Impedance Matching: When you need to know What

Devina Ramduny and Alan Dix

Computing Department SECaMS, Lancaster University, Lancaster, LA1 4YR, UK

Tel: *+44 1524 593490*

Fax: *+44 1524 593608*

Email: *devina@comp.lancs.ac.uk*

Email: *alan@hcibook.com*

URL: *http://www.hcibook.com/alan/topics/web*

Feedthrough and awareness of user activities are major issues in CSCW. A key difference between awareness and goal-directed feedthrough lies in the required pace and quality of feedthrough. Getting the right pace of feedthrough is important for usability and to avoid overloading networks. Notification mechanisms should therefore allow dynamic tuning of the pace and volume of update events. This matching of the required and supplied pace of update events we call impedance matching. A separable notification server is often ideally placed to perform impedance matching between end-user clients. These design principles have been applied in an experimental notification server Getting-to-Know.

Keywords: awareness, feedthrough, notification mechanism, separable notification server, event notification, pace, CSCW infrastructure.

1 Introduction

Shared interactive networked interfaces present a large number of objects and artefacts through which collaborative users can interact with. Feedthrough (Dix, 1994) – the ability for one participant to sense the effects of another's actions – is essential in such interfaces to promote awareness and facilitate the coordination between users' activities. The faster the rate at which the updates are communicated to the users, the higher is the pace of feedthrough.

In the real world, the feedthrough between participants is mediated by the physical properties of artefacts and space, but in distributed electronic environments, some sort of event or notification needs to propagate through the network so that applications can inform users about remote events.

Our concern in this paper lies in the underlying computational infrastructure that enable systems to support feedthrough, and in particular, in the requirements and design of notification servers. A notification server is a piece of software whose task is to relay the fact that changes in data or other events have occurred. Others have proposed notification servers that are tightly bound to the data they regulate (e.g. NSTP (Patterson et al., 1996)). However, we have previously argued that notification servers should be regarded as separate entities – certainly at a conceptual level and often physically distinct – and this was used as a design driver for our experimental notification server GtK, Getting-to-Know (Ramduny et al., 1998).

As discussed in (Dix, 1992), different types of task require a different pace of feedthrough. This may be a fraction of a second or hours or days. For example, in Pausch's 'Virtual reality for five dollars a day' he found that rapid feedback of low fidelity wireframe models was far better than slower photorealistic rendering (Pausch, 1991). Delivering feedthrough at the wrong pace can be problematic. If it is too slow, users may have to act without up-to-date knowledge of one another's actions. If it is too fast, users may be distracted by irrelevant changes.

Some feedthrough is very goal-directed – information directly used by users in their tasks. However, the collaboration literature constantly emphasises the value of awareness (Dourish & Bellotti, 1992). However, whereas goal-directed activity usually requires detailed and timely feedthrough, awareness is typically longer term and more 'fuzzy'.

For implementation, both the different feedback pace requirements and the difference between goal-directed feedthrough and awareness are largely about quality of service (QoS) (Rada, 1995).

These differences in QoS lead us to conclude that notification servers should be able to modify the rate and quality of notification to match the required feedthrough at the user interface. This facilitates the development of client applications that require rapid detailed feedback for goal-directed activity while supporting lower pace and lower granularity notification for awareness purposes. We call this matching of pace and quality, 'impedance matching' and demonstrate its application in the design of our notification server, GtK.

Due to the limitations on network bandwidth and computer resources, rapid feedthrough for all the shared objects is not always possible in collaborative interfaces. Indeed, even if a maximum rate of feedthrough is provided for each shared object over fast networks, further network congestion will arise. The extra computational load would undoubtedly imply additional delays for all the objects including the ones of higher interests to the users.

Matching the pace and quality of notification to the feedthrough required at the user interface means that we can reduce unnecessary network traffic, thus reducing 'wasted' use and making unintended delays less likely.

In summary, we have three main requirements for notification mechanisms for feedthrough and awareness:

(a) the notification server should be a separate component
(b) it should be possible to control the pace of notification for awareness
(c) it should be possible to control the quality/fidelity of the notified information

In this paper we will focus on the second requirement. The first of these has been discussed in detail in a previous paper (Ramduny et al., 1998) and the last we will leave for future work.

Section 2 shows how notification servers as mediators are ideal for supporting impedance matching by controlling the pace of feedthrough. Section 3 examines how feedthrough demands can be reduced by subsequently reducing the pace and the volume of updates. Section 4 explores the potential triggers for pace impedance by analysing their effect on event propagation. Section 5 introduces the GtK notification server and Section 6 shows how GtK has been augmented to support impedance matching. Finally, section 7 considers some outstanding issues that arise from impedance matching.

