Lessons Learned Using Notification Servers To Support Application Awareness

Cleidson R. B. de Souza*, Santhoshi D.Basaveswara, David 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 begged the question: How essential was the CASSIUS server 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.

Keywords
Event notification servers, publish/subscribe, application awareness, empirical studies of software tools

1. INTRODUCTION
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 [15]. Roughly, awareness refers to the ability to have available, pertinent information about other activities, usually the activities of co-workers. The combination of the CASS and CASSIUS was designed to mediate between various information sources and various awareness tools. The focus of that research was to investigate the possibility of having both flexibility *and* detail in awareness tools and information sources. This tradeoff was termed the detail-variety tradeoff.

Ongoing work begged the question of how essential the CASSIUS server was to our overall goal of supporting awareness. The question is biased toward practicality, although it certainly provides research challenges. Specifically, CASSIUS was designed to work optimally for certain kinds of contexts and end-user scenarios; however, other event servers are already more commonly used and have such advantages as greater efficiency scalability, and commercial support. Wherein lie the differences, and what are the implications for designers of systems that incorporate awareness?

This present work examines these questions through an evaluation of event notification servers including CASSIUS and two others more readily available, Elvin [8] and Siena [3]. Elvin was chosen because of its maturity and strong user community. Siena is relatively new but is rapidly gaining popularity in the software engineering community because it emphasizes scalability in distributed environments. We chose a scenario in which we were previously using the CASSIUS server to provide a specific kind of awareness, and implemented this scenario again in Elvin and Siena.

Our evaluation is twofold: theoretical and practical. We discuss several issues that arose naturally as the research scenario was implemented and reimplemented. Common issues that we discuss include differences in using push versus pull architectures, the power of subscription services, implications of event and object registration, the role of meta-information for events and objects, and issues of low-level interfaces. The practical lessons presented in this papers can be used by users and designers of notification servers. Users will find them helpful when choosing an event notification server, while designers will find the lessons more interesting while making design decisions.

The paper is organized as follows. The next section defines the problem more carefully. The subsequent sections describe the scenario used to study the notification servers, some background information about notification servers, and the lessons learned from our experiments with the servers. The final sections describe related work, limitations, future work, and some conclusions of the study.

2. PROBLEM
The most general challenge of concern is whether awareness in the broadest sense can be supported with event notification servers. 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.

*Also at: Informatics Department, Federal University of Pará, Belém, PA, Brazil.

CASS and CASSIUS [15] were designed to investigate two specific problems: the detail-variety tradeoff and interchangeability. Some awareness tools provide a great deal of detail about a single kind of awareness information or source, whereas others provide information about a variety of sources but without a great deal of detail. Moreover, few awareness tools are set up to be interchangeable with one another. Notification servers can both provide a solution to the detail-variety tradeoff and enable greater interchangeability of awareness tools from the end users' perspective. However, to be most useful for coping with these two requirements, notification servers need to provide a number of specific services: (1) provide a mechanism to locate information sources affecting users' work; (2) isolate relevant subsets of information from sources; (3) monitor sources and components (subsets of sources) for changes; (4) allow subscriptions to summaries of changes to the source or subset; (5) send notifications of changes to a notification server and, if needed, directly to the workers via email; and (6) represent notifications (or summaries of sets of notifications) within 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 [5]. Many awareness tools, such as Portholes systems [4][17], 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. 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 [16], 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 our previously developed CASSIUS server was able to accomplish the integration of different kinds of awareness information and deliver it to a variety of awareness tools, it was not rigorously supported because it is a research tool. Hence, we undertook the evaluation of more commonly available event-notification servers for supported general awareness information.

We took as an evaluation point, a scenario we had developed for a Defense Advanced Research Projects Agency (DARPA) project. At this stage, the project was focused on the kind of application awareness outlined above. Specifically, visualization tools, called "gauges," should show the state of applications, including warnings of problematic states. Although gauges that reflect the activity of group awareness and even process information in project awareness will eventually be developed, only this application awareness was specifically evaluated so far. Also, although a similarity exists between the concept of application gauges and performance monitoring, performance monitoring tools are usually monolithic interfaces, and we wanted to have an interface that allowed for easy interchange of narrowing directed monitors. Moreover, we are interested in a software approach that integrates conceptually with other awareness tools or gauges.

