of the game-internal Elvin-based protocol to enable researchers to replay and analyse game rounds. Both cases have demonstrated the ability of our software infrastructure to support the generation of audit trails.

R15: Support for non-standard and dynamic hardware setups. The software infrastructure has been deployed on public displays using a diverse set of underlying display hardware. While all computers are based on the Mac OS X operating system, these computers are attached to a mixture of LCD displays and projectors. So far three different models of LCD displays from two manufacturers have been used. Each model employs its own proprietary RS-232-based protocol for powering the display on or off and for selecting display inputs. While we have so far deployed only projectors of a single type, the different deployment locations require different protocols for communicating with these projectors: using a direct RS-232 connection, or using a TCP-based protocol for communicating with the projectors via an AMX controller. The use of hardware-specific Handlers has enabled us to hide the

complexity of these underlying hardware configurations from the developers of Schedulers.

6.7.2 Shortcomings

As described in the previous section, our scheduling API meets the requirements outlined in chapter 3. However, based on our experiences of evaluating our platform we also noted a mismatch between the typical requirements of developers of content and the capabilities offered by the scheduling API. The specific issues that became apparent were a repetition of effort when implementing Schedulers, complexity involved in building Schedulers, complexity involved in executing Schedulers, language-dependence of the scheduling API, and an absence of user authentication. We provide a detailed discussion of each of these points in the remainder of this section.

Repetition of Effort

Using the low-level API for scheduling content, application developers not only had to develop the actual application or piece of content that was to be displayed, but also associated scheduling functionality for controlling the visibility of this content based on constraints, such as the time of day or the receipt of interaction events. For example, in the case of the Map application and the Bus Timetable application, schedulers had to be created that listened for incoming SMS text messages and controlled the visibility of the applications on individual displays accordingly.

While the creation of customised Schedulers was necessary in cases where scheduling decisions were made on the basis of experiment-specific interaction or context events, we realised that the requirements of Schedulers that were constructed to support our items of day-to-day content were very similar: content was to be made visible if a certain set of constraints was fulfilled. These constraints typically included date, time of day, and the displays that were to be used. Once visible, the content was to be played for a certain duration, after which it was to be removed from the displays. Similar scheduling requirements could also be found in the context of some of our experimental content. Examples include the Crossword application and Capture the Campus. In both cases user trials took place on a small number of days at particular times of the day. At the start of these trial slots, the content was made visible on a selected set of displays, and stayed visible until the end of the evaluation session. In both cases the content could therefore have been

scheduled based on constraints, such as the date, the time, the list of targeted displays and the duration.

We therefore have to acknowledge that besides experimental, interactive or context-sensitive content that most likely requires the construction of customised schedulers, a significant percentage of the remaining content is scheduled based on a common set of constraints. While our software infrastructure supports the implementation of constraint-based scheduling functionality on top of the low-level API – and this is indeed what most application developers did – this led to a repetition of effort: a Scheduler providing the functionality for scheduling content based on constraints had to be implemented for each piece of content, although most of these content items had common scheduling requirements.

Complexity Involved in Designing and Implementing a Constraint-Based Scheduler

We found that the added complexity of having to implement a fully functional constraint-based Scheduler along with each piece of content proved to be a significant deterrent to content developers that were not skilled application developers, and this deterrent reduced their willingness to develop content for our system. In reality, the Schedulers for day-to-day content and for some of the research-related content items were provided by members of our research group.

Complexity Involved in Managing the Life-Cycle of a Constraint-Based Scheduler

Constraint-based Schedulers are processes that wait for certain events to occur that cause the constraints of the content to be fulfilled. As a result, Schedulers undergo a life-cycle that has to be managed. This life-cycle may, for example, involve the deployment of Schedulers onto hosts within the e-Campus network, as well as the execution and monitoring of Schedulers and handling of faults that may occur during execution. Developers are therefore not only required to program the logic for scheduling their applications, but also for monitoring and managing that logic, once deployed into the e-Campus display network, adding to the complexity of the task.

Language Dependence of the Low-Level API and the Underlying Transport Protocol

As described in chapter 5 the low-level API was implemented in Python and as such schedulers using the API had to be developed in Python as well. Developers' unfamiliarity with this language represented another stumbling block for people intending to develop applications and content for e-Campus.

Even when working at the protocol level, the choice of programming languages was in fact limited. Elvin libraries necessary for sending and receiving Elvin events that served as transport for low-level API protocol messages were only available for a small set of programming languages, namely Java, C/C++ and Python.

Not surprisingly content providers were reluctant to familiarise themselves with a specific programming language just to get their piece of content shown on the display network.

Lack of Support for Personalised Access Control and Accounting

Our software infrastructure and its scheduling API are designed to operate on a secure virtual network and thus do not contain any explicit security features. However, during our trials it became clear that there were several problems with this security model. Firstly, while access to the software infrastructure (and therefore the displays) would be regulated, it would be impossible for us to determine who scheduled, for example, an offensive piece of content without logging large quantities of network traffic. Although we did not foresee offending content appearing in the context of research-related content, we did consider the lack of accountability to become more of a problem once the display network started to be used more regularly by a wider range of content providers, including student societies.

Secondly, the lack of authentication information at the protocol level of the software infrastructure meant that we were unable implement personalised access policies. These policies might, for example, restrict the maximum priority level that individuals are able to assign to their content. For example, we would have liked to assign higher maximum priority levels to researchers than to providers of day-to-day content. By doing this we would have been able to guarantee that content related to experiments was always able to preempt day-to-day background content. By assigning even higher maximum priority levels to members of the University administration we would have

been able to guarantee that important announcements from the administration take precedence over any other content. Moreover, the lack of authentication information also meant that we were unable to restrict certain users to certain displays, and were therefore unable to exclusively reserve certain displays for experimentation.

Proposed Solution

In the following chapter we will present the design, implementation and evaluation of a high-level, constraint-based scheduling service and associated API to address the problems identified in this chapter.

6.8 Summary

In this chapter we have described the implementations status of our software infrastructure and API and provided a detailed evaluation – both qualitative and quantitative – of the implemented system. The qualitative evaluation was based on a number of trials conducted with experimental content and on experiences made with day-to-day content. The quantitative aspects of our software infrastructure were demonstrated using performance measurements and theoretical calculations.

While the low-level distributed systems infrastructure and API were generally found to fulfil the requirements outlined in chapter 3, the in-situ evaluation of the deployed system using "real-world" experimental applications and content has exposed a mismatch between the requirements of experimenters and the capabilities of the low-level API, namely the lack of an easy-to-use high-level scheduling service and API that enables experimenters to schedule content based on constraints, such as time, priority, display sets and presence of Bluetooth devices in the vicinity of displays. The following chapter will describe the design, implementation and evaluation of a set of extensions designed to eliminate these shortcomings.

Chapter 7

High-Level API and Scheduling Service

7.1 Introduction

The previous chapter has provided an overview of the implementation status and a detailed evaluation – both quantitative and qualitative – of our implemented software infrastructure. The in-situ evaluation of the deployed software infrastructure using "real-world" experimental applications and content exposed a mismatch between the requirements of experimenters and the capabilities of our scheduling API. In this chapter we discuss the implications of these shortcomings and present the design, implementation and evaluation of a (high-level) scheduling service and accompanying API aimed at addressing these shortcomings.

7.2 Discussion of Requirements

The problems described in section 6.7 that application developers encountered when using the low-level API (the complexity of designing and implementing a constraint-based scheduler, the complexity of managing the life-cycle of a constraint-based scheduler, the repetition of effort, language dependence of the low-level API and the underlying transport protocol, and the lack

of support for personalised access control and accounting) can be transformed into a number of requirements for the design of a high-level API that we discuss in this section.

7.2.1 Provision of a Reusable API for Constraint-Based Scheduling

During our trials (described in chapter 6) with our scheduling infrastructure and API we have observed that constraint-based scheduling is a common requirement for a significant portion of the content developed for public display networks. This observation suggests that this functionality should be factored out into an API or library that can be re-used by developers. By providing such an API we are therefore able to reduce the *repetition of effort* we identified as one of the major problems in our analysis of e-Campus applications.

Moreover, by providing such a common API, we are also able to address the *complexity involved in designing and implementing constraint-based scheduling functionality* by freeing application developers from having to implement this functionality themselves.

We therefore propose to *provide a reusable API for constraint-based scheduling* (high-level design proposition HL-DP1). We call this API the *high-level (or constraint-based) scheduling API*. To better distinguish the high-level scheduling API from the scheduling API that we described in chapters 4 and 5 of this thesis we call the API presented in chapters 4 and 5 the *low-level scheduling API*.

The types of constraints supported by such a constraint-based scheduling API should at least cover the constraints that we encountered in the context of the trial applications:

- *date and time ranges* during which content should be considered for scheduling, for example providing users with the ability to have content considered for scheduling on the first day of each month from 14:00h to 16:00h.
- the *set of displays* that content should be scheduled on.
- whether showings on these displays should be *synchronised or not*.
- the *priority* of content.

7.2.2 Provision of a Centrally Hosted Scheduling Service

To address the *complexity of managing the life-cycle of a constraint-based scheduler* the constraint-based scheduling API should be usable without requiring experimenters to keep processes active over extended periods of time in order to have their content scheduled. We therefore propose to design the constraint-based scheduling API to be useable by client processes that are short-lived and that terminate long before the content they have requested to be scheduled is actually displayed. Moreover, we propose to allow the computers that are used to execute these short-lived client processes to be disconnected from the display network as soon as the processes have terminated without affecting the execution of the constraint-based schedules that these processes have requested. For example, a process could use the constraint-based scheduling API to request a particular piece of content to be displayed in a week's time. We expect the content to be shown even though the requesting process has terminated and the machine that was used to execute the process has been disconnected from the network.

As a result, the logic for showing content based on the provided constraints cannot be located in the API itself or on the client machine. Instead, this functionality has to be provided by a persistent process or service in the display network.

We therefore propose to *provide a centrally hosted scheduling service* (high-level design proposition HL-DP2) that processes requests that are issued via the constraint-based scheduling API.

7.2.3 HTTP-Based Protocol

One of the aims of providing the constraint-based scheduling API is to reduce the complexity involved in displaying content on public displays in the display network. Access to the constraint-based scheduling API and its use should therefore be made as simple as possible. We therefore propose that access to the constraint-based API should be possible using any machine that is connected to the display network. For example, content providers should not have to log into dedicated machines to access the constraint-based scheduling API. Instead they should be able to use their own computers to request content to be shown on the public display network. This means that the constraint-based scheduling API has to be provided in the form of a distributed API that uses the network to communicate with the centralised scheduling service.

As outlined before, our low-level scheduling API was *language dependent* (it was only available for the Python programming language), which decreased the likelihood that content providers had the necessary skills for scheduling content using our low-level scheduling API. Two main factors for this language dependence can be identified: the language dependence of the underlying transport protocol, and the relatively high level of complexity that was contained in the client-side API. The first issue was caused by the restricted availability of Elvin client SDKs. SDKs were only available for C/C++, Java and Python. However, even if client SDKs had been available for a larger range of programming languages, porting the low-level API would have required an investment of significant effort to create each of these ports. After all, low-level API instances not only implement support for the request/response-based protocol, but also act as coordinator for transactions and implement most of the logic behind group operations.

As we have outlined above, we plan to outsource most of the complexity of constraint-based scheduling into an external scheduling service. We therefore do not expect the complexity of the high-level scheduling API to be a major factor in preventing the porting of the high-level scheduling API to different programming languages.

To ensure portability of the high-level scheduling API we therefore plan to make the protocol that is used to engineer the communication between the high-level scheduling API and the high-level scheduling service accessible from a wide range of programming languages. This can, for example, be achieved by engineering this communication using a TCP-based [Pos81] or UDP-based [Pos80] protocol. Client libraries for communicating over TCP or UDP are available for a large number of programming languages. Using a TCP-based or UDP-based protocol would therefore enable the implementation of client APIs to the constraint-based scheduling service for these programming languages.

Another option for constructing language-independent protocols and APIs is exemplified by the way that many popular commercial web-based services (such as Google Calendar [Goo08b] , Flickr [Fli08b] and Youtube [You08b]) provide programmers with access to these services. These services offer access using HTTP-based protocols that are typically communicated using HTTP GET or HTTP POST requests. Arguments are transported using HTTP request parameters, leading to the following overall syntax for operation invocations:

```
http://host/operation?argument_1=value_1&...&argument_n=value_n
```

Additionally, XML [BPSM+06] notation may be used to encode individual arguments if these represent complex data types. Moreover, XML is typically used to encode any responses that are returned to the client. The services often do not provide any programming-language-specific client libraries. Instead, documentation is provided that explains the intended format of the HTTP requests, and the structure of responses, along with examples that show how to assemble request and un-marshal responses using various programming languages.

The provision of such HTTP-based APIs provides a number of benefits. Firstly, many users are already experienced in the use of such HTTP-based APIs. Secondly, HTTP-based requests are simple to assemble and therefore do not require any additional support on the client machine: the ability to manipulate strings is usually sufficient to assemble requests, and the ability to issue HTTP requests is sufficient to invoke API operations.

We therefore propose to *expose the operations provided by the constraint-based scheduling service to users in the form of an HTTP-based protocol* (high-level design proposition HL-DP3).

7.2.4 Individually Authenticated Operations

To be able to support the introduction of features such as personalised access control and accounting, the constraint-based scheduling service needs to be able to associate individual invocations of the constraint-based scheduling API with individual users.

We do not wish to impose any additional requirements on client machines, such as support for HTTP Cookies [KM00]. Moreover, the use of authentication based on the IP address of client machines is not feasible, since we would like to enable the use of the constraint-based scheduling API on servers that are often used by multiple users simultaneously.

We therefore propose to provide support for associating individual API invocations with individual users by *submitting authentication information as part of every operation invocation* (high-level design proposition HL-DP4).

7.3 Design

7.3.1 Overview

Constraint-based scheduling functionality is provided by a centralised scheduling service. We call this service the *high-level scheduling service*. The functionality provided by the high-level scheduling service is exposed to content providers in the form of an HTTP-based API that we call the *high-level (or constraint-based) scheduling API*.

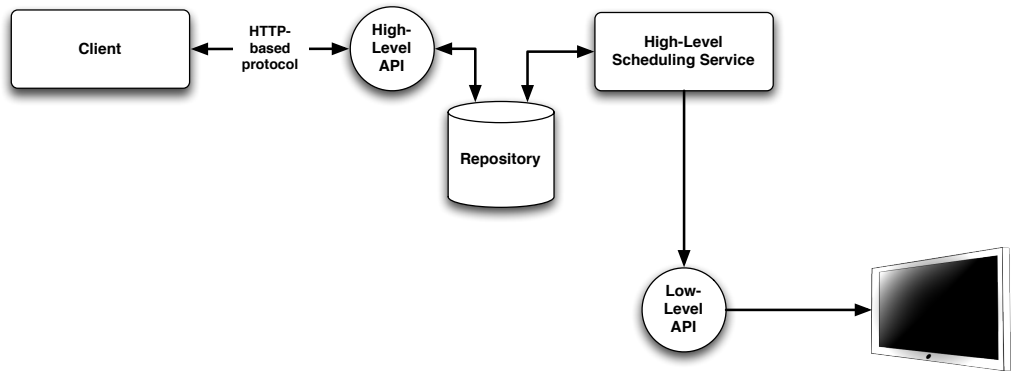


Figure 7.1: Overview of the High-Level API and the High-Level Scheduling Service.