2 Where to control pace of feedthrough

As we have already argued, delivering feedthrough that is effective for the user and efficient for the system requires impedance matching to control the pace of feedthrough. However, the best place where this should occur is not obvious. In this section, we will justify our choice of placing impedance matching within the notification server.

2.1 Interaction without notification server

For the purpose of this argument, we will assume that each user is interacting through a single client device. For any change, update or user action we can consider the client of the user who initiated the action, the active client, and the clients of the rest of the users who receive feedthrough, the passive clients.

In the absence of the notification server, the active client is responsible for propagating changes tive clients. This can either happen through a broadcast mode (figure 1(a)) or through a point-to-point interaction between the clients (figure 1(b)).

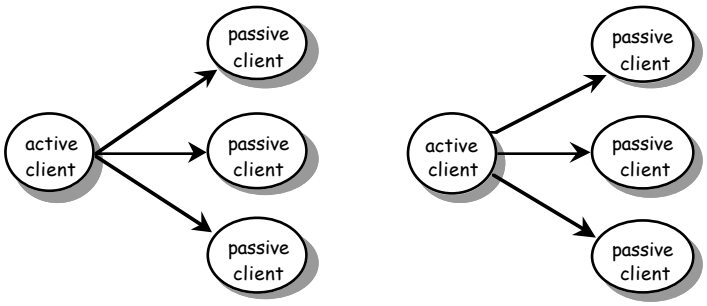

Figure 1. (a) broadcast mode of interaction **1. (b)** point-to-point interaction

In the first case (figure 1(a)), all passive clients receive the same notification events. This means that events have to be delivered at the rate of the fastest client, and any per-user impedance matching has to happen at the passive-client end.

Consider the example of a shared drawing package. All the users may not be actively involved in manipulating the shapes and their sizes on the screen at the same

time. So, when changes to the shared cursor are broadcast, the active client must broadcast each pixel movement to everyone, for the sake of the few users who are currently interacting with the particular object.

Although the passive clients can ignore unnecessary events, this consumes additional network bandwidth and computational effort.

In the peer-to-peer form of interaction (figure 1(b)), the active client maintains a separate channel with each passive client. Consequently, the event stream can also be filtered on a per-client basis by the active client. This form of interaction enables each passive client to receive different rates of feedthrough, but at the expense of some fairly complex filtering mechanism at every active client.

2.2 Interaction with notification server

The presence of the notification server allows both broadcast and point-to-point mode of interaction (figure 2). The notification server is the central point of contact between the active clients and the passive clients. The active clients send the changes to the notification server (broadcast) and it in turn can act as the mediator to adjust the rate that each passive client receives the updates independently (point-to-point).

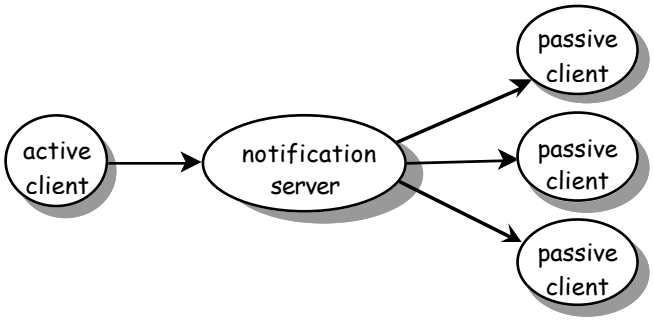

Figure 2. notification server as mediator

The notification server does not necessarily have to forward the changes to each passive client at the same rate that it received it from the active client. The pace of feedthrough between the active client and the notification server will therefore differ from that between the notification server and the passive client. So, 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 every month.

A bespoke notification server may have an in-built knowledge of the suitable pace of feedthrough required. But in general, the information as to what pace of low-level events is required to achieve appropriate user-level feedthrough will not reside in the notification server; the clients must communicate that information to the notification server.

3 Impedance Matching policies

Impedance matching embodies both the volume of updates and the rate at which the updates are notified to the users (Ramduny et al., 1998). The feedthrough demands can therefore be reduced by:

〈 sending updates less often (pace impedance)
〈 sending less in each update (volume impedance)

Pace impedance deals with the frequency or the rate of notification while volume impedance influences the amount of updates that is transmitted to the user. A reduction in the rate of notification and in the volume of changes sent to the users can in fact cause an implicit gain on the resources. But this calls for a certain amount of filtering to be carried out.

3.1 Pace Impedance

The rate at which updates are sent out can be reduced by:

(a) sending information less often

The updates are buffered and communicated to the users when it is more convenient to them. All the information gets sent, including details such as the header, destination and so on. Only the rate at which the information is sent is affected.

