MAGIC Broker: A Middleware Toolkit for Interactive Public Displays

Aiman Erbad[1], Mike Blackstock[1], Adrian Friday[2], Rodger Lea[1], Jalal Al-Muhtadi[3]

[1]*University of British Columbia, Vancouver, Canada*
{aerbad@cs, michael@cs, rodgerl@ece}.ubc.ca
[2]*Lancaster University, Lancaster, UK*
adrian@comp.lancs.ac.uk
[3]*King Saud University, Riyadh, Saudi Arabia*
jalal@ccis.ksu.edu.sa

Abstract

Large screen displays are being increasingly deployed in public areas for advertising, entertainment, and information display. Recently we have witnessed increasing interest in supporting interaction with such displays using personal mobile devices. To enable the rapid development of public large screen interactive applications, we have designed and developed the MAGIC Broker. The MAGIC Broker provides a set of abstractions and a simple RESTful web services protocol to easily program interactive public large screen display applications with a focus on mobile device interactions. We have carried out a preliminary evaluation of the MAGIC Broker via the development of a number of prototypes and believe our toolkit is a valid first step in developing a generic support infrastructure to empower developers of interactive large screen display applications.

1. Introduction

Today, we see the widespread deployment of large screen displays in public places such as airports, shopping malls, university campuses, train/bus stations and outdoor areas. These screens are predominantly used for the display of non-interactive content such as advertising and public information such as flight schedules or upcoming events. However, we are starting to witness an increasing interest in exploring interactions with these displays through the use of various personal devices, cameras, controllers and infrastructures [1] and text messaging [2].

Our previous work deploying ubicomp infrastructure using interactive displays [3, 4] reveals that existing middleware infrastructures are either too heavyweight to be used to support spontaneous interactions with public displays, or are not sufficiently flexible to support the various interaction patterns required by multiple interactive applications. Based on lessons learned from our prior deployments, we have identified five core requirements, including the need to support client-less interaction from mobile devices, and present a set of unifying abstractions that enable us to build a lightweight middleware supporting several interactive public display applications with different modes of interaction.

We have implemented the MAGIC Broker middleware to meet these identified requirements and abstractions. In the following sections we discuss the key design factors behind the MAGIC Broker, describe the underlying abstractions and protocol, present some prototype applications that are built using the middleware and provide preliminary evaluations of the middleware.

2. Key Observations and Design Factors

One key abstraction used in interactive systems is the use of events. Event-based systems deliver three main advantages: (1) event-based systems are very suitable for highly interactive systems [5, 6], and therefore, are commonly used for programming GUI applications and synchronous groupware applications, etc. (2) they provide a high level of flexibility, where the flow of the application is controlled by events rather than a sequential program, and (3) event-based systems provide higher robustness due to their loosely-coupled design, where event sources are decoupled

1-4244-0674-9/06/$20.00

from event consumers making the system less sensitive to the order of actions and more resilient toward failures. This abstraction has been exposed by various systems [7, 8], and often involves event brokers that decouple event consumers (such as display applications) and producers (such as user interaction devices) in a distributed system and allows events (messages) sent through these systems to be intercepted and transformed as needed [9]. Despite the apparent benefits of publish-subscribe event brokers, we believe that relying on this model alone can be limiting. For example, in scenarios that require fast response or spontaneous interactions, an application or a device joining the environment will be unable to receive a complete update of the current state of the environment immediately; instead, the application/device must wait until it receives relevant events to reconstruct the state of the environment gradually. For this reason, the event-based model must be augmented with a mechanism that allows applications and devices to retrieve state instantaneously to restart where they left off and support more advanced state and event interaction patterns.

Secondly, results from earlier deployments demonstrate the importance of relieving end users from the hassle and frustration of installing client software or applets on their personal mobile devices, i.e., the system should support client-less devices. We find this essential for several reasons, (1) most users are unable or hesitant to install custom software on their handsets; (2) the high device heterogeneity makes it difficult to write custom software that works on all device types; and (3) client-less devices facilitate instant interactions by the public without the need for an enrolment process. On their own, mobile phones support interactions and content exchange using SMS, MMS, Bluetooth, WiFi, and voice by making a voice call to a Voice XML [10] gateway.