The high-level API enables experimenters to request content to be scheduled based on a set of supported constraints. A central data repository is used to persistently store incoming scheduling requests, from where they can be retrieved and processed by the high-level scheduling service.

Experimenters request content to be scheduled by using the high-level API to create, modify and delete two types of entries in the repository: *Playlists* and *Requests*. A Playlist represents an orchestration of a set of content items on a set of Displays. A Request defines, for example, the dates and times during which the high-level scheduling service should consider to play the Playlist. The high-level scheduling service uses the low-level scheduling API to display Playlists. At any point in time, multiple Playlists may be in a state in which they can be scheduled (i.e. the constraints associated with their Requests can be fulfilled) and compete for airtime on the same set of Displays. The high-level scheduling service employs policies, such as the priority of Requests, to arbitrate between competing Requests.

An overview of the architecture of the high-level scheduling service and API is provided in figure 7.1.

In the following sections we provide an overview of the high-level scheduling API and its operations. Please refer to appendix A for detailed descriptions of the attributes of Playlists and Requests, and for detailed descriptions of the arguments and return arguments of the operations comprising the high-level API.

7.3.2 Playlists and Requests

Playlists

Playlists define a repeatable orchestration of content items on a set of Displays. For example, they allow experimenters to define arrangements, such as "show the content item identified by the URL `http://a.host/content1` on `display1` for 2 minutes, then show the content item identified by the URL `http://a.host/content2` on both `display1` and `display2` for 5 minutes, while at the same time showing content item `http://a.host/content3` on one of `display3` and `display4`". Playlists define the temporal composition of content items on public displays.

Playlists are identified in the repository by a unique Playlist Identifier (`playlist_id`). The actual specification of the composition of content items is defined by a list of `content` entries that are associated with the playlist. Besides the URL that can be used to retrieve the media item that is associated with the content entry, each entry specifies the Displays that the content is to be shown on, and the times that the content is to be made visible and not visible. Times are specified in seconds relative to the start of the Playlist. The list of Displays that the content item should be shown on are specified as a set of Display Identifiers. A separate attribute (`sync`) controls how this set of Display Identifiers is interpreted. Possible values are `one`, `many` and `all`, specifying that the content item is to be shown on one of the specified Displays, on as many as possible of the specified Displays, or on all of the specified Displays. In the latter case, a failure to make the content item visible on one of the specified Displays will cause the play-out of the Playlist to be aborted.

Requests

Requests specify the constraints that determine when Playlists should be shown. Each Request is associated with exactly one Playlist whose identifier is referenced in a `playlist_id` attribute.

However, each Playlist may be associated with multiple Requests.

Each Request specifies a set of constraints that all have to be fulfilled before the Request is considered for scheduling by the high-level scheduling service. We acknowledge that it is very difficult to provide support for every possible type of constraint that experimenters might wish to use to determine when content is shown. We provide support for the constraints that experimenters used in the trials that we described in chapter 6 to schedule their content. Experimenters are able to define date and time ranges that specify that content should be considered for scheduling during these periods. Using the `time_conditions` attribute an experimenter is, for example, able to specify that a Playlist should be shown between 14:00h and 15:00h, and between 16:00h and 17:30h on 02 March 2009, and between 20:00h and 21:00h on 04 March 2009. Which Displays a Playlist is shown on is influenced by the `displays` and `sync` parameters of the Playlist. Besides the ability to specify date and time ranges, Requests also allow experimenters to define the priority of the content items in the associated Playlist. The `priority` attribute of Requests is semantically equivalent to the `priority` parameter of the `Transition` operation exposed by our low-level scheduling API. The attribute determines the priority that is used to make the content contained in the Playlist visible.

Besides support for the types of constraints that were used during the trials described in chapter 6, i.e. date and time constraints and priority constraints, our design also includes support for a number of additional constraints that we considered as useful for scheduling content. Three of these constraints influence whether the Playlist should be shown repeatedly when its constraints can still be fulfilled. The attribute `repeat_spacing` contains the number of seconds that the high-level scheduling service should wait after the play-out of the Playlist has finished before considering whether to show the Playlist again on the same Display. A value of 0 indicates that play-out should be repeated without spacing. Please note that the `repeat_spacing` is a property of the Request entry and determines whether the high-level scheduling service considers the Request (and therefore the Playlist associated with the Request) as being in a state in which it can be shown on the displays. If a Playlist is associated with more than one Request, the Playlist may appear more frequently as the spacing is measured on a per-request basis.

The attributes `target_total_showings` and `target_showings_per_display` limit the number of times that the Playlist associated with the Request is shown in total in the display network and on a per-Display basis. A value of -1 indicates the absence of any limits. As in the case of

the `repeat_spacing` constraint, counters are maintained on a per-Request basis, i.e. a Playlist may overall exceed the targeted numbers of showings set by a Request if the Playlist is associated with more than one Request.

We also provide the ability to restrict the play-out of Playlists to times when certain Bluetooth devices are present in the vicinity of certain Displays. The `bt_presence_filter` constraint consists of a set of $([device_1, \ldots, device_n], display_x)$ tuples, specifying that at least one of the devices listed in each tuple should be present at the Display referenced by that tuple in order to fulfil the constraint. For example, a Request with the `bt_presence_filter` constraint [([“11:11:11:11:11:11”], “display-01”), ([“22:22:22:22:22:22”, “33:33:33:33:33:33”], “display-02”)] specifies that the Playlist associated with the Request should only be considered for scheduling if the Bluetooth device with the hardware address “11:11:11:11:11:11” is in the vicinity of Display “display-01” and if at the same time either the Bluetooth device with the hardware address “22:22:22:22:22:22” or the device with address “33:33:33:33:33:33” are in the vicinity of “display-02”. If the desired semantic is to have a Playlist considered for scheduling if one of a set of Bluetooth devices is present at “display-01” *or* if one of a set of Bluetooth devices is present at “display-02”, this can be achieved by associating two Requests with the Playlists. In this case one of these Requests contains the `bt_presence_filter` constraint for “display-01”, while the other Request contains the `bt_presence_filter` constraint for “display-02”.

Ownership Model

Requests and Playlists are “owned” by the users who created them and can only be modified and deleted by those users. Read-only access to both Request entries and Playlist entries is granted to all users, i.e. users are able to use the `RetrievePlaylist` and `RetrieveRequest` operations to obtain the specifications of Playlists and Requests, even if these are not owned by the requesting users. Moreover, users may use the search operations `ListPlaylists` and `ListRequests` to search for Playlists and Requests matching certain criteria. Both operations are capable of finding entries that are not owned by the requesting users. By providing universal read access we enable users to, for example, obtain an overview over other entries in the repositories that have similar constraints and are therefore likely to be played at the same time as their own content.

Playlists can only be associated with Requests if these Playlists are owned by the same users

that own the Requests. By imposing this restriction we avoid unwanted side effects when users delete Playlists that are still associated with Requests. Since each Request is associated with exactly one Playlist, consistency rules in the repository require any Requests to be deleted from the repository if they are associated with Playlists at the time these Playlists are deleted. By ensuring that Requests and associated Playlists are always owned by the same users we aim to ensure that the deletion of a Playlist only has side-effects on Request entries that are owned by the same user.

7.3.3 API Operations for Authentication

Access to high-level API operations is restricted to registered users. Authentication is performed as a two-stage process: users initially authenticate using the `Authenticate` operation by providing their email address and a password. If this authentication step is successful, an authentication token is returned that is subsequently passed as an argument to high-level API operations. Authentication tokens expire after a fixed period of inactivity.

7.3.4 High-Level API Operations for the Management of Playlists

The high-level API provides a set of operations allowing users of the API to manage Playlists.

`CreatePlaylist` creates a new Playlist entry in the repository according to the user's specifications. The specification of the Playlist is provided using the `playlist_spec` argument, and this specification conforms to the format described in appendix A, section A.1. On success, the unique identifier of the newly created Playlist is returned.

`RetrievePlaylist` accepts a Playlist Identifier as argument and retrieves the specification of the corresponding Playlist.

`UpdatePlaylist` allows users to modify Playlists that they have previously created using `CreatePlaylist`. The targeted Playlist is identified using the `playlist_id` field. The argument `playlist_spec` contains a full description of the modified Playlist.

`DeletePlaylist` is used to remove an existing Playlist entry from the high-level scheduling repository. The Playlist that is to be removed is referenced using its Playlist Identifier. If the

Playlist is currently associated with a Request entry, this Request entry is removed automatically as well.

`ClonePlaylist` enables developers to re-use existing playlists in new requests. The operation returns the identifier of the new playlist that represents an exact copy of the cloned playlist, i.e. URLs, start and end times, associated displays, and coordination specification match those of the old playlist.

`ListPlaylists` allows users to search for Playlists according to a specified set of criteria. Search criteria may be specified as a combination of urls that should be contained in the Playlist, identifiers of Displays that are targeted by the Playlist, and whether or not the search should be limited to Playlists that are owned by the user issuing the `ListPlaylists` request. `ListPlaylists` returns a list of identifiers of Playlists matching the search criteria. `ListPlaylists` may also be invoked without specifying any search criteria, in which case the identifiers of all Playlists that are known to the high-level scheduling service are retrieved.

7.3.5 High-Level API Operations for Managing Requests

Similar to the operations affecting Playlist entries in the repository, the high-level API provides operations for creating, retrieving, modifying, and deleting Requests.

`CreateRequest` allows users to create new request entries in the high-level scheduling repository. Requests are specified using the attributes described in appendix A, section A.2. If the operation is completed successfully, the identifier of the newly created Request entry is returned. This identifier can be used to reference the newly created requests in subsequent high-level API operations.

`RetrieveRequest` allows users to retrieve the specification of Request entries that exist in the repository. Requests that are to be retrieved are identified by their unique Request Identifiers (`request_id`). On successful completion, the operation returns the specification of the targeted Request comprising the attributes described in appendix A, section A.2.

`UpdateRequest` can be used to modify existing Request entries. The actual modifications are communicated by proving a new specification (`request_spec`) for the Request, causing the old

specification to be replaced by the new specification.

`DeleteRequest` is used to remove an existing Request entry from the high-level scheduling repository. The Request that is to be removed is referenced using its Request Identifier.

`ListRequests` provides users with the means to find Requests in the repository that match the provided search criteria. The search criteria operate on the constraints associated with Request entries. If multiple types of criteria are provided, the `ListRequest` invocation is processed as if these individual criteria were combined using logical "and" operations. On success, a list of Request Identifiers belonging to matching Request entries is returned.

7.3.6 Enforcement of Personalised Priority Limits

The high-level API enforces priority limits on a per-user basis. This means that each user is assigned a maximum priority level. Requests issued by that user are not able to exceed this priority level. If a user attempts to use the high-level scheduling API (using `CreateRequest` or `UpdateRequest` operations) to create a Request entry in the repository with a higher priority level than permitted, the high-level API blocks these operations and returns an error code. Moreover, the high-level scheduling service also compares users' maximum priority levels against the priority levels specified as part of Requests. This ensures that any changes that are made to a user's maximum priority level after the user has already issued Requests are taken into account. If a priority level of an existing Request is found to exceed a user's maximum priority level, the high-level scheduling service automatically reduces the Request's level accordingly. The enforcement of individual priority levels is an example for the support of personalised access control policies in the high-level scheduling API and service. The lack of such support was one of the shortcomings of the low-level scheduling API that we identified and discussed in chapter 6.

7.3.7 The High-Level Scheduling Service

The high-level scheduling service is centrally hosted on one of the servers in the public display network. The service continually evaluates the constraints of Request entries and Playlist entries in the high-level scheduling repository. When the constraints of a Request and its associated Playlist are satisfied, the high-level scheduling service uses the low-level API to show the Playlist's

content on the specified Displays.

In cases where at any point in time multiple Requests exist whose constraints can be fulfilled and whose Playlists require the use of the same set of Displays, additional policies are used to select which Request to schedule (i.e. which Playlist to show). A Display is considered to be "required" by a Playlist if one of the following conditions is fulfilled:

- the Display is the only Display that is targeted by a content item in the Playlist.
- the Display is part of the list of targeted Displays of a content item that has the `sync` attribute set to "all of the specified Displays" (`all`).
- of the Displays targeted by a specific content item this is the only Display that is currently available, i.e. that either does not currently show any content or shows content but is preempt-able.

Higher-priority Requests are always given precedence over lower-priority Requests. Within the same priority level, conflicts are resolved by giving precedence to those Requests that have been least recently selected for play-out, i.e. Requests that have not been shown for a longer period of time are given precedence over Requests that have been shown more recently. Finally, Requests that have been shown least frequently are given precedence over Requests that have been shown more often.

Once play-out of a Playlist has started, it is typically not interrupted by the high-level scheduling service. Interruptions may, however, occur in one of the following cases:

- the constraints of a higher-priority Request that requires one or more of the Displays that are also required by the Playlist that is currently being played can be fulfilled. In this case play-out of the Playlist is interrupted, and the play-out of the Playlist of the higher-priority Request is started.
- one of the required Displays is pre-empted using the low-level scheduling API.
- an error occurs during the play-out of the Playlist that the high-level scheduling service is unable to recover from.

7.4 Implementation

In this section we provide an overview of the implementation of the high-level scheduling service and the high-level scheduling API.

7.4.1 Implementation of the High-Level Scheduling API

In line with high-level design proposition HL-DP3 the high-level scheduling API is engineered using an HTTP-based protocol. The API is implemented in Python and served as a CGI script by an Apache Web Server [The08a] in the display network, making the API accessible using the HTTP protocol. A MySQL [MyS08] database is used as a repository for high-level scheduling Requests and Playlists. Both the high-level API and the high-level scheduling service interface with this database. The operations exposed by the high-level scheduling API for creating, modifying, deleting and retrieving Playlists and Requests map almost directly onto SQL operations for modifying the underlying representations of these data structures in the database back-end.

HTTP requests corresponding to high-level API operations are constructed by appending the operation name to the base URL of the high-level scheduling API. The arguments of the operations are passed along as request parameters. Parameter values are formatted using the JavaScript Object Notation (JSON) [JSO08], allowing for the transmission of complex data types of parameter values. The following is an example of an `Authenticate` request:

```
http://host/hl_api.py/Authenticate?email="user@host.com"&password="thepw"
```

The argument name (e.g. `email`) is used as the parameter name, and the argument value is passed along as parameter value. Additionally, the request parameters have to be percent-encoded (see RFC 3986 [BLFM05]) to escape any reserved characters before submitting the request. The `Authenticate` request above would typically be percent-encoded as:

```
http://host/hl_api.py/Authenticate?email=%22user@host.com%22&password=%22thepw%22
```

However, most modern web browsers automatically percent-encode URLs that are typed directly into the location bar, enabling high-level API operations to be invoked without requiring

users to explicitly percent-encode their requests.