(b) sending chunks of information

The information is sent in chunks to improve the overall performance. The size of the chunks or the frequency at which the chunks are transmitted can be reduced. This may cause a loss of information in some cases, but can be advantageous for lowering network overheads. For instance, message headers need not be transmitted each time messages are broadcast.

3.2 Volume Impedance

In addition to pace impedance, the volume of updates can be adjusted to make it more manageable to the users. The desire to reduce network bandwidth already puts some constraints on what users can see and how often they see them. Depending on the task, the amount of information sent across could be dropped to a minimum level and yet still be acceptable to the users. Users could thus receive a faster response time and the application could cope with high network traffic. However this should not jeopardise the quality of information broadcast.

One example of this is the use of flags marking new or changed material. Flags convey awareness information at a reduced level of detail. By their very nature they are low volume, but also extreme timeliness is rarely critical.

3.3 Impedance matching v/s QoS

Quality of Service (QoS) (Rada, 1995) ensures that the network channel has sufficient quality available to provide a better service for data transmission. This is crucial for maintaining a continuous transmission of audio, high-bandwidth video and multimedia information. QoS caters for delays and any necessary adjustments caused by the variable latency of the received data. QoS-based models also support the self-pacing of real-time data thus enabling data to be transmitted without any distortion.

When QoS is applied, for instance in the transmission of video images, the images are sent in chunks to reduce the frame rate, thus acting as a form of pace impedance. The images are also very often compressed and sent at a lower resolution and this is similar to volume impedance. So both pace and volume impedance matching can be seen to be a form of QoS. However, whereas most systems based on QoS are concerned with achieving minimum standards of throughput, the main motivation behind impedance matching is to determine whether the service can be limited to fit the data at our disposal.

4 Exploring Pace Policies

For the following discussion, we will assume a simple client-server mode of interaction where user agents send messages to each other through a central server. Messages sent across the network are usually transmitted as events at the lower level. We will explore the different ways of obtaining pace impedance and show their effects on the flow of events through the use of time-space diagrams (Lamport, 1978).

Figure 15 shows the ordering of events on a time-space diagram. The horizontal direction represents space whereas the vertical direction indicates time in ascending order, with later events being shown higher than earlier ones. The dots represent events and the horizontal lines represent messages (m). Note that any latency in the network itself is not shown, as this is not a significant feature in the examples considered below.

If the network connection between the client and the server is instantaneous, figure 3 shows the ordering of events when no impedance matching is applied. The server forwards each message it receives following an event immediately to the user agent.

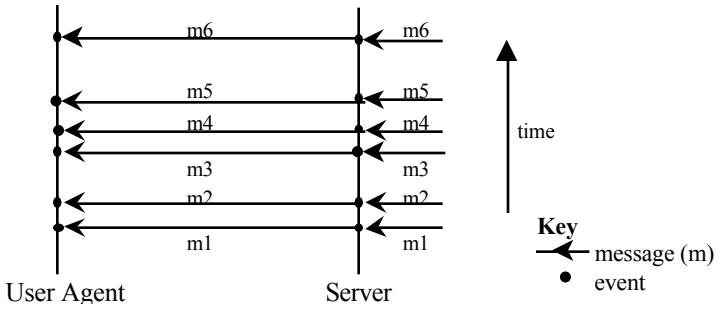

Figure 3. without impedance matching

As pace impedance is about sending information less often, one could ask the question: how often should the messages be transmitted? Surely, there must be some kind of event that acts as a trigger, which causes the messages to be sent. The potential triggers for pace impedance are: the time factor, the volume of the message and the size of the message.

4.1 Fixed time interval

The client receives messages after every fixed time interval (t). The messages are buffered at the server-end until time t is reached, in which case the messages are transmitted to the client in a single stream. In figure 4 for example, the first message stream consists of both messages m1 and m2 but only m6 is sent out in the third message stream.

Because the time interval is fixed, the client can in fact poll the server. A classic example occurs in a mail system, where the client polls for changes from the mail server at regular intervals.

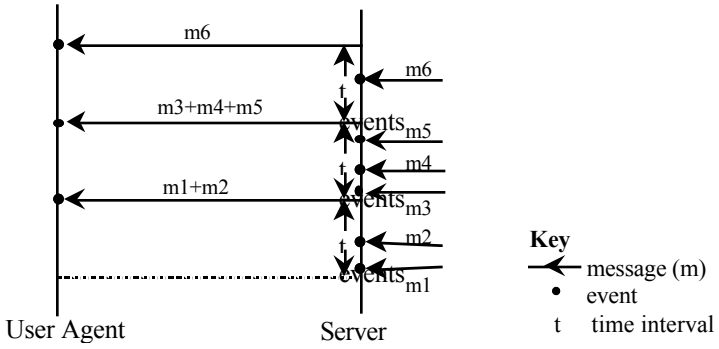

Figure **4.** fixed time interval

4.2 Time delay

This option varies slightly from the previous one. Instead of sending events after every fixed time interval, an event is only generated after a time delay is reached. So, when the server receives the first message, it starts the timer and the messages are buffered until a certain delay () is reached, after which all the messages received are transmitted in a single stream to the user agent. The timer starts again when the next message hits the server as shown in figure 5.

