Event Broadcasting Service
An Event-Based Communication Infrastructure

Gerrit Kahl, Christian Bürckert, Lübomira Spassova
DFKI
Affiliation
{*forename.surname*}@dfki.de

Tim Schwartz
Saarland University
Affiliation
schwartz@cs.uni-saarland.de

ABSTRACT
In this paper, we present the Event Broadcasting Service (EBS), which can be used in instrumented environments to exchange data between different devices and services. The EBS can be used with almost all operating systems and programming languages. It enables real-time exchange of large data sets and provides effective debugging tools. Example applications for smart spaces, such as a visualization dashboard and location observation, are presented.

Author Keywords
event broadcasting, platform-independent communication

ACM Classification Keywords
H.5.2 Information Interfaces and Presentation: Miscellaneous

General Terms
Design, Management, Measurement

INTRODUCTION
Nowadays, a variety of sensors have been installed in more and more public areas, which therefore can be regarded as instrumented environments. For example, retailers are increasingly embedding technology into their supermarkets in order to improve the shopping experience of their customers and support them in their shopping process. Moreover, customers can also act as human sensors and send current locations detected by their smartphones to the supermarket's server. This collected data could be useful for the shop manager, e.g. to send more employees to the cash points if needed. The collected data could also be deployed in other systems, for example, to personalize the content of assistant systems. To realize these goals, all information should be transmitted and received by different systems in real time. In order to manage the huge data exchange, an appropriate service has to be implemented in the instrumented environment. Due to the fact that the sensor systems in such an instrumented environment are often implemented in different programming languages, this service should provide an easy to use generic interface.

IUI Workshop on Location Awareness for Mixed and Dual Reality LAMDa12, 14, February 2012, Lisbon, Portugal.

In this paper, we propose an event-based communication structure, which is originally implemented in Java, but also offers a web interface such that events can be generated by simply calling a web page. All events are serialized to an XML representation of the corresponding class and then transmitted to a communication server. Other systems can register at this server for different types of events and are notified whenever an event is sent to the server. Afterwards, the events can be deserialized by a native Java deserializer, which can also be used on Android. In order to receive messages in other programming languages, a corresponding deserializer has to be implemented.

RELATED WORK
Johanson et al. developed a system called *iROS* (*interactive Room Operating System*), which can be used to transmit data between processes [1]. This system has been designed as middleware and used in an interactive workspace (*iRoom*) where multiple ubiquitous computing devices are connected to each other in order to help people coming together for collaborations. One part of iROS is the *EventHeap* [3], a communication server which is used by the computing devices in the iRoom to communicate in a client-server-architecture manner. Therefore, the data is packed into *events* which are serialized and transmitted through the server. A client registers at the EventHeap for the events it wants to receive. Furthermore, a TTL (Time-To-Live) can be set, so that clients who will register later can request older events from the heap. According to [3], there are Java, C++, and web service implementations, so that client processes in different languages can connect to the server implemented in Java. The authors mention that iROS is "friendly to existing languages and environments, and straightforward to support a wide range of devices and leverage their existing application bases." Unfortunately, we could not find any newer documentation, which leads to the conclusion that the development has stopped after the publication of the paper. Also the current documentation and sources are not available any longer, hence we could not validate this system using Android devices for transmitting and receiving events.

Metaglue from the Stanford University is a distributed multi-agent system for intelligent environments implemented in Java. It shall meet the requirement to handle large numbers of hardware and software components' interconnection [2]. It is possible to establish communication channels between single agents and maintain their state. It is also created to introduce and modify agents in a running system and to manage

shared resources. Furthermore, an event broadcasting mechanism is part of its capabilities. To maintain the configuration and work on the running system, Metaglue has an SQL database and a web-based interface for modifying the agent's configuration-attributes at runtime. The agents make requests in an ad-hoc direct way and can also use event broadcasting to notify groups of agents about context-shifts in room applications. For debugging, a so called "Catalog monitor" is presented which displays all running agents and their reliance interconnections. Still, the authors admit that debugging is difficult, and they hope for the Java community to make more steps into the direction of distributed agent debugging [2].