The following examples shows the structure of a Playlist specification when provided in JSON syntax:

```
{
    "playlist_id" : 815,
    "content" : [
        {
            "url" : "file:///a_video.avi",
            "rel_start" : 0,
            "rel_end" : 20,
            "displays" : [ "display-01", "display-02", "display-03" ],
            "sync" : "all"
        },
        {
            "url" : "file:///another_video.avi",
            "rel_start" : 20,
            "rel_end" : 75,
            "displays" : [ "display-01", "display-03" ],
            "sync" : "many"
        }
    ]
}
```

The Playlist in the example consists of two content items. The first content item (a video) is to be shown for 20 seconds on all of the specified Displays (display-01, display-02 and display-03). Following that content item a second video is to be shown on as many of the specified displays (display-01 and display-03) as possible. The second content item is to be shown for 55 seconds, after which the Playlist ends. The current status of our implementation of the high-level API and scheduling service limits each content item to a single display (i.e. `displays` attributes can only contain a single Display Identifier). However, we expect to eliminate this restriction in the next version of the implementation.

Please note that in the example, line-breaks and additional white-spaces have been added to better demonstrate the structure of the Playlist specification. Although the example still represents valid JSON syntax, these line breaks and white-spaces would typically be removed when using the specification as request argument, leading to the following representation:

```
http://...?playlist_spec={"playlist_id":815,"content":[{"url":...},{...}]}&...
```

The following example shows the JSON representation of a Request specification:

```
{
    "request_id" : 69,
    "playlist_id" : 815,
    "time_conditions" : [
        [
            "2008-09-30 08:00:00",
            "2008-09-30 10:00:00"
        ],
        [
            "2008-10-01 20:00:00",
            "2008-10-02 02:00:00"
        ]
    ],
    "priority" : 5,
    "repeat_spacing" : 300,
    "target_total_showings" : 20,
    "target_showings_per_display" : -1,
    "bt_presence_filter" : [
        {
            "devices" : [
                "00:10:20:30:40:50",
                "01:11:21:31:41:51"
            ],
            "display_id" : "display-01"
        },
        {
            "devices" : [
                "02:12:22:32:42:52"
            ],
            "display_id" : "display-02"
        }
    ]
}
```

The Request specifies that the high-level scheduling service should attempt to show the Playlist defined above on the 30th of September 2008 between 08:00h and 10:00h, and then again between 20:00h on the 1st of October 2008 and 02:00h on the 2nd of October 2008. Once the Request has been selected and the Playlist played, the high-level scheduling service should wait for 5 minutes before showing the Playlist again. The Playlist associated with the Request should be played for a total of no more than 20 times. The Request does not limit the number of play-outs on a per-Display basis. The Request is assigned a priority level of 5. Additionally, the Playlist associated with the Request should only be played if one of the Bluetooth devices with the hardware addresses 00:10:20:30:40:50 and 01:11:21:31:41:51 are present in the vicinity of Displays display-01, and if the device with the hardware address 02:12:22:32:42:52 is simultaneously present at Display display-02.

Responses from the high-level scheduling API are returned in the form of JSON objects. Each

operation returns a single object containing all the result parameters. Individual result parameters are represented by (name,value) pairs in the returned JSON object. The following example shows a JSON object that was returned in response to an `Authenticate` invocation:

```
{
    "error" :  0,
    "auth_token" : "803ec4071d5fe12dbfdf7efec06cc743"
}
```

Examples of the JSON representations of the remaining request parameters are provided in tables 7.1 and 7.2.

7.4.2 Implementation of the High-Level Scheduling Service

We have implemented a simple high-level scheduling service as a single Python process running on one of the server machines in the display network. The evaluation of constraints is performed using SQL queries on the database back-end and therefore benefits from the database's potential to efficiently query and combine data. Where constraints are based on information that is only available in the form of events, as is for example the case with sightings of Bluetooth devices in the vicinity of displays, this event-based information is translated into state information and entered into the database tables by an external Python process (code-named "Collector").

For each available Display the high-level scheduling service creates a list of Requests whose constraints can be fulfilled. The Requests in each list are ranked using the policies outlined in section 7.3.7. If the play-out of the top-ranked Request for a particular Display d does not require any additional Displays, it is immediately selected for play-out, and the scheduling service starts to evaluate the ranked Requests of the next Display. Otherwise the high-level scheduling service assesses whether a play-out of the Request on these required Displays would be compatible with the policies described in section 7.3.7. If this is the case, the play-out of the associated Playlist is started on all required Displays. If this is not the case, the Request is skipped and the high-level scheduling service starts to evaluate the next Request in the list of ranked Requests for Display d. Once the Requests on all Displays have been evaluated, the whole process is repeated.

Parameter Name	Type	Example
`with_urls`	array	`[ "http://www.lancs.ac.uk", "file:///home/user/content.mov", "file:///image.jpeg" ]`
`with_display_conditions`	array	`[ "display-01", "display-10", "display-22" ]`
`my_playlists_only`	boolean	`true`
`playlist_ids`	array	`[ 1, 2, 32 ]`
`with_playlist_ids`	array	`[ 1, 2, 32 ]`
`with_time_conditions`	array	`[ [ "2008-09-30 08:00:00", "2008-09-30 10:00:00" ], [ "2008-10-01 20:00:00", "2008-10-02 02:00:00" ] ]`
`with_priorities`	array	`[ 3, 4, 9 ]`

Table 7.1: JSON-formatting of request parameters.

Parameter Name	Type	Example
`with_bt_presence_filter_devices`	array	`[ "00:10:20:30:40:50", "01:11:21:31:41:51" ]`
`with_bt_presence_filter_displays`	array	`[ "display-01", "display-10", "display-22" ]`
`my_requests_only`	boolean	`true`
`request_ids`	array	`[ 815, 123, 69 ]`

Table 7.2: JSON-formatting of request parameters (continued).

Playlists are processed by the high-level scheduling service using separate threads. In large public display networks, the load on the high-level scheduling service can be eased by employing additional replicated instances of the high-level scheduling service. These replicated instances of the high-level scheduling services use the transactional features of the database back-end for inter-process coordination.

The implementation described above provides simple scheduling capabilities. However, more sophisticated scheduling services based on our high-level API could be implemented and deployed as required.

7.5 Evaluation

Being a relatively recent addition to the software infrastructure, the high-level scheduling service and API have not enjoyed the level of use that the low-level scheduling API has seen, for example in the context of student projects. Nevertheless we are able to report on the experiences we have of using the high-level API and scheduling service for showing a number of day-to-day content

items and for underpinning two experimental applications (the *poster system* and the *channels system*).

7.5.1 Day-to-Day Content

Day-to-day content items have mainly involved announcements for special events, such as lectures and workshops, and a number of videos advertising the activities of student societies. The Playlists and Requests for some of these content items were created using an experimental Web front-end that was implemented in HTML and Javascript and allowed us to enter the constraints into a web form. The Web front-end used Javascript code to assemble and emit the necessary HTTP requests to the high-level scheduling API, based on the entered form values, and for parsing the responses received from the high-level scheduling API. Other content items were scheduled with the help of a Python-based front-end that allowed users to specify Playlists and Requests in a file. The Python script was responsible for JSON-encoding these representations, for invoking high-level API operations by assembling and emitting HTTP requests, and for receiving and parsing responses. Finally, we directly used the location bar in Web browsers to schedule some of the content items. This involved creating JSON-formatted Playlist specifications and Request specifications and assembling `CreatePlaylist` and `CreateRequest` HTTP requests. While this method was by far the quickest and most straightforward way to invoke API operations, we found that the other two methods also did not require any extensive programming skills to assemble the required HTTP requests or to parse the received responses.

7.5.2 The Poster System

The poster system is an ongoing experiment investigating the creation of content authoring and scheduling tools that are simple to use and do not require users to have any prior experience with authoring tools.

The poster system allows students and staff at Lancaster University to create electronic posters for use on the e-Campus public display network. To create a poster, users use a Web-based application to carry out a two-step process. In the first step users design their posters based on templates by modifying a small set of parameters, such as poster backgrounds, text for poster

headings and bodies, and font sizes and styles. In the second step users select the public displays they would like their poster to appear on, and date and time ranges during which the poster should be shown on these displays. A screenshot of this scheduling interface is shown in figure 7.2. Moreover, users are able to direct posters to specific individuals, causing posters to appear only if these individuals are in vicinity of the targeted displays. Targeted individuals are required to register the hardware address of their Bluetooth-enabled mobile phones with the poster system.

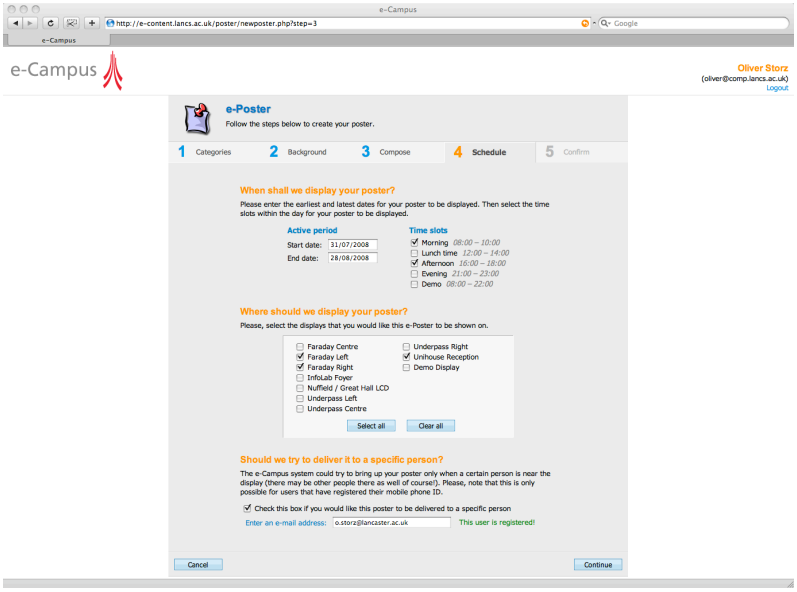


Figure 7.2: Screenshot showing the scheduling interface of the posters system.

Figure 7.3 shows an overview of the architecture of the poster system. A set of PHP scripts is used to generate the Web-based user interface of the poster system. Layout specifications for user-generated posters are stored in a MySQL database. The scheduling constraints specified by users are translated into specifications of Playlists and Requests and communicated to the high-level scheduling service via the high-level scheduling API. Each poster is represented by a separate Playlist which contains only a single content item: the poster. Each poster is shown for a fixed duration (45 seconds at the time of writing this thesis). The list of targeted displays that users provide using the Web-based UI is translated into `display_conditions`. Date and time ranges specified by users are turned into `time_conditions` constraints in the Request specifications. If posters are targeted at specific individuals, this is expressed in the form of `bluetooth_presence_filters` in the Requests' specifications.

The poster system was designed by Prof. Nigel Davies and Dr. Christos Efstratiou and implemented by Dr. Christos Efstratiou. It has so far been tested on a number of public displays

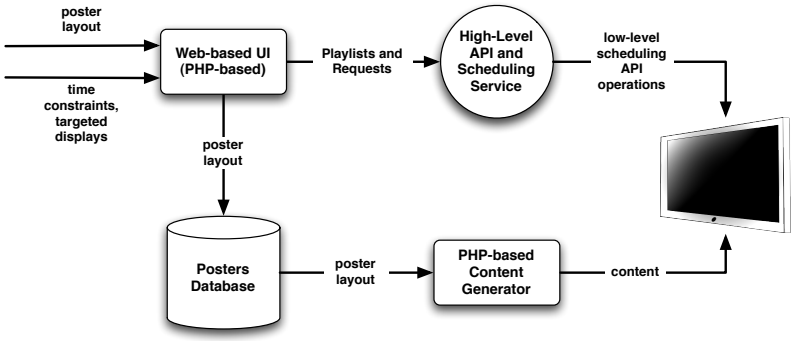


Figure 7.3: Overview of the architecture of the posters system.

in the e-Campus display network. We are currently awaiting the results of a legal review of the poster system's proposed terms and conditions and expect the system to go live once this review has been concluded successfully.

7.5.3 The Channels System

The channels system is aimed at investigating a means for allowing content providers to make their content available in the public display network, and for allowing display owners to select which content they would like to see on the public displays they manage.

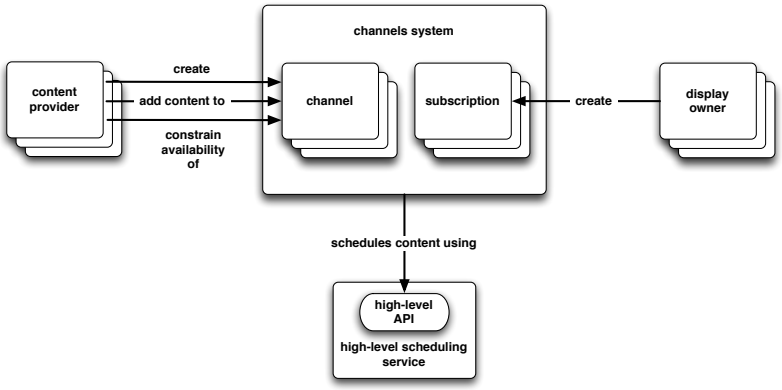


Figure 7.4: General model employed by the channels system.

Figure 7.4 provides an overview of the general model employed by the channels system. Content providers use a Web-based interface to create so-called *channels*. Channels represent a collection of content items and may be themed. Each channel is represented by a folder on a networked file system that is made available to content providers using Samba [Sam08]. Once a channel has been created, content providers are able to add content to the channel by using their computer to

drag and drop media files into the shared folder representing the channel. Content providers are able to limit the availability of their channels to certain public displays, and to certain dates and times.

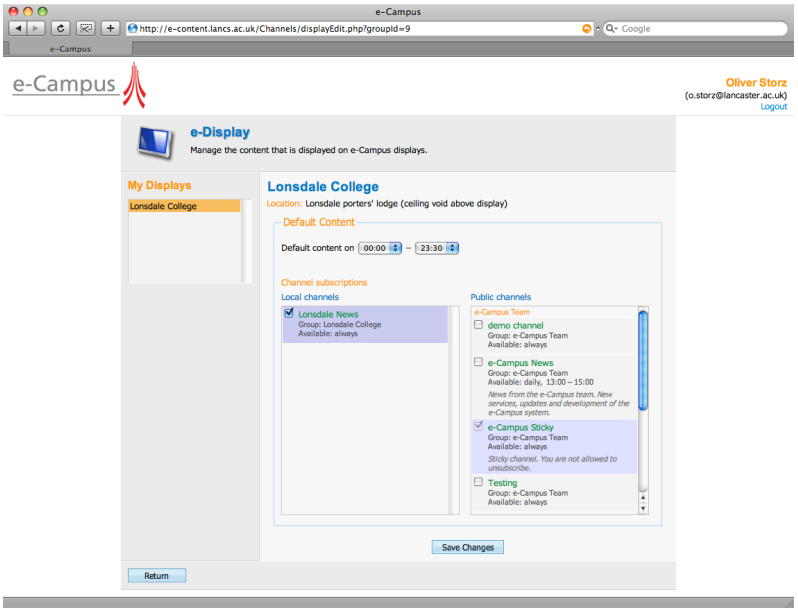


Figure 7.5: Screenshot showing the UI that display owners use to subscribe to channels.

Display owners use the Web interface (see screenshot in figure 7.5) to subscribe to channels, causing the displays to show the content contained in these channels. If a display has subscribed to multiple channels, content from these channels is shown on the display in an interleaved fashion. While currently content from the different channels is played with equal ratios, we plan to provide display owners with the possibility to specify their own ratios in the future.