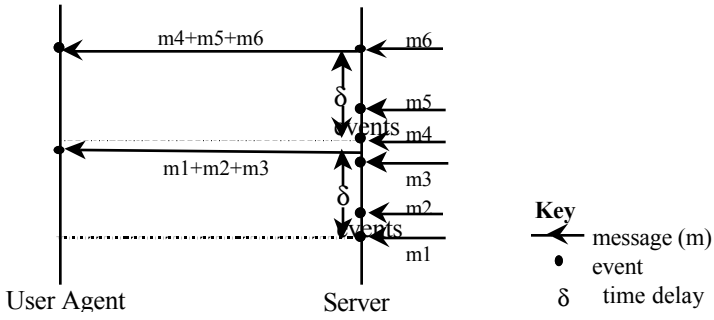

Figure 5. time delay

Unlike the previous case, this option is more server-based in that the server takes the initiative to generate events. The clients rely on the server to push messages towards them, as they have no knowledge of when the server actually starts counting the delay.

4.3 Volume of messages

In this case, the volume of the message acts as the trigger. The server buffers the messages until a maximum number of outstanding messages have been received, which are then sent out to the client in a single stream. Figure 6 shows the user agent receiving an event after the server has received a maximum of three messages.

This mode of pace impedance could be found in a shared text editor where it is not always effective to transmit all the keystrokes. The server could wait until a maximum number of keystrokes are received before sending them.

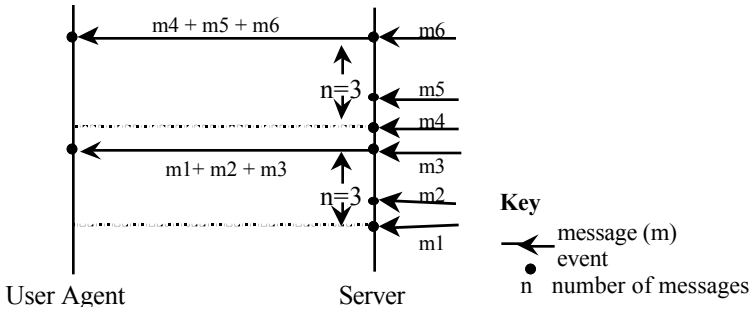

Figure 6. volume of message

4.4 Message size

With this option, the server forward messages to the clients once a maximum size is reached for it is not always effective to send several gigabytes of messages. Figure 7 shows how the server send messages to the client in a single stream, once the maximum limit (max) has been reached. If the size of the message is below the maximum value, the message is kept in a queue and subsequent messages are added onto it until (max) is reached.

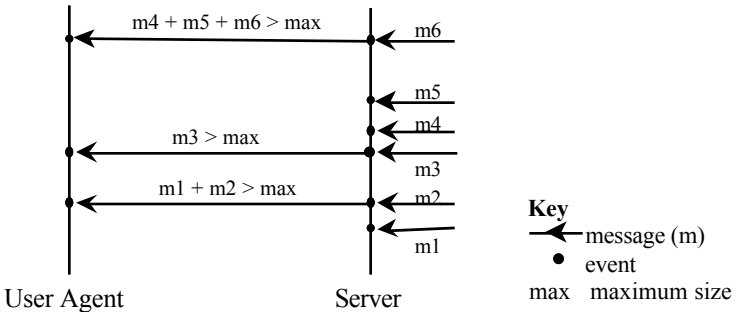

Figure 7. message size

5 Getting-to-Know notification server

The GtK server is built over several layers of custom and standard infrastructure, as shown in figure 8.

At the base lies the standard low-level Internet TCP/IP protocol accessed via Java networking classes.

On top of these there is a custom event management layer which supports directed message delivery between agents on different physical machines.

Finally, GtK itself uses the event manager to receive notifications about changes from active clients or information servers and then passes on the notifications about those events to the passive clients.