3. SCENARIO

This section describes the scenario used to compare the different notification servers. The scenario comprises the application being monitored and the gauges developed to provide application awareness about the behavior of this application. The application we selected is based on a real system initially developed at Lockheed Martin and later modified at UCI by the group in software architecture research [7]. The lessons learned about the three notification servers described in this work are based on this system; that is, the lessons are based on our experience in changing the previous notification server used (CASSIUS) to the others (Elvin and Siena).

3.1 The Application

One of our main concerns is to provide awareness about the behavior of a software system and also the means to use this awareness information to support end users in their daily tasks. Certain types of software systems, such as complex applications or safety-critical systems, have a greater need to be monitored than others. The Airborne Warning and Control System (AWACS) [1] system is one such application. AWACS is a radar and computer system that provides a survivable airborne surveillance capability with command, control, and communication functions. It is deployed on a militarized Boeing 707-model aircraft and provides real-time information. AWACS is the world standard for airborne early warning systems. It supplies tactical and air defense forces with surveillance, and command and control communications. Its flexible, multimode radar, in a rotating radome mounted above the fuselage, allows AWACS to separate maritime and airborne targets from ground and sea clutter. It has a 360-degree view of an area, and at operating altitudes can detect, identify, and display targets more than 200 miles away. It is a large and complex system comprising more than 130 components, 200 connectors, and 30 subsystems. These components constantly exchange events and other information with each other. Additionally, the components might have complicated dependency relationships with each other. Such a scenario, due to the complexity of the software being monitored, makes it very difficult for the designers of the software to anticipate all possible problems that could occur. In this case, we argue that application awareness information might be of considerable importance to the developers and users of this system.

For obvious national security reasons, we could not have access to the actual AWACS system immediately. We used a sanitized AWACS simulator provided to UCI by Lockheed Martin. This simulator was rebuilt using an XML-based architectural description language and developed as a completely new application [7]. This architecture is very large, containing similar elements over and over with about 300 components distributed among 30 subsystems. On an average, 15 events are sent within the simulator per step. In order to collect and display the data being monitored by the various notification servers, we built a number of gauges. The gauges provide application awareness about the software system being monitored. They are described in greater detail in the following sections.

The primary motivation of our research was to explore the viability of providing application awareness to users and to examine how this awareness can be used to support end users in their daily tasks. Creating awareness tools and gauges that display information collected from the application being monitored in a user-friendly manner can provide application awareness. Another part of the problem was to make an accurate guess as to the type of information of interest to end users in our particular scenario.

In summary, our interest in application awareness involved awareness of the behavior of a software system and how this awareness could support end users in their daily tasks. It is envisioned that this kind of awareness information will be combined with other sources of information to keep end users informed. Moreover, the decision about which behavior is important would depend on the user based on the application being monitored and the environment in which the application is running, among other factors.

3.2 The Gauges

To provide application awareness, we developed a set of seven different awareness gauges that present different types of information about the behavior of the AWACS simulator. Also, to take a wider range of cases into account, we created generic gauges (e.g., gauges that indicate the degree of activity in the system) as well as specialized gauges (e.g., Architectural gauges) for the system, as described below:

We developed three types of Performance gauges, which reflect the degree of activity within the AWACS simulator. Certain patterns will reflect "normal" activity while others will reflect problematic states. 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™. All of these gauges present the same data, but each one is visually different from the others (see Figures 1a, 1b, and 1c).

The Clock gauge can be used to indicate the presence or absence of activity in the AWACS system (see Figure 1d). In this application, notifications occur only when the simulator advances a step. Thus, when the Clock gauge receives a notification from the CASSIUS server, it advances the visualization. If no notifications are received, the clock remains stationary.

The Signal gauge (Figure 1e) 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.

The Progress Bar gauge (Figure 1f) counts the number of occurrences of a specific event in the AWACS simulator.

The Architectural gauge is the most complex gauge. It 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. Obviously, the appearance of this gauge is similar to the interface of the simulator itself and could be substituted for it, but more commonly it abstracts a smaller number of key components. A screenshot of the architectural gauge is presented in Figure 2.

