

USING EVENT NOTIFICATION SERVERS TO SUPPORT APPLICATION AWARENESS

Cleudson R. B. de Souza, Santhoshi D. Basaveswara, David F. Redmiles
Information and Computer Science
University of California, Irvine
Irvine, CA, USA
{cdesouza, sdb, redmiles}@ics.uci.edu

ABSTRACT

In previous work, we presented a software strategy called CASS (Cross Application Subscription Service) and an accompanying event notification server called CASSIUS (CASS Information Update Server) for supporting awareness. Roughly, awareness refers to the ability to have available, pertinent information about other activities, usually the activities of co-workers. Ongoing work raised the question of how essential the CASSIUS server was to our overall goal of supporting awareness. This present work examines that question through a twofold evaluation (theoretical and practical) of CASSIUS and the event notification servers Elvin and Siena, which are more readily available than CASSIUS. Together, these servers represent a gamut of services, from basic to extended, available among notification servers. Our interest and approach to evaluating the three servers has been very practical. We are concerned with how the services provided make the servers usable to application designers and programmers.

Keywords

Cooperative Work Support, Visualization, Event Notification Servers, Awareness

1. INTRODUCTON: EVENT NOTIFICATION SERVERS AND AWARENESS

There is a growing trend of creating distributed software applications that communicate using the Internet. Many of these applications use an event-notification or *publish-subscribe* design style [1] [13] [14]. Messages are sent not from one application directly to another, but to an event dispatcher, also called an *event notification server*. In turn, the event notification server distributes the events to the applications that *subscribed* to them. In this style of communication, there are two types of components: information *sources* and *consumers*. The information sources publish events, whereas the information consumers subscribe to particular events or categories of events. In general, a notification server usually provides a *subscription language* that allows information consumers to subscribe to different types of events by using, for example, Boolean (logical) expressions, regular expressions, sequence matching, or other.

An important advantage systems designed using event notification servers is the decoupling between the producers and consumers of the events. Decreasing coupling results in improved flexibility for the entire system. Another advantage of notification servers is the possibility of tapping into events and thus building applications that rely on monitoring events. This aspect makes it possible to build *awareness* tools.

Awareness can have many interpretations. One of the earliest definitions of awareness in computer-supported cooperative work is having information about the activities of others that affect one's work [4]. Many awareness tools, such as Portholes systems [3][11], focus only on awareness of the presence of other people and their physical activity to convey critical context. If we begin to make some distinctions among different types of awareness information, we could use the term *group awareness* for this prototypical example of awareness between human consumers. The distinction is useful for this paper because we focus on a particular scenario for evaluation purposes. In that scenario, described in the next section, awareness focuses on information about the state of an application or set of applications. Hence, for this kind of awareness, we use the term *application awareness*. Previously, we had examined how awareness of process information could support a project team [10], and we used the term *project awareness* to emphasize this information. Ideally, awareness of one's own work context would be filled with all three (and probably more) kinds of awareness information.

Although, it may seem straightforward that notification servers would support awareness, most notification servers are developed with specific goals and software contexts in mind, and thus they support only limited capabilities. For example, some CORBA orbs function only within certain operating systems and with limited services provided. In previous work, we developed a specific event notification server, CASSIUS, for supporting project and group awareness [9]. We learned that supporting these two kinds of awareness would require a number of services: (1) to provide a mechanism to locate information sources affecting users' work; (2) to isolate relevant subsets of information from sources; (3) to monitor sources and

components (subsets of sources) for changes; (4) to allow subscriptions to summaries of changes to the source or subset; (5) to send notifications of changes to a notification server and, if needed, directly to the workers, e.g., via email; and (6) to represent notifications (or summaries of sets of notifications) within awareness tools.

CASSIUS was successful in the sense that it made it easy for applications programmers to integrate awareness into different kinds of applications. It also simplified issues for mobile applications. However, questions arose. First, CASSIUS was not a commercial tool or widely available. Second, some kinds of awareness were not as complex as the kinds of project and group awareness CASSIUS supported and might be supported by servers with lesser capabilities. Third, CASSIUS provided support for interchangeability, detailed awareness information, and mobile applications. In general the challenge becomes the following: could more available and commercial servers be substituted for CASSIUS; if so, what were the trade-offs?