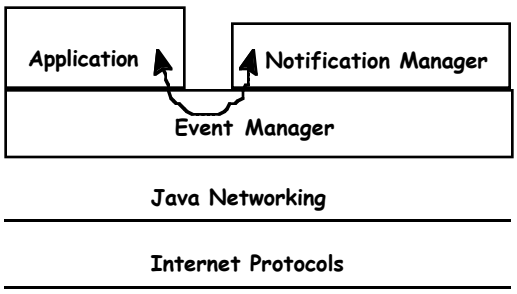

Figure 8. GtK infrastructure

5.1 Messaging and event layer

The event management layer implements a distributed asynchronous messaging protocol giving GtK a uniform, generic location-independent event model. Whereas TCP/IP gives point-to-point messaging between the application processes, the Event Manager allows point-to-point asynchronous messaging between different

communication objects in the same or in different address spaces. All messages and events are of the simple form:

```
sender reference : recipient reference : event_type : data
```

Applications add their own semantics for the uninterpreted ASCII `data` but utility classes are provided to enable standard argument marshalling.

Although the implementation of the event management layer is in Java, an ASCII protocol has been developed for message passing instead of using Java's Remote Method Invocation (RMI). This was due to several reasons. Firstly, at the time the development was started, Java RMI did not have solid foundations. Secondly, RMI is synchronous and thus does not fit closely to our preferred asynchronous event model for distributed agents (for various reasons, but mainly because our model is closer to the way modern UI code works). Thirdly, RMI depends on Java serialisation, which in our experience is not robust in web environments, where different versions of the Java code may co-exist. Finally, the use of an ASCII based asynchronous messaging protocol makes it easier to add non-Java clients and servers.

5.2 Notification manager

The distributed object layered infrastructure enables the Notification Manager to know about every other object. The Notification Manager can be controlled directly through message calls or remotely via the messaging layer. It uses the same event model as the messaging infrastructure, but also allows optional translation of event types.

The notification server provides three main functions:

- add interest – tells the server that a specific network object wants to know about specific events for a second network object
- remove interest – tells the server to cancel some or all of the interests for a given object
- tell all – asks the server to broadcast an event to all interested objects

The above three functions together with a few additional housekeeping operations allow the expression of a wide range of different application specific notification strategies. They are similar to the facilities offered by the Java AWT Observer/Observable classes and AWT 1.1 event listener model except that these Java events are limited to a single Java process.

GtK maintains an interest table, which keeps a list of interested clients for specific objects. Each object in the interest table has one or more recipients as shown in figure 9.

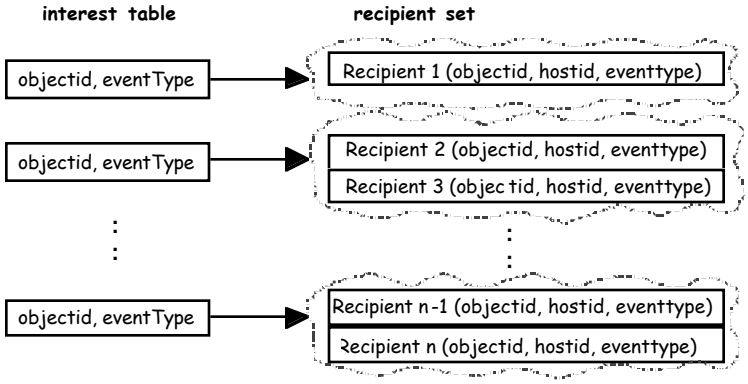

Figure 9. relationship between interest table and recipient set

The interest table is updated based on the functions 'add interest' and 'remove interest'. When an object asks GtK to "tell all", GtK first matches the event type and objects with the interest table and then passes on the event (with optional type translation) to all the interested clients.

5.3 Physical Location

GtK requires that the data invokes the 'tell all' function to inform clients of any updates. Unlike NSTP (Patterson et al., 1996) GtK is only loosely coupled to the data repository. GtK knows that data objects exist and that other objects are interested in them, but it has no other application knowledge. Similarly, the data objects have to inform GtK by using 'tell all', but they need not be aware of other remote or local interested objects. GtK thus achieves a separation of concerns between the notification server and the data repository.

GtK can run in the same address space as the data objects, on the same server or on a completely different server. In addition, the separation of concerns implies that GtK can support several heterogeneous data sources, potentially on different physical servers.

6 Augmenting GtK for Impedance Matching

In this section, we will briefly describe how our separable notification server GtK has been augmented to provide pace impedance using some of the principles discussed in section 4. Although impedance matching emerges from an abstract notion, it has been implemented in GtK to investigate its actual behaviour on a practical level (Ramduny, 2002).

The pace of feedthrough can be reduced by:

- setting a certain limit (be it fixed or variable) on the volume of updates and
- setting a time interval between the propagation of the updates.

