Exploring the design space for notification servers

Devina Ramduny, Alan Dix
School of Computing
Staffordshire University
Stafford ST18 0DG UK
+44 1785 353255
{D.Ramduny, A.J.Dix}@soc.staffs.ac.uk

Tom Rodden
Computing Department SECaMS
Lancaster University
Lancaster LA1 4YR UK
+44 1524 593823
tom@comp.lancs.ac.uk

http://www.soc.staffs.ac.uk/~cmtajd/topics/webarch

ABSTRACT
Issues of notification and awareness have become increasingly important in CSCW. Notification servers provide a notable mechanism to maintain shared state information of any synchronous or asynchronous groupware system. A taxonomy of the design space for notification servers is presented, based on theoretical results from status–event analysis. This generates a framework and vocabulary to compare and discuss different notification mechanisms to improve design. The paper shows that notification servers are often ideally placed to support impedance matching to give an appropriate pace of feedthrough to the user by allowing them to see changes to shared objects in a timely manner.

Keywords
Collaborative applications, feedthrough, awareness, status-event analysis, initiative, mediation, notification server, protocol, impedance matching

INTRODUCTION

An essential feature of collaborative interfaces is *feedthrough* – the ability of one person to see the effects of another's actions. This is important for 'functional' reasons as it allows each person to see an up-to-date version of the work; for 'coordination' by preventing inconsistent updates and finally for supporting general 'awareness' of other people at work. The first two have given rise to considerable work on algorithms for synchronous editing and for merging versions of asynchronously edited material. The last factor has instead always been an informal concern in CSCW, it has however recently been augmented by formal analysis of 'awareness' models [2,19]. This modelling approach has arisen largely because of work on virtual collaborative environments. Whereas the main objects of interest used to be documents or shared drawings, now they are the virtual locations and actions of the participants themselves. Both effective feedthrough of updates to shared data and up-to-date views of other participants require underlying computational mechanisms to distribute and inform about these updates.

There are therefore two main requirements, firstly, to access and update shared data and secondly, to know when that data has been updated. The former lies behind the design of shared data repositories, either bespoke systems designed for CSCW [4,13] or off-the-shelf databases and shared object stores. The latter requires notification mechanisms. Even if data is stored and accessed rapidly from a central location, it is of no use unless client programs know that it has changed and users' screens are updated accordingly. Notification mechanisms fulfil precisely this role – telling programs and people not about what has happened, but that it has happened. Without this, users may eventually see the changes that have occurred but at a timescale and pace that is not acceptable for the task at hand.

Each application that updates shared data can be responsible for notification, and consequently broadcast to all interested parties that the change has happened. However, as with peer-peer methods for data replication, this has a high overhead in both algorithm complexity and network load. For instance, every participating client program should know about all others in order to broadcast change information to them. Furthermore, the changes must be kept up-to-date as users join and leave the system. So, for just the same reasons that data stores are often centralised, there is a need for notification servers to keep track of interested parties and take over the task of propagating change information. Such notification servers may either be coupled closely with the data store, as is the case with some databases supporting triggered actions, or they may be entirely separate, knowing about the data but being decoupled from it.

Clearly, there are numerous ways in which notification services can be managed in a collaborative system and the central aim of this paper is to explore and clarify the design space for notification servers. We believe this is a timely point as the last few years have seen the beginnings of a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CSCW 98 Seattle Washington USA
Copyright ACM 1998 1-58113-009-0/98/11...$5.00

literature on notification servers as entities in their own right. It is also clear that the development of the web has forced the issue in showing that data storage (in the form of web pages) may be separate from control issues (such as indexing). In our previous work, we have looked in a similar fashion at web-based CSCW architectures and at awareness models [18,19]. A structured analysis of design options is important in clarifying the similarities and differences between different example systems and in identifying new directions. It is also essential for the academic credibility of the discipline.

We begin this paper with a brief description of status-event analysis, an analytic framework developed to tackle various user interface issues, which we use as the basis for our subsequent analysis of notification servers. This foundation is essential as it enables us to ascertain that the options we consider do cover the design space. We use Status-Event analysis to examine the ways in which an agent in a system can become aware of a status change. This is then employed to look at the ways in which a notification server becomes aware of changes in the shared data and how it in turn makes this available to client applications. The ultimate purpose of a notification server is to provide effective user-level behaviour. We will finally look at how a notification server can provide impedance matching in order to achieve an appropriate pace of feedthrough for the users.