Thirdly, the middleware must be lightweight and able to support control-flow interoperability across heterogeneous clients under different administrative and network domains. For this reason, we advocate the use of a lightweight, domain specific HTTP-based protocol [11] for communication between the various components of the system. We choose a web derived protocol to increase portability, to leverage browser, JavaScript and modern programming language support for web protocols and to ease communications through firewalls and proxies.

Fourthly, lessons learned from previous deployments clearly demonstrate the need for high-quality content [3]; hence, the system should provide easy and seamless tools for content providers and developers. We observe that web-oriented tools and standards, such as HTML, JavaScript, Flash, PHP, Java applets and Java servlets facilitate the creation of high-quality content as many content providers and developers are familiar with such tools.

Finally, our experiences have demonstrated the need for a well-structured namespace that facilitates flexible groupings of devices according to the current context and applications. This becomes increasingly important as the number of displays, users and applications increases beyond single display deployments.

3. Abstractions and protocol

In this section, we introduce our abstractions and protocol using examples from a scenario we have implemented in our MashUp prototype described in Section 4. In this scenario, several situated public displays are set up in a campus environment and serve both as public message boards where students can post messages to specific locations or interactive maps where students can walk up to and request directions to different locations around the campus. At each location there are two displays. One display is used to display the newest messages posted to the location, while the other is used to display location-relevant content supplied by the university.

3.1. Abstractions

The abstractions supported by MAGIC Broker are based on the core group of abstractions of the Ubicomp Common Model [4] and the key abstractions required for interaction applications [12]. The first abstraction we describe is the *channel* used as an address and container for the other MAGIC Broker facilities.

Channels: In our scenario, users interact with groups of screens in locations. To address groups of situated displays, individual screens, users, and the functionality supported by these screens, we use the notion of a *channel*. Channels typically correspond to physical entities or groupings of entities. The MAGIC Broker allows developers to group channels in a parent-child or containment hierarchy. Channel hierarchies can be used to contain displays in specific locations or users in groups as illustrated in Figure 1. When using the middleware API, channel parent-child relationships are specified using a period "." notation. For example, the channel *where.magic.map* parent channel is *where.magic*.

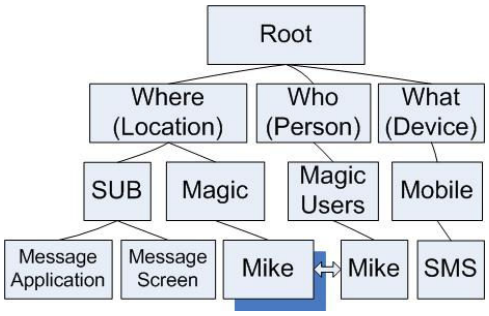

Figure 1. Scenario Channel hierarchy. The shadowed "Mike" box represents an alias.

Channels may live in more than one hierarchy using a channel *alias*. This can be used to have channels representing users to live under a *location* channel hierarchy and under a *user-group* hierarchy at the same time. For example, an alias to *who.magic-users.mike* may be called *where.magic.mike*. The organization of channels into *who* or *where* hierarchies in our prototypes are inspired by the directory organization employed by Plan B [13], but any channel organization is supported by the Broker.

Events: The backbone of the system is a publish-subscribe event broker which decouples event sources from sinks. Events sent to a channel are received by subscribers registered to that channel. Events are also sent to parent channels in the channel hierarchy, both to parents of the real channel, and parents of an aliased channel so that subscribers will receive events sent to the channel it subscribed to directly, or any of its children. In the MashUp prototype, the message screen at the Student Union Building (SUB) subscribes to *where.sub.message*, while the map screen subscribes to *where.sub.map* for example. SMS messages are sent to these channels by the SMS gateway.

State: While events are important for interactive applications, the use of persistent state is also important [12]. Applications often need to store and retrieve the current state of the system, to continue an interaction where it left off, retrieve the last object selected on the screen for example. In the MAGIC Broker, we use channels not only as an address or topic for events, but also as a container for relevant interaction state. For example, we store the last 16 messages received in the *where.sub.message* channel. In another application where users can search and download photos, we use it to store the current image selection. State can also be used to store contextual information such as the current location of the user mike in the who.magic-users.mike channel. State in our system is an untyped name/value pair that is inherited by child channels in a similar manner to class inheritance hierarchies. A state value stored in the *where.sub* channel can be retrieved by accessing *where.sub.message* if it is not overridden.