This present work examines this question through an evaluation of event notification servers including CASSIUS and two others more readily available, Elvin [5] and Siena [2] on a specific scenario, using limited awareness. Elvin was chosen because of its maturity and strong user community. Also, Elvin was used by its designers to provide some support for awareness. Siena is relatively new but is rapidly gaining popularity in the software engineering community because it emphasizes scalability in distributed environments. Together they are representative of the majority of the services available in commercial and some experimental environments.

The evaluation is formulated in terms of theoretical and practical considerations. The lessons from the evaluation should provide data for designers of event notification servers as well as applications designers and programmers interested in this technology.

2. AWACS EXAMPLE APPLICATION

We took as an evaluation point, a scenario we developed for a Defense Advanced Research Projects Agency (DARPA) project involving the Airborne Warning and Control System (AWACS) system simulator. This project was focused on providing application awareness. *However, our more general goal is to understand what kinds of servers and services can support all kinds of awareness.*

AWACS is a radar and computer system that provides an airborne surveillance capability with command, control, and communication functions. We used a sanitized AWACS simulator provided to UCI by Lockheed Martin with a realistic architecture. This architecture is very large, with about 300 components distributed among 30 subsystems. These components constantly exchange events

and other information with each other. Additionally, the components might have complicated dependency relationships with each other. Due to the complexity of the software it is very difficult for the designers of the software to anticipate all possible problems that could occur and hence, awareness tools that monitor runtime behavior are critical.

To provide application awareness, we developed a set of seven different visualizations called awareness gauges that present different types of information about the behavior of the AWACS simulator. The first gauge is presented as a line graph that measures the number of events that occur per poll. The second presents the same information as a bar chart. Finally, the third one presents this information as a graph that is very similar to the Xload application in X-Windows™. See Figures 1a, 1b, and 1c. These three gauges present the same data but with different visualizations. The Clock gauge indicates the progress of cycles used to synchronize the exchange of information. See Figure 1d.

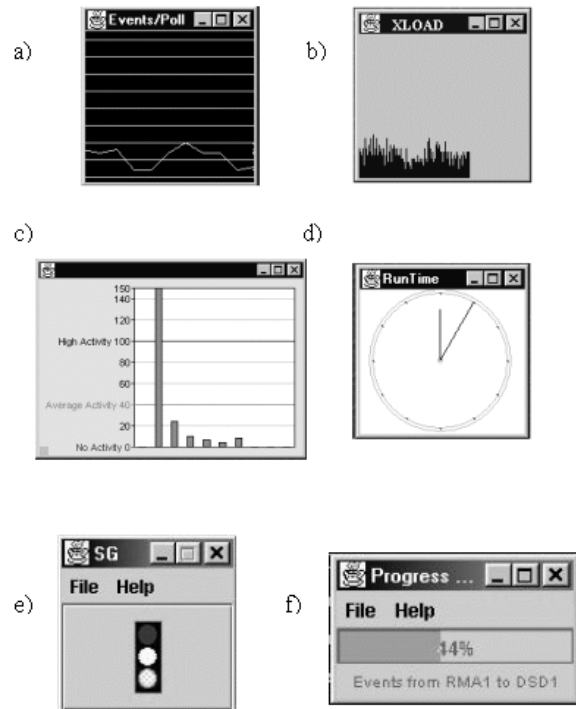


Figure 1:

- (a) The Line Graph gauge traces activity over time, emphasizing times of heavy load.
- (b) The Xload gauge also traces activity over time.
- (c) The Bar Chart gauge displays a bar chart graph with information about the activity in the application being monitored.
- (d) The Clock gauge shows how long simulation has successfully run.
- (e) The Signal gauge uses a traffic signal to indicate that the system has entered a dangerous state.
- (f) The Progress Bar gauge monitors messages passed between two components, notifying the user when the desired amount of traffic has occurred.