STATUS-EVENT ANALYSIS

Analytic techniques in Computer Science tend to focus on events as the locus of activity and control. This is natural given the discrete nature of computer systems. Also for user interfaces and collaborative systems it is a good way of describing input such as keystrokes, mouse clicks and network messages between remote applications. However, if we consider shared data in collaborative systems, the event-based models fit less well. The nature of shared data is that it persists; it doesn't just happen at a particular moment, it is always there. This is not the only phenomenon of its kind in user-interfaces; the position of a mouse and the contents of a screen are similar.

In fact, status-event analysis was developed some years ago to deal with such phenomena. It is a collection of semi-formal and formal techniques with a shared conceptual framework that includes aspects of both events and status. Status is used to describe all those occurrences which, like shared data, have a persistent value through time – events happen, status are.

Applications Of Status–Event Analysis

Status–event analysis has been applied in several contexts – from the analysis of issues in fine grained interaction [12] and auditory interfaces [5,6,9] to the specification of the complex behaviour of shared scrollbars in collaborative applications [1,10]. Status–event analysis is also the theoretical foundation of the Cameo [23] architecture which is used in the construction of CyberDesk [22], a Java based framework for dynamic integration of desktop applications. An important use of status–event analysis has been in the understanding of delays in user interfaces and collaborative systems [11,12]. The study of delay has centred around the idea of mediation and this will also be the key to an architectural understanding of notification mechanisms.

Key Concepts

The two central concepts in status–event analysis are obviously events (such as mouse click, beep, 6 o'clock) which occur at particular moments and status (shown by mouse position, screen, position of hands on the clock) which always have a value. In addition, we have to consider agents which respond to events and thereby modify the status.

A key feature of status–event analysis is the difference between the *actual event* and the *perceived event* for a particular agent. For example, the time may be six o'clock, but one may not notice it until a few minutes later when one looks at the clock. Similarly with notification servers, although events occur at certain places there may be a substantial delay before those changes are perceived.

Mediation

The most important aspect of status–event analysis for this paper is that of mediation. This is when some desired behaviour is achieved by interposing some additional agent or status entity. Consider the example of an electronic mail system – when a mail has arrived (an event) this is communicated to the user by a change of the screen icon (a status) [12]. The use of a status to mediate communication between agents is very common and in particular, it is the essence of shared data.

A second form of mediation occurs when a status–status relationship needs to be maintained [11]. For instance, when dragging a window across the screen with a mouse, the window must keep track of the mouse position. These relationships are typically maintained by a mediating agent which monitors the first status and then alters the second accordingly.

In both the case of agents communicating via status or an agent mediating a status–status relationship we face the same question: how does an agent become aware of a status-change?

Status Change Discovery

Let us reformulate the previous question. We have a status S and some agent A. An actual event occurs and the status S changes its value. What process occurs so that this can become a perceived event for the agent A?

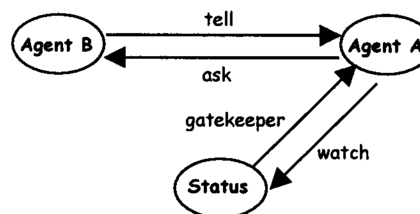

Figure 1 status-agent interaction

Figure 1 shows the alternative interactions by which agent A becomes aware of the status change. The critical point is that we are not interested in the flow of data that informs agent A of the new value of the status; instead it's simply the perceived event for agent A that a change has occurred.

Case 1 *watch*
Agent A can watch the status S. In a discrete system 'watching' means periodically *polling* the status.

A second question now arises: what event prompts the polling? This leads to 3 subcases:

- time driven – polling at fixed intervals
- demand driven – checking the status when it is needed
- spontaneous – caused by some unrelated event

The polling model of update is common in many client-server based applications where a local client needs to access some remote repository to refresh the local client's states. For example, in the case of groupware systems such as Lotus Notes[1] the system periodically accesses a remote server and updates the local client. At the moment of refresh, the user is informed of updates to the remote server.

Case 2 *told*
Agent A may be told by a second party agent B. This occurs in certain arrangements used in collaborative filtering where a user registers an interest in changes of a particular form. When the system updates a central repository the registered clients are told of changes that effect them.

Obviously we may then ask how does B know about the status change, again leading to two subcases:

- originator – B is the agent which caused S to change (B is packaged with S)
- mediator – B needs to find out itself, by one of the methods 1 – 4

Case 3 *ask*
Agent A asks the second party agent B. In this case we need to both ask what event prompts A to ask – leading to subcases as in case 1 and how B knows – leading to subcases as in case 2.