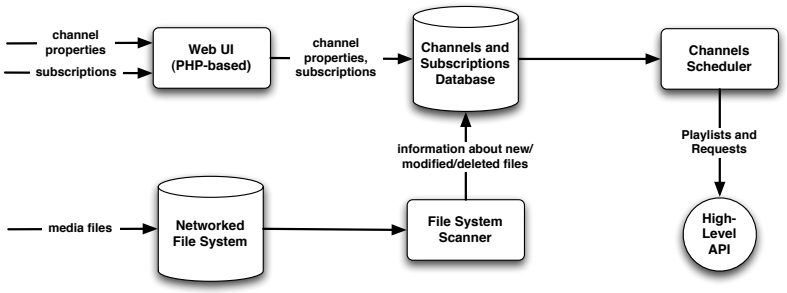


Figure 7.6: Overview of the architecture of the channels system.

Figure 7.6 shows an overview of the architecture of the channels system. The Web-based parts of the channels system are engineered as a set of PHP-scripts. Information about channels and subscriptions to channels are stored in a MySQL database. A Python process is used to periodically

scan the folder structure representing the channels for changes. The state of the file system is recorded in the MySQL database. A second Python process is responsible for using the high-level scheduling API to create Playlists and Requests for the content items in the channels based on the properties that content providers have specified for their channels and the subscriptions provided by display owners. Separate Playlists and Requests are created for each combination of a content item and a Display, provided that a subscription exists on that Display for the channel that the content item is part of. The duration of continuous media items, such as videos, is determined by the channels system, and `rel_start` and `rel_end` times are set accordingly to allow full playback of the media item. Static media items, such as web pages and images are shown for a fixed duration. Any availability constraints issued by content providers for their channels and by display owners for their public displays are translated into `time_conditions` constraints. The channels system is designed to create Request entries using a sliding window of two days, i.e. entries are periodically created covering the next two days. We employ this technique to limit the number of Request and Playlist entries that have to be validated by the high-level scheduling service. Once the Request and Playlist entries have been created, content in the channels is shown by the high-level scheduling service based on the specified constraints.

The channels system has been implemented and deployed in the e-Campus public display network. The system is used on a daily basis and has become our main tool for getting day-to-day content onto the e-Campus displays. The system's user community includes college and departmental administrators, the theatre on campus, and members of the university's press office. Moreover, we have successfully used the channels system simultaneously with other content schedulers on the same public displays. This included the poster system and the prototype application that allowed users to request content based on their Bluetooth friendly names (see chapter 6, section 6.6.6). In case of the channels and the poster system, both systems were based on the high-level API. Arbitration was performed directly by the high-level scheduling service. Arbitration in the case of the channels system and the Bluetooth friendly name application was handled by the low-level software infrastructure, and the Bluetooth friendly name application was enabled to preempt displays from the channels-based content by using a higher priority level.

The channels system also enables our software infrastructure to operate in the traditional digital signage domain (e.g. in the context of commercial public display deployments in airports or shopping centres). Since the channels system is based on our software infrastructure, such a

deployment will (in addition to its main role as digital signage platform) also provide support for the more advanced features of our software infrastructure, such as context-sensitivity, interactivity, or the precise orchestration of content.

The initial design of the channels system was performed by Prof. Nigel Davies and André Hesse. The Web-based user interface was implemented by Dr. Christos Efstratiou. The Python-based processes for scanning the folder structure and for creating high-level scheduling Requests were implemented by the author of this thesis.

7.5.4 Discussion

In chapter 6 we identified a number of shortcomings that emerged when we analysed how users used the low-level scheduling API to construct customised Schedulers. As a result we designed and implemented the high-level scheduling API and service to address these shortcomings. In this section we discuss the ability of the high-level API to support the content items and experimental applications presented above and relate our experiences to the shortcomings identified in chapter 6.

In general we had no difficulties using the high-level API to integrate constraint-based scheduling functionality into our experimental applications. We therefore conclude that the high-level scheduling service and API has been successful in addressing the "complexity involved in designing and implementing a constraint-Based Scheduler". This complexity is now effectively hidden inside the constraint-based scheduling service. By using the high-level API the functionality provided by this service can be used without requiring knowledge about how to construct such a scheduling service. Users do not even require any knowledge of programming languages to use the high-level API, as our experiences of creating Playlists and Requests by issuing HTTP requests using web browsers have shown.

Using the high-level scheduling service and API, the act of requesting content to be scheduled is decoupled from the act of showing the content: users create Playlist and Request entries in a repository, and these entries are later processed by a background service (the high-level scheduling service). As a result, client applications that use the high-level scheduling API to request content to be scheduled are able to be short-lived and are not required to be active at the time content is

finally shown. This property enabled us to construct the poster system solely using PHP scripts that were served by a Web server. In this model, no long-running threads exist. Activity threads are initiated by user interaction with the Web-based UI and are typically limited to a few seconds. Constructing the poster system solely using the low-level scheduling API would have required the engineering of an additional long-running process that would have been responsible to show the poster content. Using these experiences we therefore conclude that the high-level scheduling service and API have been successful in addressing the "complexity involved in managing the life-cycle of a constraint-based scheduler".

As we have described above, we have so far gained experience with the use of three different programming languages to access the high-level API: Python, Javascript and PHP. HTTP client libraries are available for most modern programming languages. At the time of writing this thesis libraries for translating between native data structures and JSON representations were available for a total of 34 programming languages [JSO08]. Moreover, we have been successful in invoking high-level API operations without the use of any programming language, i.e. by issuing the corresponding HTTP requests using a Web browser. We are therefore confident that the high-level API has successfully addressed the shortcoming identified as "language dependence of the low-level API and the underlying transport protocol".

In our experiments involving the high-level API so far we did not encounter any requirement for personalised access control or accounting at the API level. In the case of the poster system and the channels system access control and accounting were performed at the application-level, i.e. users wishing to use these systems were required to obtain an account for these systems and were subsequently able to access the systems through a log-in page. However, the implementation of personalised priority limits that we described in section 7.3.6 demonstrated that the high-level API does provide the foundation for implementing personalised security policies. We therefore believe that the high-level API has successfully addressed the "lack of support for personalised access control and accounting" that was identified in the context of the low-level scheduling API.

Moreover, our experiences with the high-level scheduling service represent further examples of the power and flexibility of the low-level software infrastructure and its associated API. Firstly, the low-level software infrastructure and API was found to be capable of underpinning the high-level scheduling service. The high-level scheduling service therefore represents a further example of the ability of our low-level software infrastructure to support the construction of a wide variety of

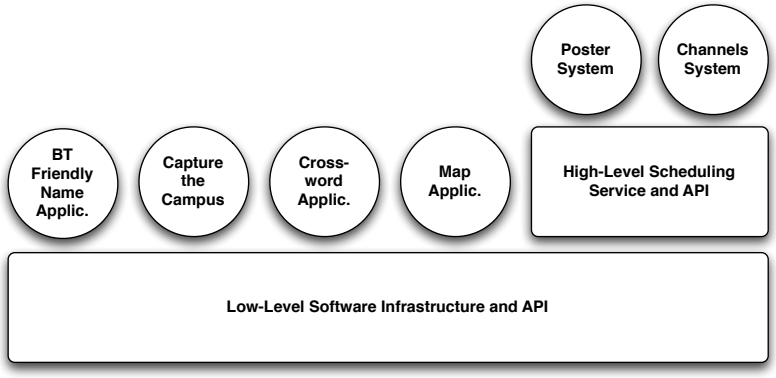


Figure 7.7: Overview of the integration of the different trial systems and applications into our software infrastructure.

Schedulers using its associated API (see requirement R4: "support for a wide range of scheduling criteria"). Secondly, the fact that content scheduled via the high-level API (e.g. through the channels system) was capable of successfully co-existing with Schedulers that were based on the low-level API (e.g. the Bluetooth friendly name system) demonstrated the ability of the low-level software infrastructure and API to support multiple concurrent schedulers that operate on the same set of displays and to arbitrate between conflicting pieces of content (see requirement R3). Figure 7.7 shows an overview of the integration of the different systems and applications into our software infrastructure.

7.6 Summary

In this chapter we have described a constraint-based high-level scheduling service and accompanying API designed to address the shortcomings identified in chapter 6. We have discussed the requirements for such a constraint-based scheduling service and derived a set of concrete design propositions. We have then described the design of the high-level scheduling service and API. The scheduling service was designed as a stand-alone centralised service based on the concepts of Playlists (pre-defined orchestrations of content items) and Requests (specifying the constraints that determine when Playlists are shown). The high-level scheduling API was designed to be HTTP-based to make it accessible from a wide range of operating system platforms and languages. The API provides operations for creating and managing Playlists and Requests and their associated constraints. The high-level scheduling service was implemented in Python. Communication between the Python-based API and the scheduling service is accomplished via a database

back-end that is used to store the representations of Requests and Playlists. The high-level API and scheduling service were evaluated qualitatively using a set of experimental content and two experimental content provision and scheduling tools.

In the next chapter we conclude this thesis by providing a summary of its content, followed by a description of its main contributions and our plans for future work.

Chapter 8

Conclusion

8.1 Overview

In this thesis we have presented the design, implementation and evaluation of a distributed systems infrastructure and associated APIs for controlling the presentation of content on medium-scale research-oriented public display networks.

We started the thesis in chapter 1 by presenting a case for creating deployments of public displays that are treated as infrastructures for research and are therefore opened up to, and shared with, other researchers. In chapter 2 we provided a historical review of research into public display systems and applications and analysed key properties of the surveyed systems. Moreover we presented an overview of public display and digital signage products and projects for commercial and non-research use.

In the following chapter (chapter 3) we presented a set of requirements for designing and building a software infrastructure to control the presentation of content on medium scale public display networks. The requirements were drawn from three main sources: an initial brainstorming phase, a series of experimental public display prototypes that we used as probes and an analysis of the existing public display systems and applications presented in chapter 2.

In chapter 4 we described the computational model of the software infrastructure and an

accompanying scheduling API, both based on the requirements presented in chapter 3. The computational model is built around a small number of functional entities and provides experimenters with a clear set of abstractions for developing experiments and reasoning about the state of the public display system. The scheduling API enables the creation of a diverse range of scheduler types. It provides experimenters with a small set of operations for controlling the life-cycle of content on individual displays and for controlling its visibility. Transactional semantics and operations on Application Groups allow for the creation of orchestrated experiments spanning multiple displays.

In chapter 5 we proposed an approach for engineering the software infrastructure described in chapter 4. We discussed options for mapping the computational entities onto processes, and for distributing these processes onto the machines in a public display network. Moreover, we provided a detailed description of the protocol that is used to interconnect the processes in our engineering model.

We described the implementation status of our software infrastructure in chapter 6, followed by detailed quantitative and qualitative evaluations of the implemented system. The quantitative aspects of our software infrastructure were demonstrated using performance measurements and theoretical calculations. The qualitative evaluation was based on a number of trials conducted with experimental content and on experiences gained from scheduling day-to-day content. While our software infrastructure and API fulfilled the requirements outlined in chapter 3, a number of shortcomings were identified during in-situ use.

We analysed these shortcomings in chapter 7 and presented the design, implementation and qualitative evaluation of a constraint-based high-level scheduling service and accompanying API designed to address the identified shortcomings.

8.2 Contributions

8.2.1 Analysis of Research into Public Display Systems and Applications

Our first set of contributions in this thesis relates to an analysis of the use of public display systems for research purposes. Specifically, in this thesis we have contributed:

C1. *an historic review of research into public display systems and applications.* The review covered over 70 individual pieces of work performed between the early 1980s and the year 2008. Besides providing an overview over the types of experiments that public displays have been used for in the past, the review also demonstrates that public and semi-public displays have provided researchers with a rich platform for experimentation and research in a wide range of disciplines (including HCI, CSCW, sociology and electrical engineering) over the past two decades.

C2. *an analysis of the key properties of the surveyed systems and applications.* The properties were based on four main dimensions: "objectives", "content", "deployment", and "evaluation techniques". The dimension "objectives" provided insights into researchers' motivations for creating and deploying public display prototypes, the benefits these prototypes offered to users and the incentives for display owners to support deployments of displays into the spaces they control. In the "content" category our analysis provided a classification of the nature and formats of content that has been used to support research into public displays and applications. We categorised the providers of content for these research experiments and provided insights into the approaches that were employed for moderating and scheduling content. We identified different purposes for employing interactivity and classified the different types of context-sensitivity that we encountered in the survey. The dimension "deployment" classified the encountered public display prototypes according to criteria, such as the scale of deployments, the deployment locations, the target audiences, and the hardware and software used to realise the prototypes. Finally, the dimension "evaluation techniques" provided an overview of the different methods that researchers used to evaluate the public display prototypes that we had surveyed.

8.2.2 Software Infrastructure Supporting the use of Public Display Networks as Shared Platforms for Research

The second set of major contributions in this thesis relates to the design, implementation and evaluation of a software infrastructure supporting the use of public display networks as platforms for research. The software infrastructure provides researchers and other stakeholders with the means to control the life-cycle and visibility of content on a shared network of public displays and provides support for both experimental and day-to-day content. Specifically in this thesis we have contributed:

C3. *a computational model* that provides researchers with a set of abstractions that enable them to reason about the state of displays and content in public display networks. The computational model hides the complexity of non-standard and dynamic hardware setups from experimenters by exposing combinations of hardware resources as single objects (i.e. conceptual Displays) that are characterised by their properties.

The associated API enables experimenters to construct experiment-specific Schedulers and does not restrict the criteria used by these Schedulers to show content on public displays. This allows experiments to be constructed that, for example, show content on-demand based on user interaction or as a result of context events. Moreover, the API allows experimenters to create orchestrations of content (possibly spanning multiple public displays) by offering precise control over the instantiation and pre-loading of content, its playback states, and its visibility. Operations at this level of abstraction also provide the means for showing content of dynamic length, such as interactive applications. Schedulers using the API can be distributed to arbitrary machines in the display network, for example enabling these Schedulers to interface with native interfaces to interaction devices or sources of context information on these machines. Moreover, the transactional semantics provided by the API allow experimenters to reliably show experimental content that spans multiple displays as an atomic unit.

Our computational model enables multiple experiments to be active in a display network at the same time and arbitrates between competing Schedulers based on the time-division of display airtime and by allowing priority-based preemption of displays.

Moreover, the functionality of Displays can be extended through the use of Handlers. Han-

dlers also provide researchers with the means to audit the visibility of their experiments on the various public displays in the network.

C4. *an extension of the concepts of atomicity and isolation to the visibility of content*: the probes in the Underpass and at the Brewery Arts Centre both indicated a need for being able to display content in an aesthetically pleasing manner. To achieve this we extended the properties of atomicity and isolation (i.e. well-established properties in the contexts of data management and system state) to the visibility of content. Our transactional API operations enable researchers to control multiple content items on different displays as atomic units. Moreover, our transactions ensure that intermittent system states are not visible to human observers. We call this property "visual isolation".

C5. *an engineering model* that describes an approach for mapping the functional entities of our computational model onto machines and processes in a public display network and a protocol to interconnect these entities. The engineering model enables researchers to audit their experiments by monitoring status events emitted by processes and by observing the exchange of protocol messages on the event channel that underpins the software infrastructure.

The engineering model enables researchers to develop interactive and context-sensitive content that interfaces with interaction devices and sensors through native APIs by allowing Applications to be implemented using the researchers' programming languages of choice. Moreover, the model simplifies the construction of Application processes by separating the functionality of Applications into a content-type-specific rendering engine and a reusable protocol engine.