6.1 Pace parameters

Two pace parameters have therefore been defined:

```
queueLength // length of queue
```

`queueLength` allows a client object to specify the maximum number of messages or events that can be placed in a queue before they are broadcast (volume trigger section 4.3).

```
time      // duration
```

`time` enables a client object to specify the time interval that triggers successive messages to be displayed on the screen (time interval trigger section 4.1).

These are combined into a single data structure:

```
Frequency = (queueLength:time)
```

Events are buffered to a queue and removed, depending on the values of `queueLength` and `time`. The default value for `queueLength` is `0`, which implies an empty queue. The default value for `time` is `−1`, which denotes infinity.

For example, a `Frequency` of (0,3) indicates that messages are buffered and sent out every 3 seconds whereas a `Frequency` of (10, -1) implies that messages are sent out in batches of 10.

Items are flushed from the queue when either:

- current queue length value set for `queueLength`.
- current delay value set for `time`.

  where current delay (current system time time first event was queued.

The next time the queue needs to be flushed = (time first event was queued + `time`).

The need to support impedance matching brings about certain changes to the main functions of GtK. In addition to keeping track of an object and its recipients, the interest table now has to be aware of each recipient's frequency (figure 10).

As every remote network object is associated with a frequency, the 'tell all' function can provide different rates of feedthrough to the clients. So when broadcast events are received, "tell all" has to first ask the server to add the event for each interested object to its recipient's queue. The server also records the time the event is added to the queue and "tell all" then checks to see if it is time to flush any event from any recipient's queue.

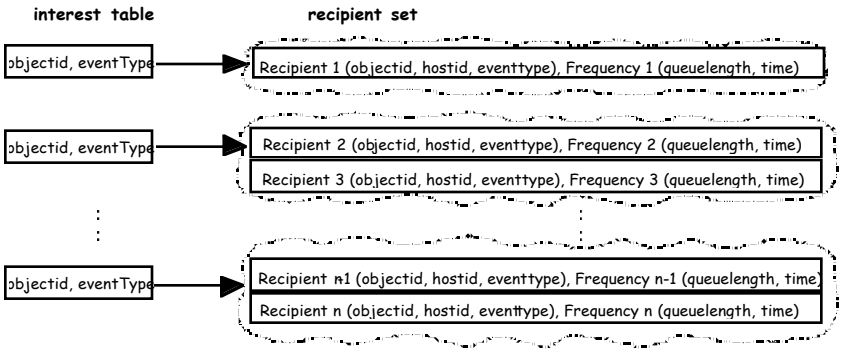

Figure 10. updated link between interest table and recipient set

6.2 Event Queue management

A FIFO queue is maintained to store each recipient's associated queue of events, such that an event is added to the rear of the queue and the first or oldest event is at the front of the queue and always the first to be removed.

An alarm is started as soon as events are placed in a queue. Events are kept in the queue until:

(a) a broadcast event is received
(b) when the alarm rings

Both triggers will prompt "tell all" to check if any recipient queue is ready to be flushed, in other words, if the item at the front of the queue meets the pace parameters limit. If the limit has been reached, the item is flushed out of the respective recipient's event queue.

6.3 Altering pace parameters

In order to provide an acceptable pace of feedthrough that matches the users' task at hand, clients should be able to adjust the rate at which they receive updates from the notification server.

A client object can therefore change the frequency with which it wants to be notified of any updates by sending a "`change frequency`" event together with the new values for `queueLength` and `time`. This will also alter the frequency of each recipient for that client object.

Note that different interface objects may require different pace of feedthrough. For example, focus objects such as the top-most window typically require a faster rate of feedback from background or iconised windows. Therefore GtK allows the setting of frequency parameters on a per object basis. Furthermore, the required pace will vary dynamically, for example, when a new object is made visible or a window is popped to the front. This is why GtK separates registering interest, typically once per object, from setting frequency which may happen repeatedly.

7 Discussion

The implementation of impedance matching generates some outstanding issues and these are discussed below.

7.1 Impedance matching in VR and visualisation

The notion of impedance matching can be found in other systems. Although the mechanism adopted in such cases is not explicitly called impedance matching and it has been employed to achieve different purposes, it does satisfy a similar functionality.

The collaborative model of awareness based on the spatial interaction of objects (Benford & Fahlén, 1993) lies on concepts of aura, nimbus and focus. Aura is a volume in space that delimits the presence of a particular object. Focus represents the objects in space that a user is interested in while nimbus represents the space controlled by those objects. The quality of information transmitted is said to depend on the level of awareness a user has of an object and this is negotiated through focus and nimbus. The role of the focus and nimbus are in fact similar to that of the focus objects. The closer the focus and nimbus, the greater is the level of awareness, hence the higher is the quality of information transmitted.