Perhaps the most notable example of this category is the logon process for computer conferencing system where an agent managing a centralised repository is asked to inform a new user of any alterations to the system. This is often presented as the number of unread messages.

Case 4 *gatekeeper*
The status is in some way active or is closely bound to an agent which 'knows' instantly when the status is changed. Such an agent can then tell A that S has changed.

This arrangement is usually employed in active databases in order to propagate the effects of changes. A similar technique is adopted in terms of the use of adaptors to underlying objects [21]. This paradigm is also applied in Suite [7] and other constraint based toolkits.

Source v/s Initiative

Figure 2 shows a source versus initiative matrix. Although the actual information resides in the status, we are particularly interested in the source as it holds the knowledge of any changes to the data. Initiative plays a key role in determining how changes of status are discovered.

Source \ Initiative	observer (agent A)	other (status/ 2nd party agent B)
status	1 watch	4 gatekeeper
2nd party agent (agent B)	3 ask	2 tell

Figure 2 Source v/s Initiative

Both cases 2 and 3 involve a second party agent, but the difference between them is one of initiative. In case 2, it is the second party agent B which takes the initiative to find out about the status change while in case 3, the responsibility lies with the agent A itself. Therefore the difference between 'asking' and 'telling' is one of initiative. Similarly, in case 1 the initiative originates from agent A as it polls or watches the status but in case 4, the initiative comes from the status itself.

Given this arrangement we may then ask ourselves what prompts one to take the initiative. For example, in case 1 this leads to subcases such as it may be internal (most likely time driven), or it may be due to a third party agent (demand driven) or it may be spontaneous (either be time driven or demand driven).

This section has examined the status event arrangement that exists between an agent altering some status value and an observing agent interested in changes to this status value. While it is possible to construct systems that manage status updates by propagating all alterations at a per update level to others, it does however require a considerable overhead at the application level. The application is in fact responsible for managing all the updates as they occur. The overhead involved in this management is one of the core motivations for the research into notification (or awareness) services that provide a set of standard techniques for managing status changes. The following section will consider the status event issues involved in the development of these notification (or awareness) servers.

NOTIFICATION SERVERS AS MEDIATORS

A similar analysis for the cases of status change discovery will now be applied to notification servers in collaborative applications. For the sake of the following discussion, it is assumed that there is a centralised architecture with some

[1] Lotus Notes is a registered trademark of Lotus Development Corporation

sort of central database or information server and client applications on each user's workstation.

Consider the following scenario. A user updates some shared data and the changes are sent by the user's client application to the central server. How does another user's client become aware of the change in order to update the screen?

Although the client applications are likely to be identical on both workstations, they take different rôles in this scenario:

Active Client (AC) – on the workstation of the user who performs the change

Passive Client (PC) – on the workstation of the user who is observing

Since the client applications perform the updates and display the resulting changes, we will concentrate on the clients themselves rather than the users who will ultimately interact through them. Thus we are concerned with how the events about the changes to the information are propagated rather than how the information is displayed.

To structure our analysis, we will first examine the scenario when there is no notification server and look at the options which enable the passive client to discover the changes. Then we will look at any additional options required when the notification server is added.

Without a Notification Server

The agents of interest are the active client and passive client and the status is the shared data (figure 3).

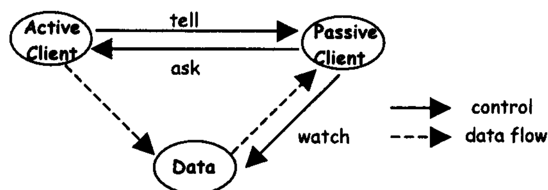

Figure 3 client-data interaction without notification server

In the previous section, we identified four cases of discovering status change. However, case 4 corresponds to the status having some closely allied gatekeeper agent and this acts as a type of notification server. Also, in cases 2 and 3 there is the possibility of a mediating agent; again this is the role of a notification server. Therefore we only need to consider the cases when the agent is the originator of the status change, namely the active client. Thus by direct application of the options from the status–event analysis we have the following options:

• passive clients polls the data (case 1)

This is the classic email arrangement where the responsibility lies with the client to interrogate the data and find out when changes have taken place.

• active client tells the passive client (case 2)

This arrangement is used in some forms of shared screen systems where screen updates are broadcast to all other clients. It is also adopted in multicast applications such as those used for virtual worlds[3].