C6. *an evaluation of the proposed software infrastructure* that demonstrated that our software infrastructure meets the requirements presented in chapter 3 for a software infrastructure for controlling the presentation of content in medium-scale research-oriented public display networks.

8.2.3 High-Level Constraint-Based Scheduling Service and Accompanying API

The third major set of contributions of this thesis relates to the design, implementation and evaluation of a high-level scheduling service and API that enables researchers and other stakeholders

to schedule content based on constraints. Specifically in this thesis we have contributed:

C7. *an identification of the requirement for a high-level constraint-based scheduling API*: the requirement is based on the experiences gained with the use of our software infrastructure to support research-related content and day-to-day content. Our investigations show that a significant amount of the content we encountered was scheduled based on constraints, such as the set of targeted displays, priorities, and date and time ranges.

C8. *a design for a high-level constraint-based scheduling service and API* that simplify the use of the software infrastructure for experimentation by providing a secure, platform-independent API to a constraint-based scheduling service that is hosted and managed centrally as part of the overall software infrastructure.

C9. *an evaluation of the high-level scheduling service and API* that demonstrated their ability to support not only a range of day-to-day content, but also to underpin two experimental scheduling systems, one of which has in the meantime become our main means for scheduling day-to-day content on e-Campus displays. Moreover, the evaluation of the high-level scheduling service and API also demonstrated the power and flexibility of the low-level software infrastructure and its associated API to support a variety of schedulers that are able to operate concurrently on the same set of public displays.

8.3 Future Work

8.3.1 Open, Interconnected Networks of Displays

The majority of displays that we find in public and semi-public spaces today simply broadcast content (e.g. advertisements or information) to passers-by. However, the results of a recent observation by Huang et al. of the use of 46 public displays in three European cities have shown [Hua07, HKB08] that if users paid attention to the displays at all, glances were typically limited to a duration of 1-2 seconds. The authors further note [HKB08] that a duration of 1-2 seconds does indeed suggest that the glances were made intentionally.

While we acknowledge that there are probably several factors contributing to these relatively short attention spans, we believe that one of the main factors is that public displays often show

content that is of little or no relevance to users. As a result users divert their attention away from the screens after having sampled the content.

We therefore believe that one of the challenges in the context of public display systems is to find means for increasing the relevance of the displayed content to each individual user, for example by replacing the current model of broadcasting the same content to all users with a model in which displays are capable of showing personalised, possibly interactive content that is tailored to each individual user.

However, currently public display deployments mainly exist as isolated islands of often no more than 10 displays, and each of these islands is typically managed by its own administrative entity. Since authoring high-quality content is generally resource-intensive, we believe it is unlikely that these entities on their own will be able to produce high-quality personalised content and services that are relevant to a large percentage of their users. Instead we believe that a large amount of this content will need to be provided by third parties. Using the analogy of the Web, a percentage of this content might be provided by commercial entities (including newspapers, TV stations and other professional media companies). Other content may be provided directly by individuals either as shared content (e.g. similar to how people use Flickr [Fli08a] or Youtube [You08a] on the Web) or for their own personal consumption. However, to enable contributions of content from such third parties, display networks will have to be opened up to such third-party content, for example requiring the development of new business models for the operation of public display installations.

Moreover, instead of existing as isolated islands, display networks will have to be interconnected on a potentially global scale to enable the widespread deployment of authored content (and potentially of profile information that can be used to personalise the displays).

The resulting wide-area display networks of potentially global scale will require new types of distributed systems support and paradigms for deploying content and for adapting the selection of content to both deployment contexts and users.

The personalisation of content in multiple administrative domains raises additional issues in the context of security and privacy, such as the safe-keeping and exchange of information regarding users' preferences, or determining appropriate times and locations for showing personal information on public displays.

8.3.2 High-Level Tool Support

The software infrastructure presented in this thesis provides the foundations for allowing researchers to use public display networks as platforms for experimentation, for example by providing the means for controlling the life-cycle and visibility of experiments, and for auditing these actions and the state of the public display network.

In the future we would hope to use this foundation to provide a set of integrated high-level tools supporting the whole workflow involved in preparing for and carrying out experiments. Tool sets with similar aims have, for example, been developed for Grid-based experimentation in the context of the UK-based MyGrid [MyG08] and Comb*e*Chem [Com08] projects.

We would envision tools, for example, to assist researchers with discovering appropriate displays (or even display networks) according to their properties, such as display type and size, deployment location and audience. During the development cycle researchers may want to periodically test prototypes of content, Applications and Schedulers that are being developed for the experiment. Ideally, it should be possible to perform these tests off-line, i.e. without showing unfinished content on the actual public display network. Suitable test environments may, for example, be provided in the form of a set of sandboxed display installations that are not accessible to the general public, or in the form of emulators that allow researchers to evaluate their Schedulers without requiring a fully featured public display installation.

Moreover, we would envision that high-level tools may be provided to assist researchers with packaging up the experiment-specific content and Applications, and with deploying them to the targeted display machines. Once experiments have been started, we would expect researchers to be able to monitor their experiments and collect results using high-level graphical front-ends.

8.3.3 Content Management and Content Distribution

Content is at the heart of every public display deployment. Our experiences so far with building an open, shared public display network on the campus of Lancaster University have revealed that managing the constant stream of incoming content and the associated workflows is a non-trivial task. Content has to be approved for play-out, taking into account various constraints that may be in place for specific locations. For example, content that is perfectly suitable for a college bar

might not be appropriate for more outward-facing locations of the university, such as the main reception area of the university's administration.

Once approved, content has to be scheduled and distributed to the individual display machines. While some content items can be hosted centrally and simply streamed to the involved computers at play-out time, this is, for example, impossible in the case of high-quality videos due to the size of the underlying media files. In these cases, content has to be distributed in advance to the computers involved and, as disk space is limited on these computers, removed again after play-out is completed.

A significant percentage of the incoming content is submitted in media formats that are unsuitable for playback in the public display network and has to be manually re-encoded. As a result, we regularly end up with multiple copies of content. Moreover, since the software infrastructure underpinning the public display network is itself the subject of research (large parts of it were presented in this thesis), content is typically archived, allowing us, for example, to compile statistics or to re-use the content for demonstrations.

In our experience so far we have found that existing content management systems provide inadequate support for these workflows. We believe that as public display networks grow, are opened up to content contributed by third-parties, and are possibly networked together, the demand for tailored solutions for the management of content in public display networks and the associated workflows is likely to increase.

8.3.4 Interaction With Public Displays

From our own experience with deploying and operating a campus-wide public display network we have learned that the provision of interaction techniques that allow for rich, fluid user interactions with public displays in a variety of contexts is still a challenge. While a significant amount of research has already been undertaken into different techniques for interacting with public displays, the developed approaches often come with significant drawbacks, complicating the deployment and maintenance of interactive displays on a larger scale. Examples include approaches for camera-based gesture-recognition that require users to wear reflective markers in order to differentiate between uninvolved bystanders and active users, touch-based interaction that requires regular

cleaning of the display surfaces and is unable to support interaction with displays from a distance, or mobile-phone-based approaches that require users to download software prior to being able to interact.

The lack of appropriate interaction technologies that are easy to use and allow for rich interactions with public displays severely limits user experience and the types of applications that can be deployed onto public displays. We therefore believe that additional research into the area of interaction with public displays is required to be able to use public displays to their full potential.

8.4 Closing Remarks

Public and semi-public displays have historically served and continue to serve as a rich platform for research and experimentation in a wide range of disciplines. Moreover, the past decade has seen a significant and continuing increase in the number of displays deployed into public spaces. The move away from the current model of broadcasting advertisements and information to a model where public displays act as rich, ubiquitous platforms for accessing personalised information poses significant challenges, making public display systems and applications an important research topic.

In this thesis we have presented a software infrastructure that enables networks of public displays to be used as shared platforms for research. The software infrastructure is divided into two parts: a low-level infrastructure and API that provide researchers with the means to control the life-cycle and visibility of content and to audit their experiments, and a high-level scheduling service (layered on top of the low-level API) and API that enable researchers to schedule content based on constraints. Both parts of the infrastructure were evaluated qualitatively in the context of the e-Campus public display network and proved able to provide support for a range of experimental content and applications.

We hope that the work presented in this thesis provides a foundation towards the more widespread sharing and re-use of public display deployments as platforms for research. Moreover, we hope that continued research in this field will help to turn the continually growing public display landscape into a rich, ubiquitous platform for information access and sharing for all of us.

Bibliography

[3M08] 3M. Digital Signage Network Products from 3M Digital Signage. `http://www.3mdigitalsignage.com`, May 2008. Last accessed 12 May 2008.

[AMX08] AMX. AMX NetLinx Controllers. `http://www.amx.com/products/categoryNetLinxControllers.asp`, May 2008. Last accessed 27 May 2008.

[AS03] Stavros Antifakos and Bernt Schiele. LaughingLily: Using a Flower as a Real World Information Display. In *Fifth International Conference on Ubiquitous Computing (UbiComp 2003)*. Seattle, Washington, USA, October 2003. Poster presentation.

[ASB+99] David Arnold, Bill Segall, Julian Boot, Andy Bond, Melfyn Lloyd, and Simon Kaplan. Discourse with disposable computers: how and why you will talk to your tomatoes. In *WOES'99: Proceedings of the Workshop on Embedded Systems on Workshop on Embedded Systems*, pages 2–2. USENIX Association, Berkeley, CA, USA, 1999.

[AWB97] Stefan Agamanolis, Alex Westner, and V. Michael Bove, Jr. Reflection of Presence: Toward more natural and responsive telecollaboration. In *SPIE Multimedia Networks*, volume 3228A. Dallas, November 1997.

[Ads08] Adspace. Adspace Mall Network. `http://www.adspacenetworks.com`, May 2008. Last accessed 12 May 2008.

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/182.358434.

[App08] Apple Inc. Mac OS X. `http://www.apple.com/macosx/`, June 2008. Last accessed 10 June 2008.

[BBC+04] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating system support for planetary-scale network services. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 19–19. USENIX Association, Berkeley, CA, USA, 2004.

[BBC08] BBC. The BBC Big Screens. `http://www.bbc.co.uk/bigscreens/`, May 2008. Last accessed 23 May 2008.

[BHI93] Sara A. Bly, Steve R. Harrison, and Susan Irwin. Media spaces: bringing people together in a video, audio, and computing environment. *Commun. ACM*, 36(1):28–46, 1993. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/151233.151235.

[BHS06] Jakob E. Bardram, Thomas R. Hansen, and Mads Soegaard. AwareMedia: a shared interactive display supporting social, temporal, and spatial awareness in surgery. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 109–118. ACM, New York, NY, USA, 2006. ISBN 1-59593-249-6. doi:http://doi.acm.org/10.1145/1180875.1180892.

[BIF+04] Harry Brignull, Shahram Izadi, Geraldine Fitzpatrick, Yvonne Rogers, and Tom Rodden. The introduction of a shared interactive surface into a communal space. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 49–58. ACM, New York, NY, USA, 2004. ISBN 1-58113-810-5. doi:http://doi.acm.org/10.1145/1031607.1031616.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987. ISSN 0734-2071. doi:http://doi.acm.org/10.1145/7351.7478.

[BJ04] R. Beale and M. Jones. Integrating Situated Interaction with Mobile Awareness. In E. Murelli, G. Da Bormida, and C. Alborghetti, editors, *Third European Conference on Mobile Learning (MLEARN 2004)*. Odescalchi Castle, Lake Bracciano, Rome, Italy, 2004.

[BKN+05] Stefan Berger, Rick Kjeldsen, Chandra Narayanaswami, Claudio Pinhanez, Mark Podlaseck, and Mandayam Raghunath. Using Symbiotic Dis-

plays to View Sensitive Information in Public. In *PERCOM '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*, pages 139–148. IEEE Computer Society, Washington, DC, USA, 2005. ISBN 0-7695-2299-8. doi:http://dx.doi.org/10.1109/PERCOM.2005.52.

[BLFM05] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, Internet Engineering Task Force, January 2005.

[BM98] Marc Böhlen and Michael Mateas. Office Plant #1: Intimate Space and Contemplative Entertainment. *Leonardo*, 31(5):345–348, 1998. ISSN 0024094X.

[BMMR96] R. Borovoy, M. McDonald, F. Martin, and M. Resnick. Things that blink: computationally augmented name tags. *IBM Syst. J.*, 35(3-4):488–495, 1996. ISSN 0018-8670.

[BMRS98] Richard Borovoy, Fred Martin, Mitchel Resnick, and Brian Silverman. GroupWear: nametags that tell about relationships. In *CHI '98: CHI 98 conference summary on Human factors in computing systems*, pages 329–330. ACM, New York, NY, USA, 1998. ISBN 1-58113-028-7. doi:http://doi.acm.org/10.1145/286498.286799.

[BMV+98] Richard Borovoy, Fred Martin, Sunil Vemuri, Mitchel Resnick, Brian Silverman, and Chris Hancock. Meme tags and community mirrors: moving from conferences to collaboration. In *CSCW '98: Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 159–168. ACM, New York, NY, USA, 1998. ISBN 1-58113-009-0. doi:http://doi.acm.org/10.1145/289444.289490.

[BPSM+06] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible Markup Language (XML) 1.1 (Second Edition). W3C Recommendation, W3C - World Wide Web Consortium, September 2006.

[BR03] Harry Brignull and Yvonne Rogers. Enticing People to Interact with Large Public Displays in Public Spaces. In Matthias Rauterberg, Marino Menozzi, and Janet Wesson, editors, *INTERACT*. IOS Press, Zürich, Switzerland, September 2003. ISBN 1-58603-363-8.

[BRSB04] Rafael Ballagas, Michael Rohs, Jennifer Sheridan, and Jan Borchers. BYOD: Bring Your Own Device. In *Proceedings of the Workshop on Ubiquitous*

Display Environments at the Sixth International Conference on Ubiquitous Computing (UbiComp 2004). Nottingham, UK, September 2004.

[Bli08] Blinkenlights. Project Blinkenlights. `http://www.blinkenlights.de/`, May 2008. Last accessed 12 May 2008.

[CAS08] CASIDE. The CASIDE Project: Hermes Door Displays. `http://www.caside.lancs.ac.uk/hermes.php`, May 2008. Last accessed 13 May 2008.

[CCD+04] Scott Carter, Elizabeth Churchill, Laurent Denoue, Jonathan Helfman, and Les Nelson. Digital graffiti: public annotation of multimedia content. In *CHI '04: CHI '04 extended abstracts on Human factors in computing systems*, pages 1207–1210. ACM, New York, NY, USA, 2004. ISBN 1-58113-703-6. doi: http://doi.acm.org/10.1145/985921.986025.

[CDF+05] Keith Cheverst, Alan Dix, Daniel Fitton, Chris Kray, Mark Rouncefield, Corina Sas, George Saslis-Lagoudakis, George Saslis-Lagoudakis, and Jennifer G. Sheridan. Exploring bluetooth based mobile phone interaction with the hermes photo display. In *MobileHCI '05: Proceedings of the 7th international conference on Human computer interaction with mobile devices & services*, pages 47–54. ACM, New York, NY, USA, 2005. ISBN 1-59593-089-2. doi: http://doi.acm.org/10.1145/1085777.1085786.

[CFDR02] K. Cheverst, D. Fitton, A. Dix, and M. Rouncefield. Exploring Situated Interaction with Ubiquitous Office Door Displays. In *Workshop on Situated Interaction at CSCW '02*. New Orleans, USA, 2002.