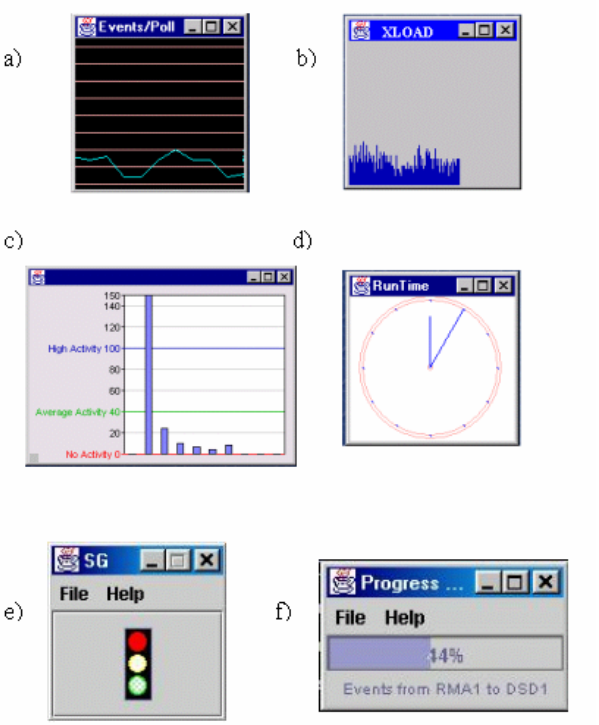

Figure 1:
(a) The Line Graph gauge traces activity over time, emphasizing times of heavy load.
(b) The Xload gauge also traces activity over time.
(c) The Bar Chart gauge displays a bar chart graph with information about the activity in the application being monitored.
(d) The Clock gauge shows how long 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.

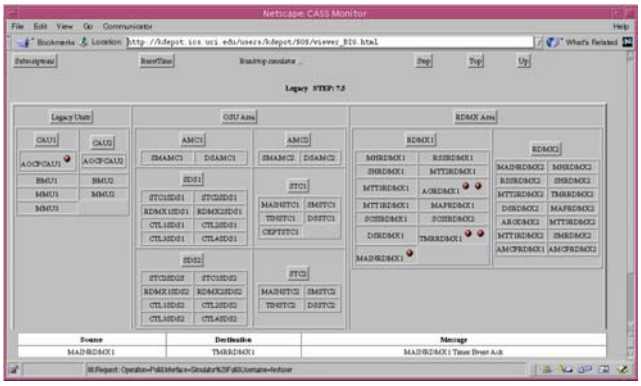

Figure 2: The Architectural gauge abstracts aspects of the software architecture of the AWACS simulator.

It is important to note that we could not interview the users of the AWACS system. Therefore, the gauges are based on suppositions and assumptions made by observing similar tools and by drawing from previous experiences and research. Thus, for example, the Xload gauge is similar to the X-Windows gauge used to measure the amount of processor activity taking place on a UNIX system.

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.

4. BACKGROUND

There is a growing trend of creating distributed software applications that run over the Internet. Because the Internet is not reliable with respect to network congestion, security, and so forth, these applications can be thought of as working as loosely coupled autonomous objects. That is, instead of using direct communication, these applications use an event-notification or publish-subscribe design style—in other words, messages are not sent from one application directly to another, but to an event dispatcher (also called a notification server), which, in turn, distributes them to the applications that expressed interest in 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 categories of events. Events are sent by the information sources to the notification server, which ensures the delivery of these events to all interested consumers. In general, in an event-based system, component interactions are modeled as events, which are transmitted in the form of notifications [21]. 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) connectors as well as pattern matching, sequence matching, and so on. Figure 3 describes the logical architecture of a generic notification server with the information sources and consumers, as well as the communication paths between them.

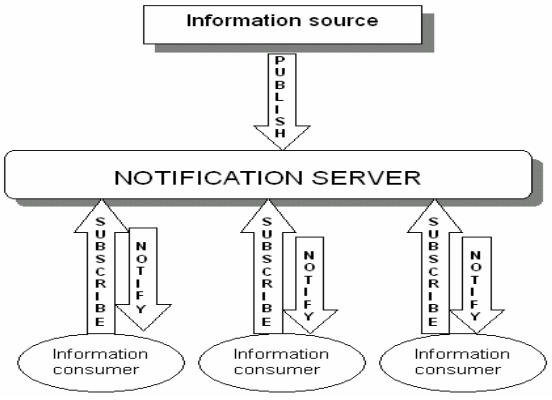

Figure 3: A generic event notification server.