In [6], two frameworks are presented: *Prism-SF*, an architectural style framework, and *Prism-MW*, an architectural middleware framework. These frameworks can be used to describe network architectures and then work on top of this to let different components communicate with each other. According to the authors, the frameworks can be used in many types of distributed systems like client-server or peer-to-peer networks. Prism-SF provides design guidelines for composing large distributed, decentralized, mobile systems and Prism-MW is a lightweight architectural middleware supporting the implementation of these guidelines [6]. The Prism-MW framework is a composition of a large amount of classes that can be extended and connected to create large distributed systems in which all components have the ability to communicate. They make use of Java's dynamic class loading and DLLs under Windows to add and remove communication ports at runtime. The authors claim that more than a dozen applications have been designed using various instances of Prism-SF, implemented on top of Prism-MW. The system supports PalmOS, WindowsCE and desktop platforms and even digital cameras and motion sensors. However, there is no support for Android devices.

Since all of these system are either very complex to install and configure (Prism and Metaglue) or are not available for modern devices like Android (iROS, Prism), we had to develop a new approach. This Event Broadcasting Service (EBS) aims at being easy to configure, to debug, and to extend.

EVENT BROADCASTING SERVICE

In order to enable an interconnection between different services in an instrumented environment, we developed an event-based communication infrastructure. For the design of this service, we devised four criteria. The developed service should enable users to easily install it and adapt their applications to the infrastructure within a relatively short timeframe. Since there is a variety of different services and sensors in instrumented environments, the communication architecture should offer interfaces for different programming languages and operating systems, especially for embedded and portable ones. In order to cope with a wide range of different applications, the communication interfaces have to be generic. Furthermore, the event broadcasting between several services should be guaranteed to work in nearly real time. Due to the huge amount of data that will be sent by several systems, a simple and easy to use debugging mechanism should be provided natively.

These requirements can be summed up by the following keywords:

1. Simplicity
2. Portability
3. Flexibility
4. Simple Debugging

In order to fulfill our predefined criteria, we decided to implement our infrastructure as an *Event Broadcasting Service* (EBS). Hence, only one server is needed, to which all clients can connect using web sockets and to which they can send events. These events are broadcast to every client connected to the server and filtered on the client side. The corresponding architecture is illustrated in Figure 1.

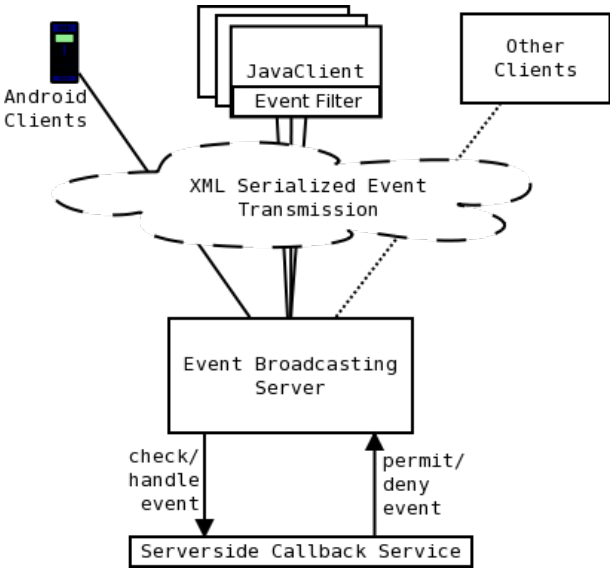

Figure 1. Architecture of the EBS

The EBS server is implemented in Java in order to be independent from the operating system. For the events, we decided to use an XML serialization before transmitting them as simple strings. Due to this serialization, events can be generated and parsed in any programming language supporting XML parsers. For example, Java already implements such a serializer and deserializer that can also be used with Android, namely the *xstream* library[1]. To also enable sensors to send events, the EBS additionally offers a web interface for generating and sending events with primitive datatypes. For this purpose, a URL containing the event name and parameters as key-value pairs has to be called. In contrast to the iROS EventHeap, we obtain the corresponding Java instance after deserialization, where all parameters and functions can be called.

[1] `http://xstream.codehaus.org/`

In order to simplify the debugging of the system, precise runtime exceptions are thrown including stack traces. For instance if the method for a *HelloWorldEvent*, which a client has registered for, is missing, the system outputs the error message "*You forgot to implement onEvent(HelloWorldEvent event) in your callback class HelloWorld*". To make runtime debugging more comfortable and efficient, a *DebugEventClient* can be connected, which simply outputs the XML string of every broadcast event. Every event contains an ID defined by the programmer, which makes it possible to infer its origin. All clients automatically send *HelloEvents* at regular intervals, containing the origin ID and a list of all events they are currently listening to, which makes it possible to recognize if clients are available or blocked. Development and debugging processes are further simplified by the opportunity to specify an auto reconnect. This means if the server is lost, a client blocks its send operations until the server is available again, while informing regularly about its own connection state. This makes it possible to restart the server and the clients in a running environment.