This model has recently been augmented with third party objects (Benford et al., 1997), which use aggregation to perform volume impedance in collaborative virtual environments, whereby a reduced level of detail is presented to the users without sacrificing the quality of the information. In order to manage the volume of data in such a complex environment, objects are grouped together and aggregate views of those objects are provided which expand further when they are selected (Ingram et al., 1996). This technique is also widely used in information visualisation of large data sets (Ellis et al., 1994) to make the data more manageable to the users.

7.2 Impact of rich media

Although not discussed in this paper, our applet-based real-time web conferencing application implemented using GtK (Ramduny, 2002) only allows information to be exchanged in a textual mode. However, some chat systems like ICQ1 also enable users to exchange communication verbally via voice-over-IP through the use of Internet phone such as BuddyPhone2. It is therefore essential that the rate at which information is exchanged through the different channels be kept in synchronicity to avoid a breakdown in communication.

When users talk through the phone while typing, the granularity of feedthrough becomes very fine-grained character level instead of words or sentences. Consequently, the task of matching the rate of feedthrough between the two channels is not trivial. Furthermore, the introduction of additional media such as real-time graphics and video adds more demands on the resources and thus making the provision of feedthrough even more problematic.

---

1 http://web.icq.com/

2 http://www.buddyphone.com/

Impedance matching is therefore required to manage the rate of feedthrough between the different channels. Some systems already provide a form of impedance matching to cope with the demands on bandwidth. For example, in media space systems such as Rave (Gaver et al., 1992) the video transmission is kept to a low volume and a low pace until a user actually clicks on the video, in which case the rate of feedthrough increases.

The solution adopted in Xerox Portholes (Dourish & Bly, 1992) makes use of frame-grabbing software for each media space and then distributing low-resolution digital images.

Similarly, in NYNEX Portholes (Lee et al., 1997) the WebCam operates at a slow rate, but the images are transmitted at full speed. An integrative view of a particular group is represented through a matrix of still video images, which are snapped periodically for instance after every five minutes.

7.3 Ordering of events

A major problem that impedance matching gives rise to relates to the ordering of events. Collaborative systems produce a large number of events of different kinds from several users at varying times. The order in which the events are broadcast may be critical in maintaining the cooperative activity. If users do not receive the events in the right order, they can easily get confused and in the worst case, they may abandon the task completely.

Let us consider the effect of impedance matching on the flow of events in a conferencing system. Figure 11 illustrates the point-to-point ordering of events between two users, John and Mary chatting on two conferences, VRML and FILE MAKER.

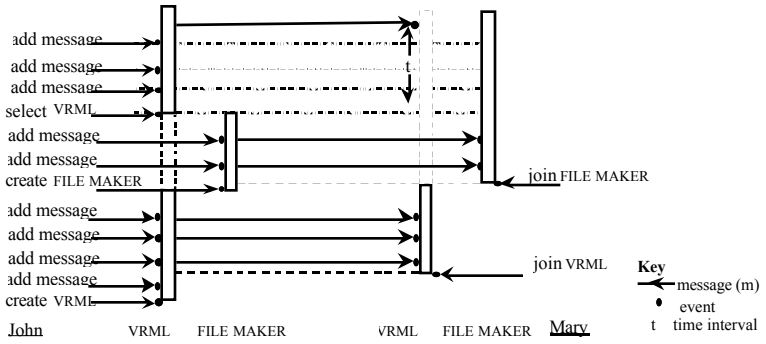

Figure 11. timing diagram with point-to-point ordering

Starting with the lowest event, the timing diagram shows that John first creates the VRML conference and adds some messages to it. Mary joins the VRML conference shortly after. John and Mary receive an instant feedthrough of each other's messages at that point, as they are both focussed on the same conference.

John goes on next to create the FILE MAKER conference, which Mary joins later. FILE MAKER now becomes the focus for both participants and conversations exchanged at that level have a higher pace of feedthrough than those in VRML.

At some stage, John decides to go back to the VRML conference while Mary is still active on the FILE MAKER conference. John now receives an instant feedthrough of all the messages added to the VRML conference but Mary only gets a set of buffered messages at a regular time interval (t). Given the different rates of feedthrough for the VRML conference, the order in which John and Mary receive the messages may differ and therefore run the risk of becoming inconsistent. The problem is amplified if there are some semantic dependencies between the messages.

A possible solution to deal with the inconsistent ordering of messages is to take a selective stance and delay all the messages until a certain time is reached and then send them out in the right order. But this measure will raise additional issues at the user interface level.

In a non-interactive system, this is only a problem if there are dependencies between the computational objects receiving the events and there are known ways of detecting this (Lamport, 1978). However, in interactive systems there are additional dependencies, as the user can see the effects on different objects, but the computer regards them as being distinct.