• passive client asks the active client (case 3)

This situation is less common but normally occurs in shared screen systems where there is a designated master version of an application that is responsible for managing distribution of updates. Systems of this form include early versions of shared screen systems.

With a Notification Server

When a notification server (NS) is introduced, it acts as an intermediary between the active and the passive clients. This offers the benefits of managing the process and allowing support for more scaleable arrangements. The notification server facilitates distributed architectures including hybrid arrangements where the advantages of a replicated architecture in terms of local responses are combined with the propagation offered from a central awareness service.

In most cases both clients communicate with the notification server in various ways and we will consider each of them shortly. Essentially the active client informs the notification server of the update while the passive client seeks to be informed of updates. The notification server therefore removes the need for direct communication between the active client and the passive client. Also the notification server does not pass on the data to the clients. Instead the data repository fulfils this role. The notification server only mediates the control between the clients and the data. Events notifying changes to the underlying state information are basically sent between the clients and the notification server.

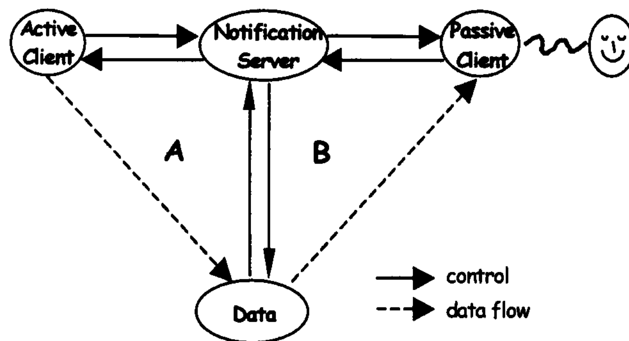

Figure 4 client-data interaction with notification server

Figure 4 shows the potential control flows between the clients and the notification server thus allowing the notification events to propagate through the system. It should be noted that not all of these control flows would be active in a particular system. By examining the combinations that may occur we will reach our taxonomy of the design space for notification servers.

Let us first consider the interaction between the notification server and the active client (figure 5). By applying the options discussed in the status-event analysis, the notification server is able to discover the stages of any change in the following ways:

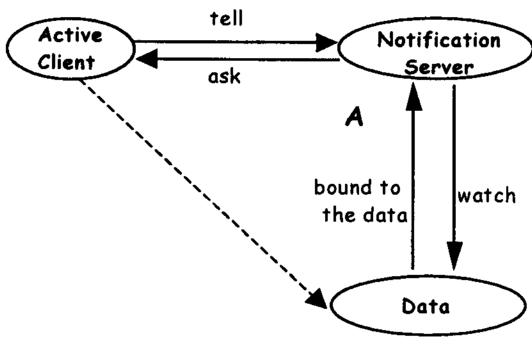

Figure 5 notification server communicating with active client and data store

• notification server polls the data (A1)

The responsibility lies with the notification server to monitor the data and detect any changes to the underlying data. This is often used in computer conferencing systems to provide some active propagation.

• active client tells the notification server (A2)

In this case the notification server is placed between the client and the data repository. This is similar to the technique used to develop shared X systems where a splitter was placed between the display and the underlying application [14].

• notification server asks the active client (A3)

This option is seldom used as it requires the notification server to ask the client if it seeks to make changes. However, with the development of mobile systems this arrangement is more likely to become significant as cellular architectures become more widely exploited.

• notification server is bound to the data (A4)

The Rendezvous [13] system adopts such an arrangement as it separates the abstract view from the data and uses some coupled mechanism to manage updates based on an encoding of constraints.

Once the change in the data has become a perceived event for the notification server, the latter must then relay that event to the passive client. The options are shown below.

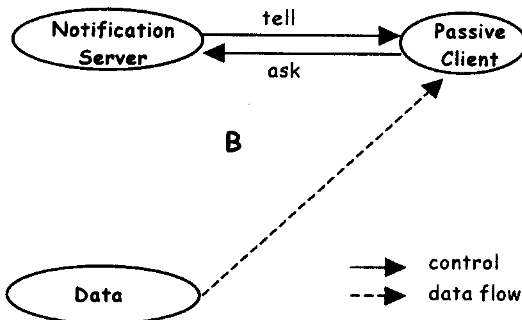

Figure 6 notification server relaying change passive client

Note that the figure does not show any explicit connection between the notification server and the data repository. Depending on the cases considered for A above, the link between the notification server and the underlying data may either be of a direct or an indirect nature.

• notification server tells the passive client (B2)

This is used by notification servers that are linked to window system where events can be sent directly to the client. The development of the push technology on the web also allows this to take place.