[CLS+05] Andrea Chew, Vincent Leclerc, Sajid Sadi, Aaron Tang, and Hiroshi Ishii. SPARKS. In *CHI '05: CHI '05 extended abstracts on Human factors in computing systems*, pages 1276–1279. ACM, New York, NY, USA, 2005. ISBN 1-59593-002-7. doi:http://doi.acm.org/10.1145/1056808.1056895.

[CND+03] Elizabeth F. Churchill, Laurent Nelson, Laurent Denoue, Paul Murphy, and Jonathan Helfman. The Plasma Poster Network: Social Hypermedia on Public Display. In K. O'Hara, M. Perry, E. Churchill, and D. Russell, editors, *Public and Situated Displays. Social and Interactional Aspects of Shared Dis-*

play Technologies, chapter The Plasma Poster Network – Social Hypermedia on Public Display. Kluwer Academic Publishers, London, December 2003.

[CND+04] Elizabeth F. Churchill, Les Nelson, Laurent Denoue, Jonathan Helfman, and Paul Murphy. Sharing multimedia content with interactive public displays: a case study. In *DIS '04: Proceedings of the 5th conference on Designing interactive systems*, pages 7–16. ACM, New York, NY, USA, 2004. ISBN 1-58113-787-7. doi:http://doi.acm.org/10.1145/1013115.1013119.

[CNDG03] Elizabeth F. Churchill, Les Nelson, Laurent Denoue, and Andreas Girgensohn. The Plasma Poster Network: Posting Multimedia Content in Public Places. In *INTERACT 2003, Ninth IFIP TC13 International Conference on Human-Computer Interaction*, pages 599–606. IOS Press, September 2003.

[CRW01] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS)*, 19(3):332–383, 2001. ISSN 0734-2071. doi:http://doi.acm.org/10.1145/380749.380767.

[Com08] CombeChem. Combe*C*hem. `http://www.combechem.org/`, August 2008. Last accessed 28 August 2008.

[Con07] Ben Congleton. Prospero – A "visual commons" framework for public displays, April 2007. Summary Project Report.

[Cyc08] Cycling '74. Max/MSP. `http://www.cycling74.com/products/max5`, May 2008. Last accessed 27 May 2008.

[DNC03] Laurent Denoue, Les Nelson, and Elizabeth Churchill. AttrActive windows: dynamic windows for digital bulletin boards. In *CHI '03: CHI '03 extended abstracts on Human factors in computing systems*, pages 746–747. ACM, New York, NY, USA, 2003. ISBN 1-58113-637-4. doi:http://doi.acm.org/10.1145/765891.765966.

[DWI98] Andrew Dahley, Craig Wisneski, and Hiroshi Ishii. Water lamp and pinwheels: ambient projection of digital information into architectural space. In *CHI '98: CHI 98 conference summary on Human factors in computing systems*, pages 269–270. ACM, New York, NY, USA, 1998. ISBN 1-58113-028-7. doi:http://doi.acm.org/10.1145/286498.286750.

[Dyn08] Dynamax. PointOfView^{NG} Digital Signage. http://www.dynamax.co.uk, May 2008. Last accessed 12 May 2008.

[E I05] E Ink Co. and Toppan Printing Co. Toppan Exhibits Wall-Sized Electronic Paper Sign at Expo 2005 Aichi, Japan. http://www.e-ink.com/press/releases/pr80.html, March 2005. Press Release. Last accessed 25 November 2008.

[E I08] E Ink Corporation. E Ink. http://www.e-ink.com, November 2008. Last accessed 25 November 2008.

[Eas04] Douglas Easterly. Bio-Fi: inverse biotelemetry projects. In *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, pages 182–183. ACM, New York, NY, USA, 2004. ISBN 1-58113-893-8. doi:http://doi.acm.org/10.1145/1027527.1027568.

[FB99] Jennica Falk and Staffan Björk. The BubbleBadge: a wearable public display. In *CHI '99: CHI '99 extended abstracts on Human factors in computing systems*, pages 318–319. ACM, New York, NY, USA, 1999. ISBN 1-58113-158-5. doi:http://doi.acm.org/10.1145/632716.632909.

[FFM07] M. Feller, I. Foster, and S. Martin. GT4 GRAM: A Functionality and Performance Study, 2007. Submitted to TeraGrid '07.

[FFP02] Adam Fass, Jodi Forlizzi, and Randy Pausch. MessyDesk and MessyBoard: two designs inspired by the goal of improving human memory. In *DIS '02: Proceedings of the 4th conference on Designing interactive systems*, pages 303 – 311. ACM, New York, NY, USA, 2002. ISBN 1-58113-515-7. doi:http://doi.acm.org/10.1145/778712.778754.

[FK98] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, July 1998.

[FKC90] Robert S. Fish, Robert E. Kraut, and Barbara L. Chalfonte. The VideoWindow system in informal communication. In *CSCW '90: Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, pages 1–11. ACM, New York, NY, USA, 1990. ISBN 0-89791-402-3. doi:http://doi.acm.org/10.1145/99332.99335.

[FMK+99] Geraldine Fitzpatrick, Tim Mansfield, Simon Kaplan, David Arnold, Ted Phelps, and Bill Segall. Augmenting the workaday world with Elvin. In *Proceedings of the Sixth European conference on Computer supported cooperative work*, pages 431–450. Kluwer Academic Publishers, Norwell, MA, USA, 1999. ISBN 0-7923-5948-8.

[FP04] A. Fass and R. Pausch. MessyBoard: Lowering the Cost of Communication and Making it More Enjoyable. In *Doctoral Symposium at UIST 2004 Symposium on User Interface Software and Technology*. 2004.

[FWD+96] J. Finney, S. Wade, N. Davies, , and A. Friday. Flump: The flexible ubiquitous monitor project. In *Cabernet Radicals Workshop*. May 1996.

[Far01] Stephen Farrell. Social and informational proxies in a fishtank. In *CHI '01: CHI '01 extended abstracts on Human factors in computing systems*, pages 365–366. ACM, New York, NY, USA, 2001. ISBN 1-58113-340-5. doi:http://doi.acm.org/10.1145/634067.634283.

[Fir08] Firefly. The Firefly project. `http://www.comp.lancs.ac.uk/~dixa/projects/firefly/`, June 2008. Last accessed 03 June 2008.

[Fle03] Rowanne Fleck. How the move to physical user interfaces can make human computer interaction a more enjoyable experience. In *Physical Interaction (PI03) – Workshop on real world interfaces at Mobile HCI*. Udine, Italy, 2003.

[Fli08a] Flickr. Flickr. `http://www.flickr.com`, July 2008. Last accessed 14 July 2008.

[Fli08b] Flickr. Flickr API. `http://www.flickr.com/services/api/`, July 2008. Last accessed 22 July 2008.

[Fog08] FogScreen Inc. FogScreen. `http://www.fogscreen.com`, May 2008. Last accessed 09 May 2008.

[Fos06] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *IFIP International Conference on Network and Parallel Computing*, LNCS 3779, pages 2–13. Springer-Verlag, 2006.

[GA86] George O. Goodman and Mark J. Abel. Collaboration research in SCL. In *CSCW '86: Proceedings of the 1986 ACM conference on Computer-supported coop-*

erative work, pages 246–251. ACM, New York, NY, USA, 1986. ISBN 1-23-456789-0. doi:http://doi.acm.org/10.1145/637069.637099.

[GBCv93] B. B. Glade, K. P. Birman, R. C. B. Cooper, and R. van Renesse. Lightweight process groups in the Isis system. *Distributed Systems Engineering*, 1(1):29–36, 1993.

[GMS03] A. Grasso, M. Mühlenbrock, and D. Snowdon. Supporting Communities of Practice with Large Screen Displays. In K. O'Hara, M. Perry, E. Churchill, and D. Russell, editors, *Public and Situated Displays: Social and Interactional Aspects of Shared Display Technologies*. Kluwer Academic Publishers, 2003.

[GR80] Kit Galloway and Sherrie Rabinowitz. Hole-In-Space. `http://www.ecafe.com/getty/HIS/`, November 1980. Last accessed 01 May 2008.

[GR01] Saul Greenberg and Michael Rounding. The notification collage: posting information to public and personal displays. In *CHI '01: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 514–521. ACM, New York, NY, USA, 2001. ISBN 1-58113-327-8. doi:http://doi.acm.org/10.1145/365024.365339.

[Glo08a] Globus. GT 4.0 WS_GRAM. `http://www.globus.org/toolkit/docs/4.0/execution/wsgram/`, June 2008. Last accessed 18 June 2008.

[Glo08b] Globus. The Globus Tookit. `http://www.globus.org/toolkit/`, June 2008. Last accessed 18 June 2008.

[Goo08a] Google. Google. `http://www.google.com/`, July 2008. Last accessed 14 July 2008.

[Goo08b] Google. Google Calendar API. `http://code.google.com/apis/calendar/developers_guide_protocol.html`, July 2008. Last accessed 22 July 2008.

[Goo08c] Google Maps. Google Maps. `http://maps.google.com/`, July 2008. Last accessed 14 July 2008.

[Gra78] Jim Gray. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*, pages 393–481. Springer-Verlag, London, UK, 1978. ISBN 3-540-08755-9.

[Gre99] S. Greenberg. Designing Computers As Public Artifacts. *International Journal of Design Computing: Special Issue on Design Computing on the Net (DCNet'99)*, November 1999.

[HBL98] Stephanie Houde, Rachel Bellamy, and Laureen Leahy. In search of design principles for tools and practices to support communication within a learning community. *SIGCHI Bulletin*, 30(2):113–118, 1998. ISSN 0736-6906. doi: http://doi.acm.org/10.1145/279044.279171.

[HHT99] Jeremy M. Heiner, Scott E. Hudson, and Kenichiro Tanaka. The information percolator: ambient information display in a decorative object. In *UIST '99: Proceedings of the 12th annual ACM symposium on User interface software and technology*, pages 141–148. ACM, New York, NY, USA, 1999. ISBN 1-58113-075-9. doi:http://doi.acm.org/10.1145/320719.322595.

[HHTM04] Manveer Heir, Harish Hoon, Goldie Terrell, and D. Scott McCrickard. Online Enlightenment: A Phidget Notification System for Online Status. Technical Report Technical Report TR-04-30, Department of Computer Science, Virginia Polytechnic Institute and State University, 2004.

[HKB08] E. M. Huang, A. Koster, and J. Borchers. Overcoming assumptions and uncovering practices: When does the public really look at public displays? In *Sixth International Conference on Pervasive Computing (Pervasive 2008)*. Sydney, Australia, May 2008.

[HKH+04] David Holstius, John Kembel, Amy Hurst, Peng-Hui Wan, and Jodi Forlizzi. Infotropism: living and robotic plants as interactive displays. In *DIS '04: Proceedings of the 5th conference on Designing interactive systems*, pages 215–221. ACM, New York, NY, USA, 2004. ISBN 1-58113-787-7. doi:http://doi.acm.org/10.1145/1013115.1013145.

[HM03] Elaine M. Huang and Elizabeth D. Mynatt. Semi-public displays for small, co-located groups. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 49–56. ACM, New York, NY, USA, 2003. ISBN 1-58113-630-7. doi:http://doi.acm.org/10.1145/642611.642622.

[HRKS06] Paul Holleis, Enrico Rukzio, Thomas Kraus, and Albrecht Schmidt. Environment Based Messaging. In *Advances in Pervasive Computing 2006*, page 55. Austrian Computer Society, May 2006. ISBN 3-85403-207-2.

[HRS04] Elaine M. Huang, Daniel M. Russell, and Alison E. Sue. IM here: public instant messaging on large, shared displays for workgroup interactions. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 279–286. ACM, New York, NY, USA, 2004. ISBN 1-58113-702-8. doi:http://doi.acm.org/10.1145/985692.985728.

[HS03] Lars Erik Holmquist and Tobias Skog. Informative art: information visualization in everyday environments. In *GRAPHITE '03: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 229–235. ACM, New York, NY, USA, 2003. ISBN 1-58113-578-5. doi:http://doi.acm.org/10.1145/604471.604516.

[HTCM02] Elaine M. Huang, Joe Tullio, Tony J. Costa, and Joseph F. McCarthy. Promoting awareness of work activities through peripheral displays. In *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*, pages 648–649. ACM, New York, NY, USA, 2002. ISBN 1-58113-454-1. doi:http://doi.acm.org/10.1145/506443.506527.

[Hua07] Elaine M. Huang. When does the public look at public displays? In *Companion Proceedings of the 9th International Conference on Ubiquitous Computing (UbiComp 2007)*. Innsbruck, Austria, September 2007.

[IBR+03] Shahram Izadi, Harry Brignull, Tom Rodden, Yvonne Rogers, and Mia Underwood. Dynamo: A public interactive surface supporting the cooperative sharing and exchange of media. In *UIST '03: Proceedings of the 16th annual ACM symposium on User interface software and technology*, pages 159–168. ACM, New York, NY, USA, 2003. ISBN 1-58113-636-6. doi:http://doi.acm.org/10.1145/964696.964714.

[INF08a] INFOSCREEN. INFOSCREEN. `http://www.infoscreen.de`, May 2008. Last accessed 12 May 2008.

[INF08b] INFOSCREEN Austria. INFOSCREEN Austria. http://www.infoscreen.at, May 2008. Last accessed 12 May 2008.

[IRF01] Hiroshi Ishii, Sandia Ren, and Phil Frei. Pinwheels: visualizing information flow in an architectural space. In *CHI '01: CHI '01 extended abstracts on Human factors in computing systems*, pages 111–112. ACM, New York, NY, USA, 2001. ISBN 1-58113-340-5. doi:http://doi.acm.org/10.1145/634067.634135.

[ISO95] ISO/IEC. Open Distributed Processing Reference Model, Parts 1-4. ITU-T Rec. ITU-T Rec. X.901 | ISO/IEC 10746-1 to ITU-T Rec. X.904 |ISO/IEC 10746-4, ISO/IEC, 1995.

[IT06] Funkwerk IT. First Ever Installation of Electronic Paper Display on HAMBURG HOCHBAHN Train by Vossloh IT. http://www.funkwerk-it.com/fs_cms/en/press/press_releases/Press_release_1377.html, July 2006. Press Release. Last accessed 25 November 2008.

[JCT+97] J. Jacobson, B. Comiskey, C. Turner, J. Albert, and P. Tsao. The Last Book. *IBM Systems Journal*, 36(3):457–463, 1997.

[JSO08] JSON. The JavaScript Object Notation. http://www.json.org/, June 2008. Last accessed 13 June 2008.

[JVG+01] Gavin Jancke, Gina Danielle Venolia, Jonathan Grudin, J. J. Cadiz, and Anoop Gupta. Linking public spaces: technical and social issues. In *CHI '01: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 530–537. ACM, New York, NY, USA, 2001. ISBN 1-58113-327-8. doi:http://doi.acm.org/10.1145/365024.365352.

[Joh03] Brad Johanson. *Application Coordination Infrastructure for Ubiquitous Computing Rooms*. Ph.D. thesis, Stanford University, December 2003.

[KD04] Karrie Karahalios and Judith Donath. Telemurals: linking remote spaces with social catalysts. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 615–622. ACM, New York, NY, USA, 2004. ISBN 1-58113-702-8. doi:http://doi.acm.org/10.1145/985692.985770.