8 Summary

The central point of this paper is the analytic framework for impedance matching, which can be applied to augment other notification mechanisms or build new notification servers over different low-level messaging infrastructures. GtK is largely an example to show that these principles for notification server design can be achieved in a practical implementation.

We have explored the issue of pace impedance matching – the use of the notification server as an intermediary to match the required pace of updates of a passive client with the supplied rate of the active clients. Different parameters for regulating pace were considered, based on both time delay and number of outstanding updates.

Our experimental notification server, Getting-to-Know, is a 'pure' separable notification server that also supports pace impedance matching through maximum delay time and queue length parameters. Although not described in this paper, the practicality of the approach has been further explored by using GtK to build an example real-time conferencing system. Also, volume impedance can be supported within the given infrastructure but it has not been actively implemented in this work.

To our knowledge, GtK is the only extant notification service embodying true separability from data and pace impedance matching.

References

Benford, S. & Fahlén, L. (1993), A spatial model of interaction in large virtual environments, *in Proceedings of the third European Conference on Computer Supported Cooperative Work, ECSCW '93,* September 1993, Milan, Italy Kluwer Academic Publishers, pp.109–124.

Benford, S., Greenhalgh, C., Lloyd, D. (1997), Crowded Collaborative Virtual Environments, *in Proceedings of Human Factors in Computing Systems, CHI'97*, March 1997, Atlanta, Georgia. ACM Press, pp. 59–66.

Dix, A.J. (1992), Pace and interaction, *in Proceedings of Human Computer Interaction: People and Computers, HCI'92,* September 1992, York, Cambridge University Press, pp.193–208.

Dix A.J. (1994), Computer-supported cooperative work – a framework, *in* D. Rosenburg & C. Hutchison (eds.), *Design Issues in CSCW*, Springer Verlag, pp.9–26.

Dourish, P., Bellotti, V. (1992), Awareness and coordination in shared workspaces, *in Proceedings of Computer Supported Cooperative Work, CSCW'92*, Toronto, Canada, CSCW'92, ACM Press, pp.107–113.

Dourish, P. and Bly, S. (1992), Supporting Awareness in a Distributed Work Group, *in Proceedings of Human Factors in Computing Systems,* CHI'92, Monterey, CA, ACM Press, pp.541–547.

Ellis G.P., Finlay J.E., Pollitt A.S (1994), HIBROWSE for Hotels: bridging the gap between user and system views of a database, *in IDS'94 2nd International Workshop on User Interfaces to Databases*, April 1994, Ambleside, UK, Springer Verlag Workshops in Computer science, July 1994, pp.45–58.

Gaver, W., Moran, T., MacLean, A., Lövstrand, L., Dourish, P., Carter, K., Buxton, W. (1992), Realizing a Video Environment: EuroPARC's RAVE System, *in* P.Bauersfield, J.Bennett and G. Lynch (eds.), *Proceedings of Human Factors in Computing Systems, CHI'92*, ACM Press, pp27–35.

Ingram, R.J., Benford, S.D., Bowers, J.M. (1996), Building Virtual Cities: Applying Urban Planning principles to the Design of Virtual Environments, *in Proceedings VRST'96,* (, July 1996, Hong Kong, ACM Press, pp.83–91.

Lamport, L. (1978), "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, **21**(7), 558–565.

Lee, A., Girgensohn, A. Schlueter, K. (1997), NYNEX Portholes: Initial User Reactions and Redesign Implications, *in Proceedings of the International Conference on Supporting Group Work, GROUP'97,* Phoenix, AZ, ACM Press, pp.385–394.

Patterson, J.F, Day, M. and Kucan, J. (1996), Notification Servers for Synchronous Groupware, *in Proceedings of Computer Supported Cooperative Work, CSCW'96*, November 1996, Boston, Massachusetts, ACM Press, pp.122–129.

Pausch, R. (1991), Virtual reality on five dollars a day, *in* S.P. Robertson, G.M. Olson, and J.S. Olson (eds.), *Proceedings of Human Factors in Computing Systems, CHI'91,* Addison Wesley, pp.265–270.

Rada, R. (1995), *Interactive Media*, Springer-Verlag, New York.

Ramduny, D., Dix, A. and Rodden, T. (1998), Exploring the design space for notification servers, *in Proceedings of Computer Supported Cooperative Work, CSCW'98*, November 1998, Seattle, Washington, ACM Press, pp.227–235.

Ramduny, D. (2002) "Frameworks for Enhancing Temporal Interface Behaviour through Software Architectural Design", *PhD thesis in final preparation*, Staffordshire University, UK.