Implementation Details

In the currently available Java and Android implementations, all events are extensions of the class *Event*. Events can theoretically be very complex data structures composed of various other classes. To simplify the usage of the EBS, we decided to limit the client's functionality to a minimum, implementing only the following methods: *connecting*, *event transmitting*, and *event receiving*. In order to establish a connection, a client has to be created by defining its event receiving handler (callback) and a set of events it listens to. Afterwards, events can be transmitted by calling the *send* method, which takes an instance of an event class that is to be broadcasted. The event received on the server side will be automatically passed to the callback instances of the connected clients. The latter must implement a special method for every event they have registered for, which defines how a specific event is to be handled. These methods must be named *onEvent* and must take one argument of the corresponding event's class type. For example, a client listening to a *HelloWorldEvent* must provide the method *public void onEvent(HelloWorldEvent event)* in its own callback class.

To make the system even more flexible, the server can also be extended by a callback method, which is guaranteed to be executed before broadcasting a detected event and thus allows to permit or deny the broadcasting of specific events (filter). Some advantages of this extension are discussed in the following section.

Deployment in an Instrumented Shopping Environment

The Innovative Retail Laboratory (IRL) [8] is a small experimental instrumented retail environment, in which modern shopping assistance systems are developed and tested. In this environment, we have to face the problem of many different systems (mobile and embedded systems, servers, etc.) running all kinds of operating systems, e.g. Windows, Android, Mac OS and Linux. Since all components should be interconnected, a middleware had to be developed that allows the different systems to interact. Furthermore, all products of the instrumented environment operate on a centralized database. One of the challenges to be handled in this context was the need to inform all systems in the environment of possible changes, e.g. the re-location of an object or person.

Apart from the transmission of data from sensors and systems to other systems, the current global state of the instrumented environment needs to be observed. For this purpose, we decided to set up a database containing all information concerning the current state of the environment, e.g. the positions of all objects. In order to keep this data coherent, we use the aforementioned events to update the database. Since the changes of the database are mainly simple transactions, we outsourced them to a centralized service and included it into the core EBS server. All events sent to the EBS server are directed to the *Synchronization Service* (*SyncService*) before being broadcast. An event received by the SyncService triggers an update of the database and is then forwarded to the broadcasting algorithm or denied depending on the current filter options. This guarantees that the clients are informed about a change only after it has been captured in the database. The SyncService is the only component which is capable of editing the database. Since the events are broadcast, every service in our environment will be informed about changes and can get the information out of the event or the database, which is updated before the event has been broadcast.

In our opinion, the EBS infrastructure provides a suitable approach to decoupling services from their corresponding user interfaces (UI). The UIs listen to events and display changes while the backend services receive sensory data and produce appropriate events. Relevant user interactions with sensors or UI elements also result in events, which offers all listening clients the opportunity to react to them. For example, one of our systems reacts to the presence of certain objects and displays relevant information on a screen as soon as an object has been detected at a specified location and clears this information when the object is removed again. Whenever sensors detect the absence or presence of an item, a corresponding event is sent by the sensor client and received by the user interface component, which then reacts appropriately. Using this approach, graphical user interfaces can also easily be decoupled from the sensors, which facilitates the development and comparison of different UIs on different operating systems using the same sensor data. Additionally, the components can be tested beforehand, without having real sensor data, by just sending the corresponding events and hence simulating certain state changes. With this architecture, systems can react to events sent by sensors or by simulators in the same way, which offers a comfortable way to debug programs.

APPLICATION USING THE DUAL REALITY PARADIGM

Using this architecture, a great number of events can be sent by the sensors and systems of the instrumented environment. These events comprise different pieces of information including data measured by sensors or information provided by other systems. In order to provide a representation of the detected state changes, we are developing a component aiming at visualizing the current state of the instrumented environment

using a dashboard-like metaphor. Each change of the environment detected by sensors will be transmitted via events to the EBS server, which will broadcast them to this dashboard component, which itself is registered as a client. The dashboard component aims at monitoring and controlling the services I/O behavior, which can be detected in the instrumented environment. Apart from the visual representation itself, interaction in this visual representation should have an influence on the real world, e.g., if the user changes a parameter in the virtual model, the corresponding change is reflected in the real environment. This bi-directional communication from the real world to a virtual model and vice versa is referred to as *Dual Reality* [5]. The dashboard component should also be implemented as a generic interface to enable the inclusion of simulators that can influence the virtual representation but also the real world [4].