• passive client asks the notification server (B3)

This is the classic arrangement used in web based awareness mechanisms such as WAP [15].

It should be noted that since the notification server is acting as the intermediary, the options of having the passive client interacting with the data repository directly for changes (B1, B4) are ruled out.

TAXONOMY OF NOTIFICATION SERVERS

We saw that the notification server can find out about changes from the active client and the data store in 4 different ways (A1–A4). Furthermore, the notification server can communicate the updates to the passive client in 2 ways (B1, B2). We can represent these possibilities in a 4x2 matrix as shown in figure 7.

	B2 (NS ➡tells PC)	B3 (PC ➡asks NS)
A1 (AC ➡polls NS)	....	....
A2 (AC ➡tells NS)	....	....
A3 (NS ➡asks AC)	....	....
A4 (NS ≡ Data)	....	....

Figure 7 (4x2) matrix

The above matrix is now populated by some example systems built using the respective protocol to produce the taxonomy of notification servers (figure 8).

(AC-NS) \ (NS-PC)	B2	B3
A1	desktop web crawler/ agents	pop server
A2	pure notification server	certain MUDs
A3		
A4	NSTP [16]	awareness protocol [15]

Figure 8 notification server taxonomy

Note that in options A2 and A3 the notification server and the data repository are separate while in options A1 and A4, the notification server has some knowledge of the data. Let us now look at each of the systems on the matrix in turn.

A desktop web crawler looks for changes on some shared files on a web server (A1). As soon as some updates occur it generates a list and informs (B2) the clients about the changes, perhaps via an email message.

A POP server watches for the data (A1) but it is activated when it receives a request (B3) from the mail client. Unlike a web crawler which asks for changes in a spontaneous fashion, the event which drives polling in a POP server may either be time driven or demand driven. By default, a POP server does not perform any automatic notification. It is only triggered by a certain event from a mail client. Therefore a POP server only acts as a weak notification server.

Case A2-B2 can be seen as a pure notification server as the server is entirely separate from the data store. The server is told (A2) about the changes from the active client and it then notifies (B2) the users' clients about them. This is similar to, for example, the locking mechanism in the UNIX file system where applications explicitly request locks on remotely stored files from a special process, the lock daemon (file d). However, the lock daemon has no control over the files it is referred to and thus it is logically distinct from the file store.

Certain Web MUDs would be of type A2-B3. It would involve a low level client such as a Java applet running on a web page and a rapidly polling notification server at a web site. If someone visits that site and requests the page, this tells (A2) the notification server that the page is being visited. So when the applet next polls the server (B3) other users see an avatar appear or may hear a door clink.

In NSTP [16] the notification server is closely coupled to the shared data repository (A4). In the event of any updates, the notification server tells (B2) the clients about the changes in the shared state information.

The awareness protocol proposed by Palfreyman et al [15] is stateless and is based on a client making a request and the server sending back a reply. The shared data is the awareness of the presence and the locations of individuals in virtual space. Therefore the awareness server is bound to the shared data (A4) and clients have to explicitly query (B3) the server.

Note that row A3 of the matrix in figure 7 is empty. Both cases A3-B2 and A3-B3 are ineffective. Case A3-B2 is particularly inefficient as it implies that the passive client would constantly have to ask for changes from the notification server and the latter would then have to send a request to the active client.

However, we are more likely to have an A3-B2 scenario where the notification server can be partially stateless. The server will have an interest in the changes without being fully aware of them. Such a situation may arise in a mobile environment where the server needs to avoid contention on the demands.

Location of Notification Server

For the sake of this discussion we effectively assumed a centralised architecture in that we have been talking about a single notification server and data repository. However, this is a conceptual architecture and the physical location of notification servers need not be centralised. The notification server may sit remotely from the data or it can be packaged within the data. Indeed there may be no single physical entity corresponding to the notification server, instead the notification service may be spread over several physical components as shown in figure 9.