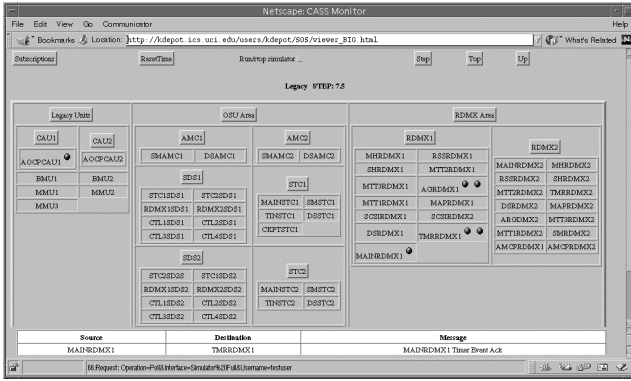


Figure 2: The Architectural gauge abstracts aspects of the software architecture of the AWACS simulator.

The Signal gauge takes the form of a traffic signal that turns red when events signal a problematic state. This state is identified as a sequence of specific events detected by using an event-matching algorithm. As well as being an alert, it may also be used for debugging. See Figure 1e. The Progress Bar gauge reflects the progress of the simulator in a particular scenario. See Figure 1f.

The Architectural gauge takes the form of a box-and-arrows (or flowchart) diagram of the components of the AWACS simulator; that is, it mimics the software architecture of the AWACS simulator. Activity in the components is presented in the form of “dots” on the respective components. It represents a selected subset of the architecture. See Figure 2.

As a design decision, we developed gauges that can be reused in different situations or conditions. For example, the Signal gauge can be used to inform a user that he or she cannot open an Internet connection with another machine, or that the user should stop modifying a file because somebody else is trying to access the same file, and so on. The unique exception is the Architectural gauge, which is specialized to represent the components of the AWACS simulator. We plan to create gauges that can be modified by the users according to their preferences and requirements.

3. LESSONS LEARNED

This section describes several lessons learned about notification servers that support application awareness based on our experience with the scenario described in the preceding section. Each lesson is organized into three parts. First, some theoretical considerations describe, for example, the general differences among the various servers’ approaches to the issue. Second, the practical considerations describe the problems that we encountered during the actual implementation and re-implementations of the scenario. Third, lessons learned entail the

implications or benefits, if any, of one approach over another.

3.1 How gauges are notified about the events, or the issue of pull versus push architectures between the notification server and the gauges

3.1.1 Theoretical Considerations. In general, an event-based system involves communication (1) between an information source and a notification server and (2) between a notification server and the information consumers (gauges in our scenario). The two types of architectures used to implement this communication are *pull* and *push*. In a pull architecture, the component interested in receiving the events is responsible for contacting the notification server to check for new events. In a push architecture, notification servers send events to components by invoking a pre-specified method that the component implements.

3.1.2 Practical Considerations. Elvin, Siena, and CASSIUS all implement a push architecture with respect to the information sources. CASSIUS also implements a pull architecture. Specifically, if the communication between the gauges and CASSIUS uses the *http* protocol to establish a connection, then this communication is performed using pull. However, if the gauges use the CASSIUS application program interface (API) to open a connection between the server and the gauge, events are sent (pushed) to the gauges as they arrive at the server.

Because the APIs for the three notification servers implement a push architecture, our implementation was made easier. Originally, we developed gauges to be used with the CASSIUS server and CASSIUS API. According to the frequency of the poll, events were pushed to the gauges. This architecture was ideal for the “performance” gauges that provided statistics about the traffic on the notification server. Porting a gauge from one notification server to the other basically involved implementing a different interface—a new set of methods. For the gauges to be attached to Elvin and Siena, they had to be modified to work with a push architecture. This required that they be equipped with a timer and a counter (to count the number of events that arrived during a certain time interval). Although this was not a substantial change, it shows that the interchangeability among notification servers can be problematic when different architectures are being used.

3.1.3 Lessons. As observed in the implementation of the CASSIUS server and its API, mixed communication models are often in use. Mix models increase the likelihood of programming errors. Also the mismatch between models and application requirements can lead to additional design and programming effort, as well as design and programming errors.