An important advantage of using an event-notification style is the decoupling between the sources and consumers of the events—that is, there is no direct communication between them. This decrease in coupling results in improved flexibility in the entire system. For example, suppose an information source publishes its events without knowing which components are interested in them. Meanwhile, the information consumers subscribe to them in order to be notified, even though they do not know which component is the source of the event. This way, if a new information consumer wants to be notified about an event, it just needs to subscribe to it. The information source does not need to be modified, and neither do the other components of the system. Furthermore, if a new information producer is added to the system, no modification is necessary. In short, the (event) technology enables a "plug and play" approach to support dynamic reconfiguration and introduction of new service components [2].

This example of how a notification server works is intentionally general to allow us to raise different questions about the details of each notification server. For example: Which information sources can send events to the notification server? How powerful is the subscription language for each notification server? Does it allow abstractly or concretely defined sequences of events? How are the consumers that subscribed to particular events notified? Which meta-information (e.g., location of the information source, time stamp when the event was sent, etc.) is added to the events sent to the notification server? These questions will be answered in the next section, which describes our experiences in using CASSIUS, Elvin, and Siena.

5. 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 reimplementations of the scenario. Third, lessons learned entail the implications or benefits, if any, of one approach over another.

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

5.1.1 Theoretical Considerations

In general, as shown in Figure 4, 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 other component to check for new events. Usually, it is necessary to specify how often one component wants to pull the other. For example, since CASSIUS implements a pull architecture when communicating with the gauges, the gauges poll the notification server at specific intervals to check for new events.

Conversely, in a push architecture components send events or "push" them onto other components. Thus, a receiving component

is invoked or "called" by the one that is sending events. In this case, these components need to implement some kind of a standard interface to be "called" by the others. For example, since Elvin uses push technology between the notification server and the consumers (gauges), the gauges need to implement the method *notificationAction(Event e)* to be called by the Elvin server. The code to be executed when an event arrives is also included in this method. Therefore, as soon as an event arrives, the corresponding action is executed. Each notification server used in this study defines a different interface with a different set of methods.

5.1.2 Practical Considerations

Elvin, Siena, and CASSIUS all implement a push architecture with respect to the information sources. That is, the information sources are responsible for connecting to the notification server and sending events to it. However, for communication with the gauges, Elvin and Siena implement a push architecture, whereas CASSIUS uses a combination of the pull and push architectures. Specifically, if the communication between the gauges and CASSIUS uses the *http* protocol to establish a connection, then this communication is performed using a pull architecture. However, if the gauges use the CASSIUS application program interface (API), a combination of push and pull is employed. In this case, pull is used between the notification server and the API, whereas push is used between the API and the gauges.

Although the CASSIUS API implements a push architecture to contact the gauges, it is different from that of Elvin and Siena, which send the received events to the gauges as soon as they receive them. In contrast, the API sends the events to the gauges after pulling the CASSIUS server and receiving events. This means that, in the worst case scenario, a gauge using the CASSIUS API will wait the entire interval between two pulls before receiving some event. This difference is crucial for our scenario, in which the user wants to be immediately aware of modifications in the state of the AWACS simulator. In a real-time scenario such as ours, some gauges might have to receive a notification as soon as it occurs.

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. So, 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, such as the number of events that arrived per five seconds, in a graphical form,. 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). Please note that although this was not a substantial change, it shows that the interchangeability among notification servers can be problematic when different architectures are being used.

5.1.3 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

Thus, instead of thinking of the two as completely different architectures, we rather think of push and pull architectures as extremes of a spectrum of communication models between components such as information sources and notification servers or notification servers and gauges. For example, as implemented in the CASSIUS server, an API could poll the server for new events and then push them to the gauges. In this case, a combination of pull and push is used. Alternatively, a gauge could specify a frequency for receiving events from the notification server as being different from the frequency of polling in order to conserve bandwidth usage.

Furthermore, we envision future notification servers implementing this entire spectrum, i.e., combining both architectures to simplify the construction of more powerful information consumers. According to the constraints and requirements of the consumer, models could be customized. Moreover, the gauge could be "intelligent" enough to make "on-the-fly" changes to the architecture used according to environmental factors such as network bandwidth, location, previous events received, etc. Hopefully, an integrated model would avoid the same problems of a ad hoc, mixed model.

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

5.2.1 Theoretical Considerations

An interesting feature that can be used to compare notification servers involves the types of event matching they support. 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 [13]. 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 [13].

Finally, it is important to mention that 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. This subscription involves substring matching, which is a type of regular expression.

[”ComponentFrom==RDMA?”]