Services: Services are another important abstraction supported by the Broker. Services are a way to support synchronous RPC-style two-way interactions with a service hosted outside the Broker. A service can be used to request directions from Google or perform a Flickr photo search for example. Services are contained in a channel and they use a similar design pattern to event subscribers, except that a service must be addressed directly by name.

Content: Finally, the Broker supports storage and retrieval of *content* such as images, videos, text, and HTML documents within a channel. This allows application developers to store content associated with the entities addressed using channels. In our MashUp prototype for example, we store photos of users in the *who* channel hierarchy, and images used for interactive displays within the *where* channel hierarchy.

3.2. Protocol

To support heterogeneous clients, cross domain interaction and web-oriented application design, we employed a web service protocol. However, based on our experience [4], conventional Web Service standards using SOAP [14] are fairly heavyweight, and impractical for use in interactive applications. This led us to the use of a simple HTTP-based protocol loosely referred to as a *RESTful* web service, since it leverages the Representational State Transfer (REST) architectural style introduced by Roy Fielding [11] and is used by the World Wide Web [15]. Like SOAP-based web services, RESTful web services allow heterogeneous clients developed using any language to interoperate; however, since no standard messaging layer like SOAP is used, clients can make use of RESTful services using only HTTP without the need to install specialized web services client libraries. Moreover, it is easier to program and produce content for large screen display applications using web-oriented tools, such as HTML, JavaScript, and Flash.

In a RESTful interface, resources such as Broker channels, state or services are identified by a URL. All interactions between a client and a web service are done with four simple HTTP operations: GET to retrieve information, PUT to create new information, POST to update existing information, and DELETE to delete information.

The REST protocol used by the MAGIC Broker is summarized in Table 1. Each one of our abstractions uses the four HTTP operations to perform the required task. To send events between publishers and

subscribers, for example, publishers use an HTTP POST method to send an event to a channel. A subscriber behind a firewall can then retrieve events using an HTTP GET request.

Table 1. The REST protocol

HTTP Commands / Abstraction	GET (Read/ Copy)	PUT (Create)	POST (update)	DELETE (Cut)
Subscribe	List channel subscribers	Subscribe to a channel		Deletes subscriber from channel.
Event	Return next events	Send an event		N/A
Service	Return service description	Call service and wait for a response		N/A
State	Get all variable value/ List	Create the state in the current channel	Update an existing state	Remove a variable from channel
Registry	Get registered services list	Register service		Remove service registration
Content	Get files/file	Upload content		Remove content
Channel	Get channel info	Create channel		Delete channel

The current implementation of the Broker supports three modes of mobile phone interaction "out of the box": SMS, Voice XML, and mobile web browsers as shown in Figure 2. The workstation driving the large screen application subscribes to events sent to the appropriate channels, and begins polling for events using a HTTP GET request. When the user POSTs a form from his phone web browser to the screen channel, the event is relayed by the Broker to the waiting screen which processes it by displaying a message. Note how the system does not require mobile handsets to download any client software, it only utilizes the available capabilities in the mobile phones.

4. Prototype Applications

We developed a number of prototype applications, some as part of the Lancaster eCampus deployment [10] and others as part of the UBC Ubicomp testbed. For this paper we focus on the UBC experience where we have integrated 3 different large screen applications into the Campus MashUp prototype. The MashUp has three modes of interaction:

- **Non-interactive content and message board**. In this mode, the screen displays non-interactive content such as advertising or other public information along with posted messages at the screen bottom.
- **Image search**. In this mode, the system displays images that have been requested by users by sending a specific tag on the mobile phones.
- **Map and directions**. In this mode, a campus map and directions are displayed using Google Maps on both the large screen, and when a mobile web browser is used, on the mobile device.

Users can interact with the large screen by voice by calling a Voice XML gateway, by sending SMS messages, or using a mobile web browser. The Mash-Up system used two channel hierarchies. The first hierarchy is based on locations such as *magic* for the Magic Lab or *barn* for the Barn restaurant. The image search application uses *magic.flickr*, and the map application uses *magic.map*. The second channel hierarchy is used to store user information. When a new user registers, a channel is created under *users* channel so user Crystal gets the *users.Crystal* channel.