Instead of thinking of the two architectures as completely different, we rather think of push and pull as extremes of a spectrum of communication models. We envision future notification servers implementing this entire spectrum to simplify the construction of more powerful consumers. According to the requirements of the consumer, models could be customized. Moreover, the gauge could be “intelligent” enough to make dynamic changes to the architecture used according to environmental factors such as network bandwidth, location, previous events received.

3.2 How powerful the subscription service is for each notification server or the issue about the types of matching that are supported

3.2.1 Theoretical Considerations. A critical feature of notification servers is event matching. For example, a simple form of event matching is possible by using logical connectors. Usually, in this case, the matching is performed on the attributes of a single event. For example, a gauge could subscribe to the notification server by using the following logical expression:

```
[“ComponentFrom == RDMA1”] AND  
[“ComponentTo == MDRA1”]
```

Thus, all events that match this condition would be sent to the gauge. A more complex and interesting type of event matching is sequence detection, which is used to detect occurrences of concrete or abstractly defined “target” sequences within “source” sequences [7]. Sequence detection is carried out over a set of events received instead of a unique event, as in logical connectors. For example:

```
[“ComponentFrom == RDMA1”] followed by  
[“ComponentFrom == MDRA1”]
```

In response to this subscription, the gauge is notified when the notification server receives two consecutive events sent by RDMA1 and MDRA1, respectively. A more complicated type of sequence detection can be achieved if regular expressions are used. For example, the following formula describes a subscription that can detect one or more events sent by RDMA1 (represented by the symbol “+”) followed by an event sent by MDRA1:

```
[“ComponentFrom == RDMA1”]+ followed by  
[“ComponentFrom == MDRA1”]
```

Similarly, another example of sequence detection is described by the following formula:

```
[“ComponentFrom == RDMA1”] followed by  
[“ComponentFrom == *”] followed by  
[“ComponentFrom == MDRA1”]
```

This formula represents a subscription in which the first event must be sent by the component RDMA1. The following zero or more (represented by the *) events can be sent by any component, and finally, the last event must be

sent by the component MDRA1. Hilbert and Redmiles present more detailed discussion about sequence detection and examples [7].

Finally, regular expressions can also be used to match the meta information elements of a single event. For example, the following formula describes a subscription to all events being sent by components that have the first initials RDMA and one extra letter as identification. Therefore, all events being sent by RDMA1, RDMA2, and RDMA3 will be delivered to the gauge that makes this subscription:

```
[“ComponentFrom == RDMA?”]
```

The location at which event matching is performed is another factor to be considered. According to Gruber, Krishnamurthy, and Panagos [6], the choice can be any of the following: the information sources, the information consumers, or the notification server. The design decision about where to locate event matching is based on the constraints of the computational environment. In particular, if event matching is located in the consumers, it means that the gauges will receive many events, even though they are interested in only a few of them. Conversely, the matching may be fully decentralized. Namely, if the matching is performed at the notification server, the network traffic is decreased, but the server might experience scalability problems, depending on the number of consumers, generated events, and information sources [6]. Finally, if the information source processes the matching, it means that some events may not be injected into the network, thus reducing the traffic on the network. Persistence is a difficult feature to achieve in this case.

3.2.2 Practical Considerations. The subscription language provided by Elvin supports logical connectors and a simple form of regular expressions. However, it does not support any type of sequence detection. Siena supports logical and regular expressions and the simplest form of sequence detection. Finally, CASSIUS, supports logical and regular expressions, as well as sequence detection. In our scenario using CASSIUS, the Signal gauge uses complex sequence detection. However, because the other notification servers do not support this type of matching, we changed the Signal gauge’s subscription when we used Elvin and Siena.

3.2.3 Lessons. The power of the subscription service supported by the notification server determines the power of the applications that can be built using this server. On the other hand, if an information consumer needs to use a complex subscription, and this subscription is not fully supported by the subscription service, the application will need to implement extra functionality to support it. Adding extra functionality to information consumers (or sources) can be problematic since a) it decreases the flexibility of the application; and b) it increases the likelihood of errors in the application.