The location at which event matching is performed is another important factor to be considered. According to Gruber, Krishnamurthy, and Panagos [10], 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. For example, 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 [10]. 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.

5.2.2 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 all the types of sequence detection mechanisms described above[1].

The location at which event matching is performed is also important. The best location depends on the environment in which the notification server is being used. For example, adding event matching to the client is a ***reasonable*** design decision in a situation in which the network speed connections are reliable and fast [10]. Another approach is to distribute the matching among the components according to the workload. For example, the algorithms could be executed at the server when there are only a few events. Later, when the number of events increases, the matching could be performed at the information consumers. This technique is used in the notification server READY [10]. This approach is not used, however, in any of the notification servers that were evaluated in this study.

Siena and Elvin perform event matching at the notification server, whereas CASSIUS applies the algorithms at the server as well as at the gauges. In CASSIUS, the logical and regular expressions are executed at the server, and the sequence matching is carried out in the API, at the client side, and then the matched events are pushed to the gauges. Thus, when a gauge subscribes to a sequence of events such as [file=A]; [file=B]; [file=C], events that match the three conditions will be sent from the server to the client. However, the API hides those events from the gauge, calling it only when the sequence is complete, and providing the entire set of events that match that sequence as a *java.util.Vector* object. In this case, the gauge can use the meta-information provided with each event of the sequence to make calculations such as “How long did it take from the first event to the last event in the sequence?" and so on. Although we did not perform formal tests, we did not detect performance problems with the sequence detection algorithms being performed at the API.

5.2.3 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:

- It decreases the flexibility of the application; and
- 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).

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

5.3.1 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

[1] In our original 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.

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 [10]. 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 [10]. However, to the best of our knowledge, no notification server uses this approach.

5.3.2 Practical Considerations

CASSIUS adopts a strategy in which the information sources need to register with the notification server before sending events [15]. 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 [14]. 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.

In our scenario, because it is not mandatory to send this additional information when using Elvin and Siena, an extra effort had to be made to add it to the event. In a scenario such as ours, the amount of additional information required was small. Hence, the effort to add it to an event was negligible, but in more complex scenarios, the effort required for doing so could be complicated and expensive. However, this additional information about event types can significantly improve the performance of event matching algorithms (as discussed next).

5.3.3 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.

Furthermore, the biggest problem encountered when a notification server does not provide event registration is that information consumers must be aware of the naming conventions used by information sources when they are publishing events. Otherwise, events may be missed due to name mismatches. For example, different information sources may simply use different names for what is actually the same event, or different names in the event attributes may have reflected event sub-typing. However, having to know these naming conventions means a tight coupling between the information sources and consumers, because if the information sources changes its naming convention the information consumers must be notified about it, otherwise they will not function properly anymore.

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

5.4.1 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. Therefore, this meta-information, in addition to the other information, is available to the consumers. Examples of meta-information that can be provided by a notification server are, among others, 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.

It is important to note that meta-information is extremely important to our scenario because the AWACS simulator comprises several hundreds of components, and information about which component sent an event to which other component might be crucial to truly expose the behavior of the system.

5.4.2 Practical Considerations

An information source in Elvin and Siena that wants to provide meta-information while sending events has to explicitly add this information along with the event. In fact, the same methods used to send the event are used to send the meta-information. 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.

Subscriptions in Elvin are based on the content of the events; that is, one could subscribe to [“ComponentFrom”==”RDMA1”], which subscribes to all events sent by the component RDMA1. However, as described earlier in our scenario, we are interested in all the events from the AWACS simulator because we want to be aware of the complete behavior of the system. To do so using Siena and Elvin, we could just use a subscription such as [“ComponentFrom”==”*”], meaning that we are interested in all events being sent by any component in the system. However, the problem with this approach is that other events that are not initiated in a component will not be pushed to the gauges. To avoid this problem, we decided to add to each event meta-information that describes the type of the event being sent. This can be achieved by using the subscription ["EventType"=="AwacsSimulatorEvent"]. As in the previous subscription, this one matches all events being sent by any component within the AWACS simulator; along with that, it also matches events being sent by other objects[2].

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.

5.4.3 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.

5.5 What interfaces are implemented by the notification servers, or Changing from one notification server to another

5.5.1 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.).