[KHS+08] Dagmar Kern, Michael Harding, Oliver Storz, Nigel Davis, and Albrecht Schmidt. Shaping how advertisers see me: user views on implicit and explicit profile capture. In *CHI '08: CHI '08 extended abstracts on Human factors in computing systems*, pages 3363–3368. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-012-X. doi:http://doi.acm.org/10.1145/1358628.1358858.

[KKK05] Christian Kray, Gerd Kortuem, and Antonio Krüger. Adaptive navigation support with public displays. In *IUI '05: Proceedings of the 10th international conference on Intelligent user interfaces*, pages 326–328. ACM, New York, NY, USA, 2005. ISBN 1-58113-894-6. doi:http://doi.acm.org/10.1145/1040830.1040916.

[KM00] D. Kristol and L. Montulli. HTTP State Management Mechanism. RFC 2965, Internet Engineering Task Force, October 2000.

[KPD07] M. Karam, T.R. Payne, and E. David. Evaluating BluScreen: Usability for Intelligent Pervasive Displays. In *The Second IEEE International Conference on Pervasive Computing and Applications (ICPCA '07)*. Birmingham, UK, July 2007.

[KW06] Satoshi Kuribayashi and Akira Wakita. PlantDisplay: turning houseplants into ambient display. In *ACE '06: Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology*, page 40. ACM, New York, NY, USA, 2006. ISBN 1-59593-380-8. doi:http://doi.acm.org/10.1145/1178823.1178871.

[LBF99] Peter Ljungstrand, Staffan Björk, and Jennica Falk. The WearBoy: A Platform for Low-Cost Public Wearable Devices. In *ISWC '99: Proceedings of the 3rd IEEE International Symposium on Wearable Computers*, page 195. IEEE Computer Society, Washington, DC, USA, 1999. ISBN 0-7695-0428-0.

[LS79] B. W. Lampson and H. E. Sturgis. Crash recovery in a distributed data storage system. Technical report, Xerox Palo Alto Research Center, June 1979.

[Lit94] Thomas D. C. Little. Time-based media representation and delivery. In *Multimedia systems*, pages 175–200. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994. ISBN 0-201-53258-1.

[MC05] René Meier and Vinny Cahill. Taxonomy of Distributed Event-Based Programming Systems. *The Computer Journal*, 48(5):602–626, 2005. ISSN 0010-4620. doi: http://dx.doi.org/10.1093/comjnl/bxh120.

[MCL01] Joseph F. McCarthy, Tony J. Costa, and Edy S. Liongosari. UniCast, OutCast & GroupCast: Three steps toward ubiquitous, peripheral displays. In *3rd International Conference on Ubiquitous Computing (UbiComp'01)*, pages 332—345. Springer Verlag, Atlanta, Georgia, 2001.

[MDS+04] Joseph F. McCarthy, David W. McDonald, Suzanne Soroczak, David H. Nguyen, and Al M. Rashid. Augmenting the social space of an academic conference. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 39–48. ACM, New York, NY, USA, 2004. ISBN 1-58113-810-5. doi:http://doi.acm.org/10.1145/1031607.1031615.

[MHT04] Kento Miyaoku, Suguru Higashino, and Yoshinobu Tonomura. C-blink: a hue-difference-based light signal marker for large screen interaction via any mobile terminal. In *UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 147–156. ACM, New York, NY, USA, 2004. ISBN 1-58113-957-8. doi:http://doi.acm.org/10.1145/1029632.1029657.

[MK06] Jörg Müller and Antonio Krüger. Towards Situated Public Displays as Multicast Systems. In *UbiqUM 2006 Workshop on Ubiquitous User Modeling, The 17th European Conference on Artificial Intelligence.* 2006.

[MR97] Nobuyuki Matsushita and Jun Rekimoto. HoloWall: Designing a Finger, Hand, Body, and Object Sensitive Wall. In *Proceedings of UIST'97.* 1997.

[MRS06] K. Mitchell, N. J. P. Race, and M. Suggitt. iCapture: Facilitating Spontaneous User-Interaction with Pervasive Displays using Smart Devices. In *Workshop on Pervasive Mobile Interaction Devices - Mobile Devices as Pervasive User Interfaces and Interaction Devices (PERMID) at Pervasive 2006.* Dublin, Ireland, May 2006.

[Max08] Maxim Integrated Products. iButton. `http://www.maxim-ic.com/products/ibutton/`, May 2008. Last accessed 09 May 2008.

[McC02] Joseph F. McCarthy. Using Public Displays to Create Conversation Opportunities. In *Workshop on Public, Community, and Situated Displays at the ACM 2002*

Conference on Computer Supported Cooperative Work (CSCW 2002). New Orleans, November 2002.

[Mea55] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[Mid08] Midori Mark. Midori Mark. `http://www.midorimark.co.jp/`, November 2008. Last accessed 25 November 2008.

[Mis06] Kazuo Misue. Mosaic View: Modest and Informative Display. In *2006 International Conference on Image Processing, Computer Vision, & Pattern Recognition (IPCV'06)*, pages 250–255. June 2006.

[MyG08] MyGrid. MyGrid. `http://www.mygrid.org.uk/`, August 2008. Last accessed 28 August 2008.

[MyS08] MySQL. MySQL. `http://www.mysql.com/`, July 2008. Last accessed 23 July 2008.

[NTDM00] D. H. Nguyen, J. Tullio, T. Drewes, and E. D. Mynatt. Dynamic Door Displays. Technical Report GIT-GVU-00-30, Graphics, Visualization & Usability Center, Georgia Tech, 2000.

[Neo08] Neolux. Neolux. `http://neoluxiim.com/`, November 2008. Last accessed 25 November 2008.

[Net08] Netpresenter. Netpresenter. `http://www.netpresenter.com`, May 2008. Last accessed 12 May 2008.

[New97] New York Times. IMES SQ. SIGN TURNS CORNER INTO SILEN. `http://query.nytimes.com/gst/fullpage.html?res=9F01E6D81330F935A35756C0A961958260`, May 1997. Last accessed 01 May 2008.

[OHU+05] Kenton O'Hara, Richard Harper, Axel Unger, James Wilkes, Bill Sharpe, and Marcel Jansen. TxtBoard: from text-to-person to text-to-home. In *CHI '05: CHI '05 extended abstracts on Human factors in computing systems*, pages 1705–1708. ACM, New York, NY, USA, 2005. ISBN 1-59593-002-7. doi: http://doi.acm.org/10.1145/1056808.1057002.

[OLJ+04] Kenton O'Hara, Matthew Lipson, Marcel Jansen, Axel Unger, Huw Jeffries, and Peter Macer. Jukola: democratic music choice in a public space. In *DIS '04: Proceedings of the 5th conference on Designing interactive systems*, pages 145–154. ACM, New York, NY, USA, 2004. ISBN 1-58113-787-7. doi: http://doi.acm.org/10.1145/1013115.1013136.

[OPL03] Kenton O'Hara, Mark Perry, and Simon Lewis. Situated Web Signs and the Ordering of Social Action: Social Co-ordination Around a Public Room Reservation Display Appliance. In Kenton O'Hara, Ethan Perry, Elizabeth Churchill, and Daniel M. Russel, editors, *Public and Situated Displays - Social and Interactional Aspects of Shared Display Technologies*, pages 105 – 140. Kluwer Academic Publishers, Dordrecht, 2003.

[Ope08] OpenGL. OpenGL. `http://www.opengl.org/`, June 2008. Last accessed 26 June 2008.

[PHP08] PHP Group. The PHP Hypertext Preprocessor. `http://www.php.net/`, May 2008. Last accessed 09 May 2008.

[PPL+03] Gopal Pingali, Claudio Pinhanez, Anthony Levas, Rick Kjeldsen, Mark Podlaseck, Han Chen, and Noi Sukaviriya. Steerable Interfaces for Pervasive Computing Spaces. In *PERCOM '03: Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, page 315. IEEE Computer Society, Washington, DC, USA, 2003. ISBN 0-7695-1893-1.

[PRS+03] Thorsten Prante, Carsten Röcker, Norbert Streitz, Richard Stenzel, Carsten Magerkurth, Daniel van Alphen, and Daniela Plewe. Hello.Wall®– Beyond Ambient Displays. In *Video Track and Adjunct Proceedings of the Fifth International Conference on Ubiquitous Computing (UBICOMP'03)*, pages 277–278. Springer, Seattle, WA, USA, October 2003.

[PSJ+07] Peter Peltonen, Antti Salovaara, Giulio Jacucci, Tommi Ilmonen, Carmelo Ardito, Petri Saarikko, and Vikram Batra. Extending large-scale event participation with user-created mobile media on a public display. In *MUM '07: Proceedings of the 6th international conference on Mobile and ubiquitous multi-*

media, pages 131–138. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-916-6. doi:http://doi.acm.org/10.1145/1329469.1329487.

[PSR+04] Thorsten Prante, Richard Stenzel, Carsten Röcker, Norbert Streitz, and Carsten Magerkurth. Ambient agoras: InfoRiver, SIAM, Hello.Wall. In *CHI '04: CHI '04 extended abstracts on Human factors in computing systems*, pages 763–764. ACM, New York, NY, USA, 2004. ISBN 1-58113-703-6. doi:http://doi.acm.org/10.1145/985921.985924.

[Pas08] Passenger Terminal Today. Manchester's Terminal 2 goes digital. `http://www.passengerterminaltoday.com/news.php?NewsID=3242`, May 2008. Last accessed 01 May 2008.

[Pin01] Claudio Pinhanez. Using a steerable projector and a camera to transform surfaces into interactive displays. In *CHI '01: CHI '01 extended abstracts on Human factors in computing systems*, pages 369–370. ACM, New York, NY, USA, 2001. ISBN 1-58113-340-5. doi:http://doi.acm.org/10.1145/634067.634285.

[Pla08a] Planar Systems. CoolSign Digital Signage Software. `http://www.planardigitalsignage.com`, May 2008. Last accessed 12 May 2008.

[Pla08b] PlanetLab. PlanetLab. `http://www.planet-lab.org/`, June 2008. Last accessed 02 June 2008.

[Pos80] J. Postel. User Datagram Protocol. RFC 768, Internet Engineering Task Force, August 1980.

[Pos81] J. Postel. Transmission Control Protocol. RFC 793, Internet Engineering Task Force, September 1981.

[Pro08] Prospero. The Prospero Project. `http://mblog.lib.umich.edu/prospero/`, May 2008. Last accessed 13 May 2008.

[Pyt08] Python. The Python Programming Language. `http://www.python.org`, May 2008. Last accessed 20 May 2008.

[RDO+05] Ismo Rakkolainen, Stephen DiVerdi, Alex Olwal, Nicola Candussi, Tobias Hüllerer, Markku Laitinen, Mika Piirto, and Karri Palovuori. The

interactive FogScreen. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Emerging technologies*, page 8. ACM, New York, NY, USA, 2005. doi:http://doi.acm.org/10.1145/1187297.1187306.

[RDS02] Daniel M. Russell, Clemens Drews, and Alison Sue. Social Aspects of Using Large Public Interactive Displays for Collaboration. In *UbiComp '02: Proceedings of the 4th international conference on Ubiquitous Computing*, pages 229–236. Springer-Verlag, London, UK, 2002. ISBN 3-540-44267-7.

[RG01] Daniel M. Russell and Rich Gossweiler. On the Design of Personal & Communal Large Information Scale Appliances. In *UbiComp '01: Proceedings of the 3rd international conference on Ubiquitous Computing*, pages 354–361. Springer-Verlag, London, UK, 2001. ISBN 3-540-42614-0.

[RG04] Michael Rohs and Beat Gfeller. Using Camera-Equipped Mobile Phones for Interacting with Real-World Objects. In Alois Ferscha, Horst Hoertner, and Gabriele Kotsis, editors, *Advances in Pervasive Computing*, pages 265–271. Austrian Computer Society (OCG), Vienna, Austria, April 2004. ISBN 3-85403-176-9.

[RLHS04] Josephine Reid, Mathew Lipson, Jenny Hyams, and Kate Shaw. Fancy a schmink?: a novel networked game in a café. In *ACE '04: Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 18–23. ACM, New York, NY, USA, 2004. ISBN 1-58113-882-2. doi:http://doi.acm.org/10.1145/1067343.1067345.

[RM97] Jun Rekimoto and Nobuyuki Matsushita. Perceptual Surfaces: Towards a Human and Object Sensitive Interactive Display. In *Workshop on Perceptural User Interfaces (PUI'97)*. 1997.

[RSH00] Johan Redström, Tobias Skog, and Lars Hallnäs. Informative art: using amplified artworks as information displays. In *DARE '00: Proceedings of DARE 2000 on Designing augmented reality environments*, pages 103–114. ACM, New York, NY, USA, 2000. doi:http://doi.acm.org/10.1145/354666.354677.

[RTD04] Daniel M. Russell, Jay P. Trimble, and Andreas Dieberger. The Use Patterns of Large, Interactive Display Surfaces: Case Studies of Media Design and Use for BlueBoard and MERBoard. In *HICSS '04: Proceedings of the Proceedings of*

the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 4, page 40098.2. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2056-1.

[RTW02] Daniel M. Russell, Jay P. Trimble, and Roxana Wales. Two paths from the same place: Task driven and human-centered evolution of a group information surface. In *Make IT Easy Conference*. June 2002.

[Ric07] T. Richardson. The RFB Protocol. `http://www.realvnc.com/docs/rfbproto.pdf`, June 2007. Last accessed 26 June 2008.

[Rod99] Roy Rodenstein. Employing the periphery: the window as interface. In *CHI '99: CHI '99 extended abstracts on Human factors in computing systems*, pages 204–205. ACM, New York, NY, USA, 1999. ISBN 1-58113-158-5. doi:http://doi.acm.org/10.1145/632716.632844.

[SA97] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Queensland AUUG Summer Technical Conference*. Brisbane, Australia, 1997.

[SFD06a] Oliver Storz, Adrian Friday, and Nigel Davies. Supporting content scheduling on situated public displays. In *Computers & Graphics*, volume 30, pages 681–691. Elsevier, October 2006.

[SFD+06b] Oliver Storz, Adrian Friday, Nigel Davies, Joe Finney, Corina Sas, and Jennifer Sheridan. Public Ubiquitous Computing Systems: Lessons from the e-Campus Display Deployments. *IEEE Pervasive Computing*, 05(3):40–47, 2006. ISSN 1536-1268. doi:http://doi.ieeecomputersociety.org/10.1109/MPRV.2006.56.

[SFD07] Benjamin Sherratt, Adrian Friday, and Nigel Davies. Investigating the use of Bluetooth for Location Sensing in the Implementation of Location-Based Games. Technical report, Lancaster University, July 2007.

[SG86] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Trans. Graph.*, 5(2):79–109, 1986. ISSN 0730-0301. doi:http://doi.acm.org/10.1145/22949.24053.

[SG02] Dave Snowdon and Antonietta Grasso. Diffusing information in organizational settings: learning from experience. In *CHI '02: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 331–338. ACM, New York, NY, USA, 2002. ISBN 1-58113-453-3. doi:http://doi.acm.org/10.1145/503376.503435.

[SH00] Tobias Skog and Lars Erik Holmquist. WebAware: continuous visualization of web site activity in a public space. In *CHI '00: CHI '00 extended abstracts on Human factors in computing systems*, pages 351–352. ACM, New York, NY, USA, 2000. ISBN 1-58113-248-4. doi:http://doi.acm.org/10.1145/633292.633502.