Furthermore, depending on the distribution of the service, adding functionality to the information source might result in a decrease in the traffic on the network. However, coupling between the information sources and consumers is increased if the information source is involved in performing the subscription service (whether hard-coded or specified).

3.3 Which objects can send events to the notification server, or issues about event types and registration

3.3.1 Theoretical Considerations. Most notification servers implement the publisher-subscriber protocol to provide basic event notification services. In a publisher-subscriber protocol, information sources publish data or events, in the form of notifications, for information consumers. The information consumers, in return, specify (subscribe to) specific data or events in which they are interested and about which they wish to be notified. Some notification servers may provide additional types of services. For example, CASSIUS provides support for the definition of event types as well as for the registration of information sources. The purpose of having a registration step is twofold. First, it provides a higher level of security because the notification server can validate the information sources that can send events to it. Second, information collected during registration can be used to create a hierarchical structure that allows navigation and selection of the objects of interest. Depending on how the registration is performed, data about the information sources and their properties can also be collected.

A notification server can also define event types. For example, a “document changed” event might have specific attributes, such as the name of the document being changed, the time the document was changed, and so on. Each of those attributes can also have a type such as String, Integer, and so forth. Event types are important because they avoid performance problems in event matching algorithms [6]. Subtypes can also be supported, allowing the reuse of types already defined. For example, a subtype declaration might simply add required or optional fields to those of its parent type [6]. However, to the best of our knowledge, no notification server uses this approach.

3.3.2 Practical Considerations. CASSIUS adopts a strategy in which the information sources need to register with the notification server before sending events [9]. They can also define object types and associate events to these types, but this is not mandatory because CASSIUS provides support for generic event types [8]. Registration is managed by the notification server, which creates a hierarchical structure of all the objects and associated events. Although this registration process may delay the process of sending

events, it has the advantage that the additional information sent by the information sources can be made available to the information consumers. For example, this process allows gauges or other information consumers to “browse” the server in order to identify objects, types, and events of interest.

In contrast, Siena or Elvin do not require the registration step. They both allow any arbitrary program to send events to them. These events can have any desired number of fields. For example, one information source can send events with two fields, such as name and age, whereas another one can send events with three fields, such as first name, last name, and date of birth. Even though this approach is direct and easy, it can create problems. For instance, the developers of the information source and the developers of the gauges must agree to a common format of the events before building an application. Otherwise, a gauge might subscribe to events that are never sent to it due to differences among details about the field name adopted or used.

3.3.3 Lessons. If a notification server supports event and object registration, and maintains a repository of objects, the information consumers are more decoupled from the information sources. Thus, they can browse the notification server for the names of objects and events instead of being hardwired to specific information sources. Likewise, notification servers that do not support event and object registration do not allow consumers (gauges) to browse potential object hierarchies and event types, limiting the power of the consumers as well increasing the coupling between information sources and consumers. Events may be missed due to name mismatches.

3.4 Which meta-information is associated with the events sent to the notification server, or How powerful the events are

3.4.1 Theoretical Considerations

We define meta-information as any additional information about an event that is recorded by the information source or notification server and is also sent along with the event to the consumer. Examples of meta-information that can be provided by a notification server include the type of event being sent, the information source that sent the event, the timestamp when the event was received by the notification server, and the timestamp when the event was sent by the information source. Ideally, the information sources and notification server should collect this information without having to make any additional efforts.

3.4.2 Practical Considerations. Using Elvin and Siena, an information source has to explicitly add meta-information along with the event. For example, a possible implementation of a program that sends events from the

AWACS simulator to Elvin can be described by the following code:

```
notif.put("ComponentFrom","RDMA1");
notif.put("ComponentTo","TDMXAR1");
notif.put("Message",
    " RDMA1 is sending a message to TDMXAR1");
notif.put("EventType","AwacsSimulatorEvent");
```

The first three lines of code describe the methods necessary to send an event produced in one component (*ComponentFrom*) to another (*ComponentTo*), and the *Message* field defines the real message being sent from one component to another. The fourth line of code describes the meta-information, which in this case is information about the type of event being sent because event types are not defined in Elvin.