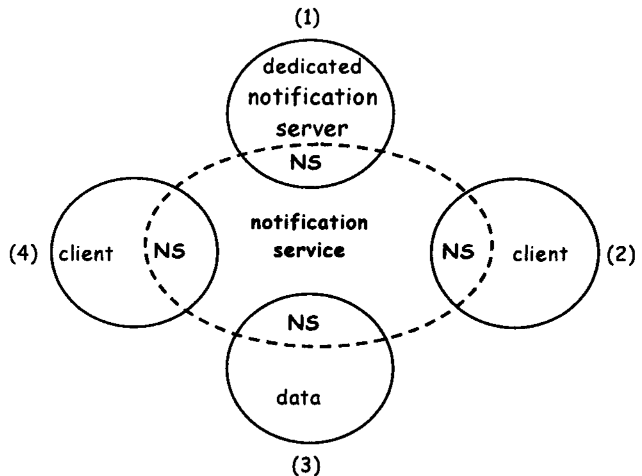

Figure 9 location of notification server

We can therefore identify 4 major options:

1. the notification server is *closely bound* to the data repository

Clearly, the notification server must reside in the same physical address space as the data store or the data must at least be part of the server. An example is a database supporting triggered actions. Instead of having the notification server explicitly asking for the changes, the data store informs the server about them.

2. the notification server and the data repository are *loosely coupled* together

In software engineering terms, the notification server is regarded as a separable component which may reside in the same physical address space as the data store but it which also sit somewhere else on the network.

3. a distributed *peer-peer* notification service

Often a conceptual notification server is realised as a software abstraction within the clients using peer-peer communication. An extreme example is AETHER [20], which percolates awareness information from node to node in a network thus effectively providing an emergent distributed notification service uniformly throughout the network.

4. a *hybrid* of the above

In practice, systems may include elements of all the above three options. For example, a single notification server may

be running on the network but a notification service component may be integrated within each client in order to provide an effective application interface.

NOTIFYING USERS

We are interested in the timeliness of information by providing rapid feedback and appropriate pace of feedthrough to collaborative users; hence the need for an underlying notification server which will provide such a level of user awareness. Although the notification server may use a particular mechanism at the protocol level, the user may perceive it in a different way at the behavioural level.

Low level software may poll reasonably rapidly when the client talks to the notification server but at a higher level of abstraction, the server could actively send messages to notify users despite the fact that it received those messages via polling. For instance, a notification server may watch for changes in web pages and then email a list of updated pages to the users.

Similarly, if we compare the true push technology (based for example on a Java applet) with a browser refreshing web pages based on expiry time then from the user point of view they may appear little different. However, at a lower level, the applet is informed by the 'pushed server' while the browser polls after the expiry date.

Layering

To understand these complex interactions we return again to status-event analysis. In status-event analysis we find that the mode and the pace of interaction may change radically at different layers between the software, the hardware and the human parts of the system. Consider the scenario when a person decides to press on a keyboard. When a key is pressed, the wire changes to a high or low voltage (status) and as soon as the chip notices the change it causes an interrupt (event) at the lowest level which in turn is processed at higher levels.

Similarly, the interaction between the user and the passive client creates an extra layer of indirection. Let us look at Lotus Notes as an example. By default, Notes does not actively notify users when the database has changed. It puts a mark against the changed notes in a list view. It is only when the user explicitly looks at the relevant list view that he/she becomes aware of the change. At the user level this corresponds to case 3 (asking) but at the lower level, the Notes server informs the Notes client when changes occur (case 2) so that it can update its local structures.

Alternatively, many mail clients will periodically poll the server (case 3) but when they notice that mail has arrived they inform the user by popping up a dialogue box (case 2). Even when low level protocols do not directly support the desired user level behaviour it is possible to provide different types of notification although this may be less efficient, for instance in terms of network traffic.

We can use different layers between client-notification server and client-user in order to achieve the desired pace of interaction.

IMPEDANCE MATCHING

We have so far discussed the various scenarios when notification takes place and we will now consider the frequency of notification. So how often must the notification server send updates to the users? This will depend heavily on the desired pace of interaction.

The pace of interaction is the rate at which users interact with computer systems, the physical world and with one another. Three major factors influence the pace of interaction: the pace of communication channels, the pace of the shared task and the pace at which the users operate. Unlike bandwidth, which gives a measure of the amount of information that is transmitted, the pace of channels indicates the frequency of communication. A previous study [8] highlighted the problems which may arise when there is a mismatch between the communication channel and the cooperative task.

Collaborative participants not only interact individually with the system but also with other group members via the shared objects. As a result, it is important to see both the user's own updates (feedback) and the effects of other users' actions (feedthrough). Feedthrough is a form of awareness [18]; it is an awareness of the effects of group members actions, hence the changes that have occurred. By observing the intermediate steps and the way the changes happen, we can also infer the reasons why the changes take place. The pace of the feedthrough is directly affected by the pace of the interaction.