5.5.2 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' codes were necessary. For example, the Clock gauge in Siena (see Figure 2d) uses the same implementation for both methods; the same actions are required whether one or a set of events arrives. In this case, it might be a better approach to create a third method that contains the common code between the methods, thus avoiding redundancy of code and reducing maintenance problems.

Another example involves the gauges that reflect the degree of activity in the simulator, by creating a graphical display for this (see Figures 2a-c). When these gauges are implemented using Siena as their notification server, they have very similar implementations for both methods of Siena’s interface. In the first method, the amount of activity detected can be described by one event, whereas in the second, the activity is represented by a set of events. Again, a good implementation could use the amount of activity in the simulator (one or the number of the set of events, depending on the method called) as a parameter to a common method that graphically displays this information. But, as described before, the programmer is responsible for taking care of that. It is an issue that is potentially problematic.

5.4.3 Lessons

The first and most 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

[2] Note that in the specific example presented here, it is possible to construct another subscription (using a wildcard expression) that is equivalent to the one created using event types. By equivalent, we mean that two different gauges, each one using one of these subscriptions, would receive the same set of events. However, this is not necessarily true for all cases. One can also argue that even though adding meta-information is not a difficult task, a certain amount of additional effort might be required from the programmer's side. Event types are also important to avoid performance problems in event matching algorithms because those algorithms can be costly [GKP99].

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. On the other hand, since all the notification servers require different interfaces to be implemented, in *theory* it would be possible to easily create an information gauge that receives notifications from all of them, assuming that there are no inconsistencies between the object types required for each API. In case of inconsistency problems, one solution would be design systems with consumers linked always to the same server and connectivity achieved through server interconnections. Another solution would be for designers of gauges to apply additional effort to design for modification and heterogeneity (in lieu of a standard).

6. RELATED WORK

Fuggetta and colleagues present a framework for comparing different notification servers [2]. 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 [20]. 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. Fuggetta and colleagues do not compare Siena or CASSIUS, but they do discuss several issues presented in the current paper. For example, they refer to the issue of push versus pull architectures as the observation and notification models between the server and the consumers.

Although we agree that a formal classification framework is important, it is not our purpose in this paper to propose a formal classification. Instead, our present concern is to describe the practical lessons we learned while using different servers. We believe that other developers will find these practical considerations helpful in choosing an appropriate notification server for their needs.

Fuggetta and colleagues [2] go into a great deal of detail about subscription approaches used in different notification servers, an issue of great pragmatic importance. They describe the following approaches:

- Content-free, in which the subscription is accomplished by specifying a *channel*—each notification sent to this channel will be delivered to the consumer;
- Subject-based, in which the subscription is defined based on the *subject* of interest—each event is labeled with a subject;
- Content-based, in which the subscriptions are specified as expressions evaluated over the event content—logical operators and other constructors are used; and
- Event combination, in which a sequence of events could be defined—the subscription can be created using combinations of different events instead of combinations of fields of the same event, as in the previous cases.

According to this classification, Siena and CASSIUS implement event combination, whereas Elvin implements content-based subscriptions. Therefore, in the area of subscription languages, their classification augments our work by describing several other notification servers.

Finally, Rodden and colleagues use event-status analysis as a framework for analyzing cooperative applications and notification servers that provide feedthrough [18]. 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, especially on actual developers. Moreover, the scheme presented by Rodden and colleagues creates a space in which more possibilities might be explored, but many of them have not yet come into practice.

7. LIMITATIONS

We recognize some limitations in our experiences as reported. First, for the purpose of this paper, we used only three of the several existing notification servers. However, we argue that these systems were enough to provide an overview of the different issues that system designers will face when working with notification servers. Second, our scenario was very specialized. Third, this scenario was not optimal for any of the notification servers used; that is, it was not similar to the scenarios idealized by the designers of Elvin, Siena, or CASSIUS. For example, CASSIUS is best suited to provide awareness information in a mobile environment, in which the information consumers can disconnect [15]. However, from one perspective, this specialized scenario provided a better evaluation of the notification servers because they are used to implement a real scenario, different from their original context. After reading the lessons we learned, one might argue that the problems we encountered were due to a bad choice of notification server; that is, different notification servers could have avoided the problems that we had. Nonetheless, Elvin and Siena are being promoted by their developers and sponsors for many different applications, so many developers will be faced with less than an ideal context for applying these servers.