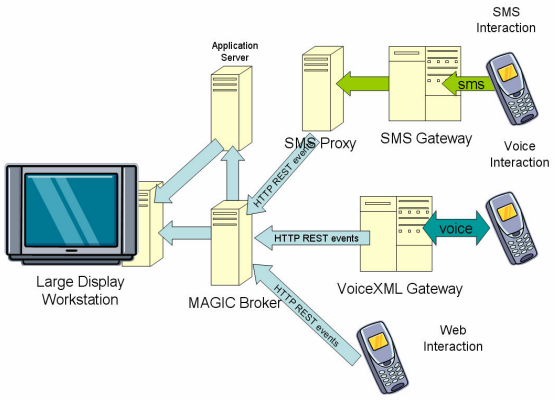

Figure 2. MAGIC Broker event flow

Events are used to send information between the user's mobile device via SMS, the VoiceXML gateway, or the mobile browser directly to the large screen display channels. *State* is used to store user information, such as name, phone number, and a picture URL. User channels contain the actual photo *content* so that they are displayed next to posted messages.

5. Preliminary Evaluation

In this section, we briefly evaluate the MAGIC Broker in terms of performance and the application developers experience to get an early measure on how easy our platform is to learn and use.

5.1 Performance

The use of a centralized design in the MAGIC Broker raises a scalability concern especially since we target interactive applications. To test the scalability of our system, we measured the latency of event delivery against event throughput. The experiment was performed in a local area network using two PCs with the following specifications: dual core 2.8 GHz

Pentium 4 with 1 GB of RAM under Windows XP Pro. One PC was used for the clients (sources/sinks of events) and the other for the MAGIC Broker server. As Figure 3 illustrates, the MAGIC Broker event delivery had an end-to-end latency below 100 milliseconds as we increased the throughput to up to 350 events per second, meeting the timing requirements of interactive applications [16].

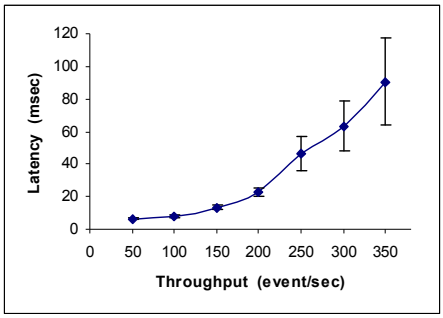

Figure 3. Latency versus event throughput.

5.2 Qualitative Evaluation

To evaluate how easy or difficult is it to learn and then use the MAGIC Broker abstractions and RESTful protocol we created a questionnaire and interviewed three prototype developers from our lab who were not involved in implementing the MAGIC Broker. Two developers had previous experience with web programming but were not experts and one developer had no previous experience. Participants were asked to report their experience using a 5-point Likert scale.

Two participants found it fair (3/5) to learn the REST protocol and the third found it easy (4/5). Once the developers were familiar with the MAGIC Broker model, all of them found that developing the prototype was "easy" (4/5): "Once I learned how everything worked (what needed to be sent and what I should receive back) it was very easy". Similarly, the three main abstractions used in the majority of the prototypes (event, state, channel) were found easy to understand (SD=0.9).

The MAGIC Broker was used as a fast prototyping tool. The developers reported the following times: 1.5, 2.5, 3 weeks to develop the first working version of an application. The two developers with previous web development experience agreed that the MAGIC Broker made it easier to develop their prototypes in comparison to other approaches, such as other event-based systems, and SOAP web services.

6. Related Work

Interactive public display applications such as the Notification Collage [17] and the Opinionizer [18] have been used to study the design challenges facing end-user interactions. The focus of our work is to empower application developers with a toolkit to facilitate building such interactive applications quickly and easily.

The iROS project [7] addressed the unique requirements of interactive environments by extending a tuple space coordination model using the Event Heap. Gaia [8], and One.World [19] also expose events as a key abstraction for coordination and user interaction. The MAGIC Broker extends the basic event abstraction with centralized and persistent interaction state stored in channels, provides support for larger multi-site deployments, and uses a wide area (HTTP) RESTful protocol. Unlike the MAGIC Broker, the e-Campus [3] infrastructure focuses on managing and scheduling the presentation of content for large networks of screens rather than interaction.

The Notification Server [6] supports events (notifications) and maintains shared state for use in interactive groupware systems. It provides a good example of how the event and state abstractions together fulfill the requirements of interactive systems such as groupware applications.