Pace Impedance

Collaborative users often have to work with a large number of shared objects and it is not always possible to maintain an appropriate rate of feedthrough for each object, even over fast networks. Indeed, if we do try to obtain a maximum rate of feedthrough for all objects the resulting network congestion and computational load would undoubtedly mean delays for all objects including the most salient and important. We clearly need some form of *impedance matching* between the active and passive clients.

Impedance matching can be achieved by simply reducing the rate at which the active client generates update messages. For example, a shared cursor need not broadcast changes after each pixel movement. However, impedance matching at the active client level is global – all passive clients will receive the same pace. This implies that the passive clients must further filter the event stream. Although this will achieve appropriate user level feedthrough, it still consumes network bandwidth and computational effort.

As the notification server acts as an intermediary, it can be used to adjust the pace of feedthrough. It does not necessarily have to forward the changes to the passive client at the same rate that it received it from the active client. Consequently, the pace between the active client and the notification server (side A of figure 4) may differ from that between the notification server and the passive client (side B).

In order to obtain the right pace and the right granularity of the changes the clients will have to negotiate with the notification server. For example, at a user level, mailing lists distribute messages to subscribed users each time they hit the server. In contrast, moderated lists may send digests to users each month.

Volume Impedance

In addition to pace impedance, the volume of data sent to users can be adjusted to make it more manageable. For example, some moderated lists send users a list of the contributions main headings and if someone is interested in a particular article then he/she sends a request. A similar example arises in Lotus Notes and bulletin board systems. The user sees some mark against changed items, but not the full text of changes until requested. In fact, in many such systems the entire text has already been downloaded. So this is a case of volume impedance matching between the passive client and the user. Impedance matching between the server and client can be found in some POP email clients where only the headers of large messages are downloaded until the text is actually requested.

Both pace and volume impedance matching can be seen as a form of quality-of-service [17]. However, whereas most QoS is concerned with achieving minimum standards of throughput, we are trying to limit it.

Implementation Issues

If the notification server is bespoke then it may have an in-built knowledge of the appropriate pace of feedthrough for impedance matching purposes. However, in general, the information as to what pace of low-level events is needed to achieve appropriate user-level feedthrough will not reside in the notification server, but must be communicated to it from the clients.

The frequency of updates may be controlled either explicitly by the user or may be built into the client application. For example, certain objects may be regarded as *active* and require almost instantaneous feedthrough to be effective, whereas other objects may be less active and the rate of notification can be reduced accordingly.

In order to enable the notification server to provide effective impedance matching, the client needs to use some form of protocol to inform the server of the required pace. However, this communication between the client and the server does not require the server to have any knowledge of the application semantics.

Because the server is delaying feedthrough to the clients, some form of event queues must be held at the server end before they are sent across to the clients. This lays an extra storage load on the server. It also means that when updates are sent they have the accumulated size of all the delayed messages. Ideally the server should be able to compress the event queues, for example the event queue:

```
insert(" hello" ), insert(" world" )
```

can be reduced to:

```
insert(" hello world" )
```

However this requires the server to have substantial knowledge about the events and the objects.

SUMMARY

We have taken a general model of status change discovery from status–event analysis and applied it to notification server architectures. This highlighted the similarities between the communication (A) from the active client to the notification server and (B) from the notification server to the passive client. Grounding our framework for notification architecture on this generic theoretical analysis gives us confidence in its coverage.

The status–event analysis also highlighted the important distinction between the knowledge of *what* has changed in shared data and the knowledge *that* it has changed.

This analysis allowed us to construct a taxonomy based on issues of source and initiative within the three way PC–NS–AC communication.

We also saw that a conceptually single notification server may be placed in various physical locations within a CSCW system and may even be distributed over several components. In the latter case it may be more appropriate to think of a notification service operating within the system as a whole.

Notification servers operate at a low-level within the computer system, but their purpose is to provide user-level behaviour in the form of feedthrough and awareness. We looked at the way that different categories of notification may be seen at different levels so that a client application may make use of non-optimal low-level notification services to achieve acceptable user-level behaviour.

A crucial behavioural issue is achieving the appropriate pace of feedthrough and awareness. The central mediating position of the notification server allows it to perform impedance matching and thus reduce network bandwidth and improve user-level notification.

In order to demonstrate some of the issues discussed in this paper, we are constructing an experimental notification server called 'Getting to Know' (GtK) within a distributed agent architecture that supports impedance matching. The notification server operates at the same location in the taxonomy as NSTP, but in a more loosely coupled fashion thus allowing a separation of concerns between notification and data.