8. FUTURE WORK

An interesting point to be further considered is the creation of a common specification format to define event types. The software architecture community adopted a similar approach by defining xArch, as an "XML-based representation for building ADLs [architecture description languages]" [6]. xArch inherits the extensible markup language's (XML's) extensibility mechanism based on schemas. Therefore, users can independently extend xArch with features that support their particular needs. In fact, xArch has been extended to specific domains such as configuration management, architectural diffing, Java implementations, and so on [6]. We are evaluating the possibility of using a similar approach to define events. A generic event would be defined in XML, and specific event subtypes could be created using XML's schemas. For example, by using this approach, a specific event could be created to integrate different software engineering tools, thus allowing them to communicate and cooperate [19]. In addition to extensibility, another advantage lies in the tool support that would be provided to event authors because a growing number of XML tools are becoming available to assist schema designers. Finally, as discussed previously, event typing is important to avoid performance problems in event matching algorithms [10]. Once the events are defined in a common language, the notification server could publish them,

thus allowing information consumers to browse and choose the events in which they are interested (this is somewhat similar to CASSIUS and to the concept of *advertise* in Siena [3]). Very preliminary work has sketched out one approach to creating XML events [9]. That work associates structural and semantic information with the event data in order to assist the interpretation of the events by gauges.

Another issue discussed before involves event-matching algorithms, which are a powerful technique that can be used to detect inconsistent behavior, dangerous situations, and so on. However, to provide such a service, the notification service must provide some kind of event caching. We are studying the possibility of altering Elvin or Siena to provide such a service.

9. 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 a particular kind of awareness, which we termed project awareness. Although the CASSIUS server had advantages for some kinds of awareness situations, it had limitations of speed and robustness when compared to more commonly available servers. Hence, we were motivated to experiment with these other servers to learn about their advantages and disadvantages by using them in a few very real scenarios. Thus, the lessons presented here speak to the community of designers and developers who work with collaborative applications and who seek to support multiple uses of awareness in their applications. In the lessons we presented, we sought to maintain a balance between the theoretical and the practical and to provide a data point for future designers and practitioners.

10. 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.

11. REFERENCES

[1] United States Air Force, AWACS E-3 Sentry Fact Sheet. URL: http://www.af.mil/photos/Mar1999/981211awacs.html

[2] 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.

[3] 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.

[4] Dourish, P., and Bly, S. Portholes: Supporting Awareness in a Distributed Work Group. CHI'92 ACM, pp. 541-547, 1992.

[5] 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.

[6] Dashofy, E. M., van der Hoek, A., and Taylor, R. N., A Highly-Extensible, XML-Based Architecture Description Language. Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001), Amsterdam, Netherlands, August 2001.

[7] Dashofy, E. M., van der Hoek, A., and Taylor, R. N., An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. To appear on the Proceedings of the International Conference on Software Engineering, Orlando, Florida, 2002.

[8] 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.

[9] Gross, P. N., Gupta, S., Kaiser, G. E., Gaurav, S. Kc., and Parekh, J. J. An Active Events Model for Systems Monitoring. Proceedings of the Working Conference on Complex and Dynamic Systems Architecture, December, 2001.

[10] 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.

[11] Gutwin, C., Roseman, M., and Greenberg, S. A Usability Study of Awareness Widgets in a Shared Workspace Groupware System. Proceedings of the ACM Conference on Computer Supported Cooperative Work CSCW'96, pp. 258-267. New York: ACM, 1996.

[12] Hilbert, D., and Redmiles, D. An Approach to Large-Scale Collection of Application Usage Data over the Internet. Proceedings of the Twentieth International Conference on Software Engineering (ICSE '98, Kyoto, Japan), pp. 136-145. IEEE Computer Society Press, April 19-25, 1998.

[13] Hilbert, D., and Redmiles, D. Extracting Usability Information from User Interface Events. ACM Computing Surveys, 32(4), December 2000.

[14] Kantor, M. Creating an Infrastructure for Ubiquitous Awareness. Doctoral dissertation, University of California, Irvine. Information & Computer Science Technical Report UCI-ICS-01-16, 2001.

[15] 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.

[16] 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.

[17] Lee, A., Girgensohn, A., and Schlueter, K. *NYNEX* Portholes: Initial User Reactions and Redesign Implications. CHI'97, pp. 385-394, 1997.

[18] 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.

[19] Reiss, S.P. Connecting Tools Using Message Passing in the Field Environment, IEEE Software, 7(4), 57-66, July 1990.

[20] 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.

[21] 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.