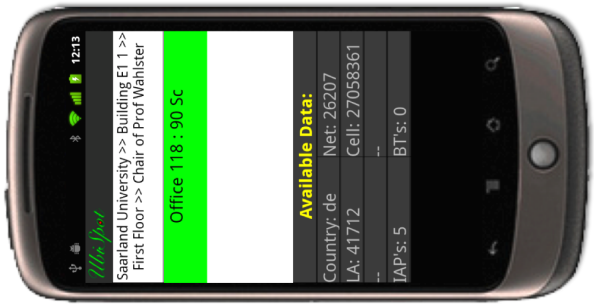

Figure 2. Smart phone running UbiSpot

In smart spaces, it is important to keep track of the locations of people in this environment. In [7], an *Always Best Positioned* system called UBISPOT is described, which uses cell towers, WiFi and Bluetooth information to estimate the current position of an object or person. Originally, this system has been implemented for SymbianOS smart phones. We reimplemented the algorithms for Android devices (a screenshot of UBISPOT for Android devices is shown in Figure 2). Using EBS, users can decide to share their anonymized locations with the infrastructure of a smart space. In this context, "anonymized" means that the identity of the person is not revealed. Still, the provided information can be helpful, for example in airports or large malls, to enable a manager to assign workers appropriately. These locations can, for example, be visualized in the previously introduced dashboard.

CONCLUSION AND FUTURE WORK

The Event Broadcasting Service provides an easy and efficient way to interconnect different services running on different devices by exchanging events. The server itself is implemented in Java, the clients, however, can run on any operation system due to the XML representation of the events. While interfaces for receiving events have just been implemented in Java, client services can be written in any programming language to send events using the web interface for event generation and transmission. Early tests of the EBS showed fast performance. In these test, clients ran on Mac OS X, Linux, Windows XP, Windows 7, and Android.

For future work, we plan to run intensive tests of the server and to further extend the EBS. One such extension will be a protocol for question answering, which enables to trace possible communication problems and further improves the stability by monitoring if queries have already been answered and optionally resending the appropriate events. Furthermore, the dashboard system will be extended to visualize further events delivered by the intelligent environment as well as usage statistics.

ACKNOWLEDGEMENT

This research was funded in part by the German Federal Ministry of Education and Research and developed in the projects EMERGENT (grant number 01 IC 10S01H) and SWINNG (grant number 01 IC 10S05A). The responsibility for this publication lies with the authors.

REFERENCES

1. Borchers, J., Ringel, M., Tyler, J., and Fox, A. Stanford Interactive Workspaces: A Framework for Physical and Graphical User Interface Prototyping. *Wireless Communications 9*, 6 (2002), 64–69.
2. Coen, M. H., Phillips, B., Warshawsky, N., Weisman, L., Peters, S., and Finin, P. Meeting the computational needs of intelligent environments: The metaglue system. In *Proceedings of MANSE'99* (1999), 201–212.
3. Johanson, B., and Fox, A. The event heap: A coordination infrastructure for interactive workspaces. In *Proceedings of the Fourth Workshop on Mobile Computing Systems and Applications*, IEEE (2002), 83–93.
4. Kahl, G., Warwas, S., PascalLiedtke, Spassova, L., and Brandherm, B. Management dashboard in a retail scenario. In *Workshop on Location Awareness in Dual and Mixed Reality. International Conference on Intelligent User Interfaces (IUI-11), February 13, Palo Alto,, California, United States*, G. Kahl, T. Schwartz, P. Nurmi, E. Dim, and A. Forsblom, Eds., Online-Proceedings (2011), 22–25.
5. Lifton, J., and Paradiso, J. A. Dual Reality: Merging the Real and Virtual. In *Proceedings of the First International ICST Conference on Facets of Virtual Environments (FaVE)* (July 2009).
6. Medvidovic, N., Mikic-Rakic, M., Mehta, N. R., and Malek, S. Software architectural support for handheld computing. *Computer 36* (September 2003), 66–73.
7. Schwartz, T., Stahl, C., Müller, C., Dimitrov, V., and Ji, H. UbiSpot – A User-Trained Always Best Positioned Engine for Mobile Phones. In *Proceedings of Ubiquitous Positioning Indoor Navigation and Location Based Services (2010)*, IEEE XPlore (2010). Online Publication, no pagenumbers.
8. Spassova, L., Schöning, J., Kahl, G., and Krüger, A. Innovative retail laboratory. In *Roots for the Future of Ambient Intelligence. European Conference on Ambient Intelligence (AmI-09), in Conjunction with 3rd, November 18-21, Salzburg, Austria*, o.A. (2009).