In our scenario, we are interested in all the events from the AWACS simulator. Using Siena and Elvin, we could subscribe to all events. However, the problem with this approach is that other events that are not initiated in an AWACS component will be pushed to the gauges. To avoid this problem, we added meta-information that classified the type of the event being sent.

Meta-information about the type of the events is directly supported in CASSIUS. Furthermore, it is enforced; that is, no event can be sent without this information, so every information source has to also explicitly add this information to its events. Also, the CASSIUS server validates this information before forwarding the event to the gauges, thus it has an additional level of security.

3.4.3 Lessons. Several advantages can be achieved if a notification server supports event meta-information. First, the notification server can use this additional information to provide consistency checks based on event and object types. Second, there is no mixing between the information about an event and meta-information about the same event. This clear separation of information and meta-information reduces the likelihood of errors and makes it easier to build and maintain notification servers. Third, more powerful information consumers can be built because of the additional data that can be added to an event in the form of meta-information without changing the event *per se*. This additional data can be used to enable the server to provide additional services as guaranteed delivery, security, event ordering, etc.

3.5 What interfaces are implemented by the notification servers, or changing from one notification server to another

3.5.1 Theoretical Considerations. As previously discussed, when a push architecture is used between the information consumers and the notification server, the consumers are

invoked or “called” by the notification server. Thus, the consumers must conform to a standard interface, which is usually described as a set of methods with a specific signature. For example, every consumer that uses Elvin needs to implement the method *public void notificationAction(Event e)*. Although the gauges do not need to provide an implementation of these methods, a typical approach is to add to this method the code to be performed when an event arrives (e.g., increment the events counter, check the timestamp, etc.).

3.5.2 Practical Considerations. Each notification server we studied defines a unique interface with a different set of methods. For example, the interface to CASSIUS provides two different methods: one that is called when *a set* of events is received and another that is called when *no* event is received. The interface is as follows, respectively:

```
public void notification
    (java.util.Vector NotificationList);
public void noNotification();
```

Elvin’s interface provides just one method to be called for each event received. If several events are received, this method will be called several times, once for each notification. The interface is as follows:

```
public void notificationAction(Notification event);
```

Finally, Siena’s interface defines two different methods: one for receiving *one* event and another for receiving *a set* of events, respectively:

```
public void notify(Notification e);
public void notify(Notification s [ ]);
```

Thus, when using each of the different notification servers, modifications in the gauges’ code were necessary.

3.5.3 Lessons. The obvious lesson is that information consumers need to be modified to adapt to different servers because each notification server requires the implementation of a different interface. Furthermore, when multiple notification servers are in use an N-N problem exists between information consumers (gauges) and servers (as well as between information sources and servers), i.e., each information consumer must implement all the different interfaces for each notification server. This increases the likelihood of errors and inconsistencies among the objects and types used.

4. RELATED WORK

Fuggetta and colleagues present a framework for comparing different notification servers [1]. This framework is a pragmatic specialization of previous work by Rosenblum and Wolf that was created to be a guide for practical comparison between their work and that of others [13]. Their classification framework comprises the following components: the event model, which focuses on

characteristics of events; the subscription approach used in the notification server; the observation and notification models (i.e., the mechanisms through which the events are observed and consumers are notified); and, finally, the architecture of the notification server. Although there is some overlap between this comparison framework and the issues in this present paper, we took a perspective more akin to an empirical evaluation, a case study, or even a usability evaluation. We sought to provide detailed information to designers and practitioners who would be new to these kinds of systems.

Also, Rodden and colleagues use event-status analysis as a framework for analyzing cooperative applications and notification servers that provide feedthrough [12]. Feedthrough is defined as “the ability of one person to see the effects of another’s action”; that is, it is awareness about the actions of other users. Their approach to classifying event notification servers extends a theory of human-computer interaction. Because it provides a radically different classification scheme and terms compared to what most system developers are accustomed to, it may have less impact on actual developers.