It is essential in any design process to be able to effectively compare and discuss different design options. We believe that the framework provided in the paper begins such a design vocabulary for the implementation of notification services.

REFERENCES

1. Abowd, G. and Dix, A., Integrating status and event phenomena in formal specifications of interactive systems, in *Proceedings of SIGSOFT'94* (New Orleans, 1994), ACM Press, 44–52.

2. Benford, S. and Fahlén, L. A spatial model of interaction in large virtual environments. *Proceedings of ECSCW'93*, Kluwer Academic. (1993). pp. 109–124.
3. Benford, S., Bowers, J., Fahlén, L., Mariani, J, Rodden. T, Supporting Cooperative Work in Virtual Environments. *The Computer Journal, 38*, 1 (1995).
4. Bentley, R. Supporting Multi-User Interface Development for Cooperative Systems, *PhD Thesis* (University of Lancaster, UK, 1994).
5. Brewster, S.A. Providing a structured method for integrating non-speech audio into human-computer interfaces. *PhD Thesis* (University of York, UK, 1994).
6. Brewster, S.A., Wright, P.C. and Edwards A.D.N. The design and evaluation of an auditory-enhanced scrollbar, in *Proceedings of CHI'94* (Boston, Massachusetts, 1994), ACM Press, 173–179.
7. Dewan, P, A tour of the Suite user interface software, *in Proceedings of UIST'90* (1990), ACM Press, 57-65.
8. Dix, A.J. Pace and interaction, in *Proceedings of HCI'92: People and Computers VII*, (Sept. York, 1992), Cambridge University Press, 193–208.
9. Dix, A. and Brewster, S.A. Causing Trouble with Buttons, in *Ancillary Proceedings of HCI'94* (Glasgow, UK, 1994). Available at http://www.soc. staffs.ac.uk/~cmtajd/papers/ buttons94/
10. Dix, A. and Abowd, G. Modelling status and event behaviour of interactive systems. *Software Engineering Journal 11*, 6 (1996), 334–346.
11. Dix, A. and Abowd, G. Delays and Temporal Incoherence Due to Mediated Status–Status Mappings, *SIGCHI Bulletin 28*, 2 (1996), 47–49.
12. Dix, A., Finlay, J., Abowd, G., and Beale, R. *Human–Computer Interaction*. Prentice Hall (second edition 1998). Available at http://www.hiraeth. com/books/hci/
13. Hill, R.D., Brinck, T., Rohall, S.L., Patterson, J.F. and Wilner, W. The Rendezvous architecture and language for constructing multi-user applications. *ACM Transactions on Computer-Human Interaction, 1*, 2 (1994), 81–125.
14. Lauwers J.C. and Lantz K.A. Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems, in *Proceedings of CHI'90* (Seattle, Washington, April 1990), ACM Press, 303–311.
15. Palfreyman, K. and Rodden, T. A Protocol for User Awareness on the World Wide Web, in *Proceedings of CSCW'96*, (Boston, Massachusetts, Nov. 1996), ACM Press, 130–139.
16. Patterson, J.F, Day, M. and Kucan, J. Notification Servers for Synchronous Groupware, in *Proceedings of CSCW'96* (Cambridge Massachusetts, 1996), ACM Press, 122–129.
17. Rada, R. *Interactive Media*. Springer-Verlag, New York (1995).
18. Ramduny, D. and Dix, A. Why, What, Where, when: Architectures for Cooperative Work on the World Wide Web, in *Proceedings of HCI'97*, (Bristol, UK, Aug. 1997), Springer-Verlag, 283–301.
19. Rodden, T. Populating the Application: A Model of Awareness for Cooperative Applications, in *Proceedings of CSCW'96*, (Boston, Massachusetts, Nov. 1996), ACM Press, 87–96.
20. Sandor, O., Bogdan, C and Bowers, J. Aether: an awareness engine for CSCW, in *Proceedings of ECSCW'97*, (Lancaster, UK, 1997), Kluwer Academic, 221–236.
21. Trevor J., Mariani J. Rodden T. The use of adaptors to support cooperative work, in *proceedings of CSCW'94* (Chapel Hill, North Carolina, Oct. 1994), ACM Press, 22–26.
22. Wood, A., A. K. Dey and G. D. Abowd (1997). CyberDesk: Automated Integration of Desktop and Network Services, in *Proceedings of CHI '97*, ACM Press, 552–553.
23. Wood, A. CAMEO: Supporting Agent-Application Interaction, *PhD Thesis* (University of Birmingham, UK, 1998).