[SO05] Jürgen Scheible and Timo Ojala. MobiLenin combining a multi-track music video, personal mobile phones and a public display into multi-user interactive entertainment. In *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, pages 199–208. ACM, New York, NY, USA, 2005. ISBN 1-59593-044-2. doi:http://doi.acm.org/10.1145/1101149.1101178.

[SPK+03] Noi Sukaviriya, Mark Podlaseck, Rick Kjeldsen, Anthony Levas, Gopal Pingali, and Claudio Pinhanez. Augmenting a retail environment using steerable interactive displays. In *CHI '03: CHI '03 extended abstracts on Human factors in computing systems*, pages 978–979. ACM, New York, NY, USA, 2003. ISBN 1-58113-637-4. doi:http://doi.acm.org/10.1145/765891.766104.

[SPR+03] Norbert Streitz, Thorsten Prante, Carsten Rcker, Daniel Van Alphen, Carsten Magerkurth, Richard Stenzel, and Daniela Plewe. Ambient displays and mobile devices for the creation of social architectural spaces. In Kenton O'Hara, Mark Perry, and Elizabeth Churchill, editors, *Public and Situated Displays: Social and Interactional Aspects of Shared Display Technologies*, chapter 16, pages 387–409. Kluwer Academic Publishers, 2003.

[SSKB05] Christoph Stahl, Michael Schmitz, Antonio Krüger, and Jörg Baus. Managing Presentations in an Intelligent Environment. In *MU3I Workshop at IUI 2005*. San Diego, USA, 2005.

[SWS01] Nitin Sawhney, Sean Wheeler, and Chris Schmandt. Aware Community Portals: Shared Information Appliances for Transitional Spaces. *Personal Ubiqui-*

tous Comput., 5(1):66–70, 2001. ISSN 1617-4909. doi:http://dx.doi.org/10.1007/s007790170034.

[Sam08] Samba. Samba. `http://www.samba.org`, July 2008. Last accessed 24 July 2008.

[Sie08] Sierra Video Systems. Sierra Pro XL. `http://www.sierravideo.com/en_group_sierra_pro_xl.html`, May 2008. Last accessed 01 May 2008.

[Sig08a] Signindustry.com. LED Electronic Message Reader Boards: Watching the world go by in streaming headlines. `http://www.signindustry.com/led/articles/2003-02-28-LB-LED-Zippers.php3`, 2008. Last accessed 01 May 2008.

[Sig08b] Signindustry.com. One Times Square. `http://www.signindustry.com/led/articles/2002-05-30-LB-TimeSquareOne.php3`, May 2008. Last accessed 01 May 2008.

[Sol08] Solari. Solari Split Flap Displays. `http://www.solari.it/eng/company/research_development.html`, 2008. Last accessed 01 May 2008.

[Son08] Sony Venue Solutions. Sony Ziris Digital Signage. `http://www.venue-solutions.com/pages/3systemsforvenues/2sony_2digitalsignage.html`, May 2008. Last accessed 12 May 2008.

[Sou08] Sound & Video Contractor. The Digital Signage Installation at JFK. `http://svconline.com/digitalsignage/features/avinstall_tunnel/`, June 2008. Last accessed 26 November 2008.

[Stu08] Appliance Studio. PrintSign. `http://www.ambientweb.co.uk/sectors/smartsigns/printsign.htm`, May 2008. Last accessed 12 May 2008.

[Sun08] Sun Microsystems. The Java Media Framework API (JMF). `http://java.sun.com/javase/technologies/desktop/media/jmf/`, June 2008. Last accessed 18 June 2008.

[TM06] Goldie B. Terrell and D. Scott McCrickard. Enlightening a co-located community with a semi-public notification system. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 21–24. ACM, New York, NY, USA, 2006. ISBN 1-59593-249-6. doi:http://doi.acm.org/10.1145/1180875.1180879.

[The97] The Open Group. crontab (in The Single Unix Specification v2). http://www.opengroup.org/onlinepubs/7990989775/xcu/crontab.html, 1997. Last accessed 17 July 2008.

[The07] The Object Management Group. OMG Unified Modeling Language^{TM}(OMG UML), Superstructure, V2.1.2. Specification, The Object Management Group, November 2007.

[The08a] The Apache Software Foundation. Apache HTTP Server. http://httpd.apache.org/, July 2008. Last accessed 23 July 2008.

[The08b] The New York Yankees. Yankee Stadium History. http://mlb.mlb.com/nyy/ballpark/stadium_history.jsp, November 2008. Last accessed 26 November 2008.

[The08c] The Wall Street Journal. Mass of Messages Lands at Heathrow – New Terminal to Sport Hundreds of Ad Screens To Generate Revenue. http://online.wsj.com/article/SB120296058551467765.html, February 2008. Last accessed 26 November 2008.

[Uni08] University of Michigan. Michigan Stadium Renovation – Stadium History. http://www.umich.edu/stadium/history/, November 2008. Last accessed 26 November 2008.

[Urb08] Urban Screens. Urban Screens. http://www.urbanscreens.org/, May 2008. Last accessed 23 May 2008.

[VB04] Daniel Vogel and Ravin Balakrishnan. Interactive public ambient displays: transitioning from implicit to explicit, public to personal, interaction with multiple users. In *UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 137–146. ACM, New York, NY, USA, 2004. ISBN 1-58113-957-8. doi:http://doi.acm.org/10.1145/1029632.1029656.

[WB96] Mark Weiser and John Seely Brown. Designing Calm Technology. *PowerGrid Journal*, v 1.01, jul 1996. Also appeared as Chaper 6 - "The Coming Age of Calm Technogy" in the book "Beyond Calculation - The Next Fifty Years of Computing" by Peter J. Denning and Robert M. Metcalfe, Copernicus/An Imprint of Springer-Verlag.

[WID+98] Craig Wisneski, Hiroshi Ishii, Andrew Dahley, Matthew G. Gorbet, Scott Brave, Brygg Ullmer, and Paul Yarin. Ambient Displays: Turning Architectural Space into an Interface between People and Digital Information. In *CoBuild '98: Proceedings of the First International Workshop on Cooperative Buildings, Integrating Information, Organization, and Architecture*, pages 22–32. Springer-Verlag, London, UK, 1998. ISBN 3-540-64237-4.

[Wat08] Watch World. A Short History of Olympic Games Timing. `http://www.watchworldmagazine.com/watchworld/vol6/timekeeping5.asp`, November 2008. Last accessed 26 November 2008.

[Wei91] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):66–75, September 1991.

[Wel05] Welfare State International. Metamorphosis. `http://www.welfare-state.org/pages/projects/metamorphosis.html`, September 2005. Last accessed 28 May 2008.

[Wik08] Wikipedia. Wikipedia. `http://www.wikipedia.org/`, July 2008. Last accessed 14 July 2008.

[You08a] YouTube. YouTube. `http://www.youtube.com/`, July 2008. Last accessed 14 July 2008.

[You08b] YouTube. YouTube API. `http://code.google.com/apis/youtube/developers_guide_protocol.html`, July 2008. Last accessed 22 July 2008.

Appendix A

The High-Level Scheduling API

A.1 Playlists

Summary: A Playlist defines an orchestration of a set of content items on a set of Displays.

Attributes:

- `playlist_id`: a globally unique identifier that can be used to reference the Playlist.
- `content`: a specification of when and on which Displays individual content items are to be played in the form of a list of Playlist entries. Each entry consists of the following attributes:
  - `url`: the URL of the content item.
  - `rel_start`: the time at which the content item is to be made visible, expressed in seconds since the start of the Playlist.
  - `rel_end`: the time at which the content item is to be made not visible, expressed in seconds since the start of the Playlist.
  - `displays`: the Displays that the content item is to be shown on in the form of a list of Display identifiers. The exact semantics of this list are defined by the `sync` attribute.
  - `sync`: this argument determines the level of synchronisation that is to be applied when showing content on the Displays specified in the `displays` attribute. Three different

levels are possible:

  * *on all the specified Displays* (`all`): the content item is to be shown on all the specified Displays in a synchronised fashion.
  * *on one or more of the specified Displays* (`many`): specifies that the content item should be displayed on as many of the specified Displays as possible in a synchronised fashion.
  * *on exactly one of the specified Displays* (`one`): indicates that content is to be shown on one of the specified Displays.

A.2 Requests

Summary: Requests specify the constraints (e.g. dates, times and external conditions) that trigger a Playlist to be played.

Attributes:

- `request_id`: a globally unique identifier that can be used to reference the Request.
- `playlist_id`: the identifier of the Playlist that is associated with the Request.
- `time_conditions`: a list of date and time ranges during which the Playlist should be shown.
- `priority`: the priority of the Request.
- `repeat_spacing`: an integer value specifying the minimum temporal spacing in seconds between repetitions of the Playlist on the same Display.
- `target_total_showings`: an integer value specifying the maximum number of times the Playlist associated with the Request should be shown in the public display network.
- `target_showings_per_display`: and integer value specifying the maximum number of times that the Playlist associated with the Request should be shown on the same Display.
- `bt_presence_filter`: allows experimenters to use the presence of Bluetooth devices in the vicinity of Displays as a condition for triggering the content associated with this request. The attribute is specified as a set of (`devices`, `display_id`) entries:

– `devices`: a set of Bluetooth hardware addresses.

– `display_id`: the identifier of a Display.

Each $([device_1, \ldots, device_n], display_x)$ tuple should be interpreted as "this Request should only be considered to be schedule-able if one or more of the specified Bluetooth devices are in the vicinity of $display_x$". If more than one tuple is specified, the conditions represented by the individual tuples are combined using logical "and" operators.

A.3 Operations for Authentication

A.3.1 Authenticate

Summary: Used to retrieve an authentication token that is required to access the remaining high-level API operations.

Arguments

- `email`: the user's email address.
- `password`: the user's password.

Returns

- `error`: 0 if the authentication was successful. Otherwise, this field contains an error code.
- `auth_token`: If the operation was successful, the resulting authentication token is returned as a string in this field. Otherwise, the field is empty.

A.4 Operations for Managing Playlists

A.4.1 CreatePlaylist

Summary: creates a new playlist in the repository according to the user's specifications.

Arguments

- `auth_token`: the authentication token that was returned by `Authenticate`.
- `playlist_spec`: the specification of the Playlist (see section A.1).

Returns

- `error`: 0 if the operation was carried out successfully. Otherwise, this field contains a numerical error code.
- `playlist_id`: If the operation was successful, this field contains a numerical *Playlist Identifier* that uniquely identifies the newly created Playlist in the system.

A.4.2 RetrievePlaylist

Summary: `RetrievePlaylist` is used to retrieve the specification of an existing Playlist.

Arguments

- `auth_token`: the authentication token that was returned by `Authenticate`.
- `playlist_id`: the unique Playlist Identifier of the Playlist that is to be retrieved.

Returns

- `error`: 0 if operation was successful. Otherwise, this field contains an error code.
- `playlist_spec`: if the operation was successful, this field contains the specification of the retrieved Playlist (see section A.1).

A.4.3 UpdatePlaylist

Summary: `UpdatePlaylist` is used to modify existing Playlists.

Arguments

- `auth_token`: the authentication token that was returned by `Authenticate`.
- `playlist_id`: the Playlist Identifier of the playlist that is to be modified.
- `playlist_spec`: the specification of the modified Playlist.

Returns

- `error`: 0 if the operation was successful. Otherwise, this field contains an error code.

A.4.4 DeletePlaylist

Summary: used to delete an existing Playlist.

Arguments

- `auth_token`: the authentication token that was returned by `Authenticate`.
- `playlist_id`: the unique Playlist Identifier of the Playlist that is to be deleted.

Returns

- `error`: 0 if the operation was successful. Otherwise, this field contains an error code.

A.4.5 ListPlaylists

Summary: this operation provides the means for searching for Playlists matching certain criteria.

Arguments

- `auth_token`: the authentication token that was returned by `Authenticate`.

- `with_urls`: (optional) only Playlists containing one or more of the specified urls are returned.
- `with_display_conditions`: (optional) a list of Display Identifiers. Only playlists are returned whose `display_conditions` contain one or more of these Display Identifiers.
- `my_playlists_only`: (optional) if this flag is present and set, the search is limited to playlists that are owned by the user who issued the `ListPlaylists` request.

Returns

- `error`: 0 if the operation was successful. Otherwise, this field contains an error code.
- `playlist_ids`: a list of Playlist Identifiers of Playlists matching the search criteria.

A.5 Operations for Managing Requests

A.5.1 CreateRequest

Summary: this operation provides the means for creating new Request entries in the repository.

Arguments

- `auth_token`: the authentication token that was returned by `Authenticate`.
- `request_spec`: the specification of the Request, as outlined in section A.2.

Returns

- `error`: 0 if the operation was successful. Otherwise, this field contains an error code.
- `request_id`: if the operation was successful, this return argument contains the Request Identifier of the newly created Request entry.

A.5.2 RetrieveRequest

Summary: this operation provides the means for retrieving the specification of an existing Request.

Arguments

- `auth_token`: the authentication token that was returned by `Authenticate`.
- `request_id`: the identifier of the Request that is to be retrieved.

Returns

- `error`: 0 if the operation was successful. Otherwise, this field contains an error code.
- `request_spec`: the specification of the Request.

A.5.3 UpdateRequest

Summary: this operation allows users to modify existing Request entries.

Arguments

- `auth_token`: the authentication token that was returned by `Authenticate`.
- `request_id`: the identifier of the Request that is to be modified.
- `request_spec`: the specification of the modified Request.

Returns

- `error`: 0 if the operation was successful. Otherwise, this field contains an error code.

A.5.4 DeleteRequest

Summary: `DeleteRequest` allows users to delete Requests from the repository.

Arguments

- `auth_token`: the authentication token that was returned by `Authenticate`.
- `request_id`: the identifier of the Request entry that is to be deleted.

Returns

- `error`: 0 if the operation was successful. Otherwise, this field contains an error code.

A.5.5 ListRequests

Summary: this operation provides the means for searching for Requests according to the specified criteria.

Arguments

- `auth_token`: the authentication token that was returned by `Authenticate`.
- `with_playlist_ids`: (optional) a set of Playlist identifiers causing only those Requests to be returned that are are associated with one of the specified Playlists.
- `with_time_conditions`: (optional) a set of date and time ranges, causing only those Requests to be returned whose date and time ranges intersect one of these ranges.
- `with_priorities`: (optional) a set of priority values, causing only those Requests to be returned whose priority constraint is equal to one of the priorities in the set.
- `with_bt_presence_filter_devices`: (optional) a set of Bluetooth hardware addresses, causing only those Requests to be returned whose `bt_presence_filter` attribute contains one or more of the hardware addresses listed in `with_bt_presence_filter_devices`.

- `with_bt_presence_filter_displays`: (optional) a set of Display Identifiers, causing only those Requests to be returned whose `bt_presence_filter` attribute contains one or more of the Display Identifiers listed in `with_bt_presence_filter_displays`.
- `my_requests_only`: (optional) if this flag is present and set, the search is limited to Requests that are owned by the user who issued the `ListRequests` request.

Returns

- `error`: 0 if the operation was successful. Otherwise, this field contains an error code.
- `request_ids`: if the operation was successful, a list of identifiers of those Requests matching the search criteria is returned.