5. CONCLUSIONS

We have taken the concept of awareness and used it in the broadest sense and in the spirit we believe the early innovators of awareness in the computer-supported cooperative work (CSCW) community intended—namely, awareness involves information about the activity of software systems as well as information about the activity of human collaborators. In previous work, we developed an event notification server that was tailored to support awareness with great flexibility and detail. While CASSIUS worked well, it was not a server that was widely available. If others were to learn from the CASSIUS foundation, we would have to attempt to recreate some of its effects with more commonly available servers. As a first step, we focused on evaluating a specific scenario in three servers. The evaluation provided lessons about the difficulties in adapting different server models. These should provide data for designers of event notification servers as well as applications designers and programmers interested in this technology.

ACKNOWLEDGMENTS

This effort was sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. It was also partially funded by the National Science Foundation under grant numbers CCR-0205724 and 9624846. The U.S. government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, the Air

Force Laboratory, or the U.S. government. The first author was also supported in part by CAPES grant BEX 1312/99-5.

REFERENCES

- [1] Cugola, G., Di Nitto, E., and Fuggetta, A. The JEDI Event-based Infrastructure and Its Application to the Development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9), 827–850, Sept. 2001.
- [2] Carzaniga, A., Rosenblum, D., and Wolf, A. Design and Evaluation of a Wide-Area Notification Service. *ACM Trans. Computer Systems*, 19(3), 332-383, 2001.
- [3] Dourish, P., and Bly, S. Portholes: Supporting Awareness in a Distributed Work Group. *CHI'92 ACM*, pp. 541-547, 1992.
- [4] Dourish, P. and Bellotti, V. Awareness and Coordination in Shared Workspaces. *Proc. ACM Conf. Computer-Supported Cooperative Work CSCW'92*, New York: ACM, pp. 107-114, 1992.
- [5] Fitzpatrick, G., Mansfield, T., et al. Augmenting the Workaday World with Elvin. *Proceedings of 6th European Conference on Computer Supported Cooperative Work ECSCW'99*, 431-450. Kluwer, 1999.
- [6] Gruber, R. E., Krishnamurthy, B. and Panagos, E. The Architecture of the READY Event Notification Service, *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshops on Electronic Commerce and Web-based Applications and Middleware*, pp. 108-113, 1999.
- [7] Hilbert, D., and Redmiles, D. Extracting Usability Information from User Interface Events. *ACM Computing Surveys*, 32(4), December 2000.
- [8] Kantor, M. Creating an Infrastructure for Ubiquitous Awareness. Doctoral dissertation, University of California, Irvine. Information & Computer Science Technical Report UCI-ICS-01-16, 2001.
- [9] Kantor, M., and Redmiles, D. Creating an Infrastructure for Ubiquitous Awareness, Eighth IFIP TC 13 Conference on Human-Computer Interaction (INTERACT 2001 - Tokyo, Japan), pp. 431-438, July 2001.
- [10] Kantor, M., Zimmerman, B. and Redmiles, D. From Group Memory to Project Awareness Through Use of the Knowledge Depot. *California Software Symposium*, Irvine, California, pp. 19-30, 1997.
- [11] Lee, A., Girgensohn, A., and Schlueter, K. *NYNEX Portholes: Initial User Reactions and Redesign Implications*. *CHI'97*, pp. 385-394, 1997.
- [12] Ramduny, D., Dix, A., and Rodden, T. Exploring the Design Space for Notification Servers. *Proceedings of the ACM 1998 conference on Computer-Supported Cooperative Work*, pp. 227–235, Seattle, Washington, 1998.
- [13] Rosenblum, D. S., and Wolf, A. L. A Design Framework for Internet-Scale Event Observation and Notification. *Proceedings of the Sixth European Software Engineering Conf./ACM SIGSOFT Fifth Symposium on the Foundations of Software Engineering*, pp. 344-360, Sept. 1997.
- [14] Sutton, P., Arkins, R., and Segall, B. Supporting Disconnectedness – Transparent Information Delivery for Mobile and Invisible Computing. *Proceedings of IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01)*, 2001.