The use of a hierarchical organization of channels was inspired by Plan B [13]. Plan B uses standard file directories stored in a hierarchy to organize users, locations, and devices and a file interface, not a web services interface for cross domain interaction. One.World also organizes container abstractions called *environments* in a hierarchy. Similar to channels in the MAGIC Broker, the hierarchy eases intermediation and monitoring of events.

7. Conclusion and Future Work

In this paper, we have demonstrated that the MAGIC Broker is a promising toolkit that meets the requirements of public interactive large screen display applications. The MAGIC Broker incorporates the following novel aspects: addressable channels as both event topics and state containers, the use of channel hierarchies to organize event propagation and state inheritance, a REST web service protocol for cross domain interaction, native support for client-less mobile device interactions. The MAGIC Broker was found to be useful in supporting a wide range of interactive display applications and interaction modes. Our preliminary evaluation shows that the abstractions

were easily understood and the REST protocol for rapid prototyping. Moreover, the MAGIC Broker scales well in terms of latency under high event throughput meeting the timing requirements of interactive applications.

To further evaluate our abstractions and protocol, we are building an interactive campus guide application that allows users to interact using Voice XML and SMS as a part of the eCampus deployment. Moreover, we are developing a security model to deal with data security, and privacy in usable fashion using the proposed abstractions.

8. References

[1] Z. Rodgers. "Nike iD Billboard Invites Mobile Users," September 17, 2007; http://www.clickz.com/showPage.html?page=3502186.

[2] "Blinkenlights," September 17, 2007; http://www.blinkenlights.de/.

[3] O. Storz, A. Friday, N. Finney *et al.*, "Public ubiquitous computing systems: Lessons from the e-campus display deployments," *IEEE Pervasive Computing,* vol. 5, no. 3, pp. 40-47, 2006.

[4] M. Blackstock, R. Lea, and C. Krasic, "Toward Wide Area Interaction with Ubiquitous Computing Environments," in 1st European Conference on Smart Sensing and Context (EuroSSC), Enschede, The Netherlands, 2006.

[5] "The Swing Tutorial," September 17 2007; http://java.sun.com/docs/books/tutorial/uiswing/index.html.

[6] J. F. Patterson, M. Day, and J. Kucan, "Notification servers for synchronous groupware," in ACM CSCW'96 Conference on Computer Supported Cooperative Work, Boston, Mass., 1996.

[7] S. R. Ponnekantia, B. Johanson, E. Kiciman *et al.*, "Portability, extensibility and robustness in iROS," in Proceedings of IEEE International Conference on Pervasive Computing and Communications, Dallas-Fort Wirth, 2003.

[8] M. Roman, C. Hess, R. Cerqueira *et al.*, "Gaia: a middleware platform for active spaces," *SIGMOBILE Mob. Comput. Commun. Rev.,* vol. 6, no. 4, pp. 65-67, 2002.

[9] R. Ballagas, A. Szybalski, and A. Fox, "Patch Panel: Enabling Control-Flow Interoperability in Ubicomp Environments."

[10] "VoiceXML Forum," http://www.voicexml.org/.

[11] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Information and Computer Science, UC Irvine, Irvine, CA, 2000.

[12] A. Dix, "Status and Events: Static and Dynamic Properties of Interactive Systems," in Eurographics Seminar: Formal Methods in Computer Graphics, Marina di Carrara, Italy, 1991.

[13] F. J. Ballesteros, E. Soriano, K. Leal *et al.*, *Plan B: An Operating System for Ubiquitous Computing Environments*: IEEE Computer Society, 2006.

[14] "SOAP Version 1.2 Part 0: Primer," January 27, 2006, 2006; http://www.w3.org/TR/2003/REC-soap12-part0-20030624/.

[15] "Representational State Transfer," September 18, 2007; http://en.wikipedia.org/wiki/Representational_State_Transfer.

[16] R. Miller, "Response Time in Man-Computer Conversational Transactions," in AFIPS Fall Joint Computer Conference, 1968.

[17] S. Greenberg, and M. Rounding, "The Notification Collage: Posting Information to Public Displays in Public Spaces," in SIGCHI conference on Human factors in computing systems, Seattle WA, 2001.

[18] H. Brignull, and Y. Rogers, "Enticing People to Interact with Large Public Displays in Public Spaces," in Interact '03, 2003.

[19] R. Grimm, J. Davis, E. Lemar *et al.*, "System support for pervasive applications," *ACM Transactions on Computer Systems,* vol. 22, no. 4, pp. 421-486, November 2004, 2004.