Frameworks for Enhancing Temporal Interface Behaviour through Software Architectural Design

Devina Ramduny-Ellis

A thesis submitted in partial fulfilment of the requirements of Staffordshire University for the degree of Doctor of Philosophy

December 2002

Abstract

Frameworks for Enhancing Temporal Interface Behaviour through Software Architectural Design

Devina Ramduny-Ellis

A thesis submitted in partial fulfilment of the requirements of Staffordshire University for the degree of Doctor of Philosophy

December, 2002

The work reported in this thesis is concerned with understanding aspects of temporal behaviour. A large part of the thesis is based on analytical studies of temporal properties and interface and architectural concerns. The main areas covered include:

(i) analysing long-term human processes and the impact of interruptions and delays

(ii) investigating how infrastructures can be designed to support synchronous fast pace activity

(iii) design of the Getting-to-Know (GtK) experimental notification server

The work is motivated by the failure of many collaborative systems to effectively manage the temporal behaviour at the interface level, as they often assume that the interaction is taking place over fast, reliable local area networks. However, the Web has challenged this assumption and users are faced with frequent network-related delays. The nature of cooperative work increases the importance of timing issues. Collaborative users require both rapid feedback of their own actions and timely feedthrough of other actions.

Although it may appear that software architectures are about the internals of system design and not a necessary concern for the user interface, internal details do show up at the surface in non-functional aspects, such as timing. The focus of this work is on understanding the behavioural aspects and how they are influenced by the infrastructure. The thesis has contributed to several areas of research:

(a) the study of long-term work processes generated a trigger analysis technique for task decomposition in HCI

(b) the analysis of architectures was later applied to investigate architectural options for mobile interfaces

(c) the framework for notification servers commenced a design vocabulary in CSCW for the implementation of notification services, with the aim of improving design

(d) the impedance matching framework facilitate both goal-directed feedthrough and awareness

In particular, (c) and (d) have been exercised in the development of the GtK separable notification server.

Acknowledgements

I would like to thank my supervisor, Alan Dix, for his guidance and support throughout the duration of this research. His critical comments on drafts of this thesis have been invaluable.

I would also like to share this achievement with my parents and parents-in-laws who have been wondering what I have been doing for so many years.

Finally, a special thank you to my husband, Geoffrey, for his patience and encouragement. I am also very grateful for his useful suggestions on the final draft of this thesis.

Table of contents

List of figures

List of tables

Chapter 1 Introduction

The rapid growth in worldwide communications has enabled users to collaborate and access shared resources remotely. Most systems assume that the network communications are fast enough to give the illusion of communicating over local networks. However, these assumptions do not always hold true and this may give rise to unexpected behaviour for the users. This thesis deals with the issues of time and collaboration and looks at how temporal factors affect collaborative work, particularly when it involves interaction over a wide area network.

Consider an example where a number of remote users are collaborating through a chat system. Each user's contribution to a certain topic has to be broadcast to all other users who are interested in that particular chat session. This implies that the contributions have to be sent across the network. If the network suffers from delays, the interested users will not be able to see the contributions within an acceptable time. The flow of conversation can easily get out of synchronisation and users will be confused. On the other hand, if the contributions are sent rapidly over a very fast network, users may find it too distracting to cope with many contributions simultaneously, especially if they have launched several chat sessions on different topics. It is therefore desirable that users see the contributions to each chat session in a timely manner.

Temporal properties have traditionally been linked to the system response time, in other words, the delay between a user's action and the system displaying results back on the screen. In single-user interaction, feedback is the dominant temporal property. Feedback is the rate at which users see the effects of their own actions. But with collaborative interaction (figure 1.1), there is another major temporal property in addition to feedback. Feedthrough is the rate at which users see the effect of other group members' actions.

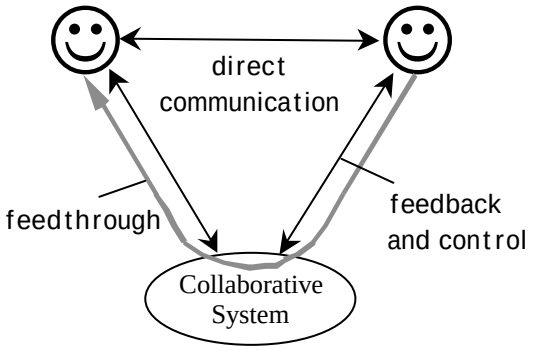


Figure 1.1 Collaborative interaction

Collaborative work introduces delays and lags as users have to wait for both feedback and feedthrough information. Furthermore, interaction over a communication channel increases the likelihood of delays as a result of high network traffic, low bandwidth or remote site failures. Even a high bandwidth connection will affect the system response time during peak

network usage. The unreliability of timely responses increases user frustration and application errors, and can eventually lead to a complete breakdown in the work process.

Collaborative users require both feedback and feedthrough information at a fast enough rate to allow the flow of collaboration to take place successfully. However, the provision of feedthrough is more problematic in a distributed environment as the application may execute on a completely different server from the local user interface through which each user is interacting with. Components placed at different locations face higher communication costs and delays than those at the same location.

There is this fallacy when building infrastructures that we can ignore the implementation details so long as the platform has sufficient capability. The infrastructure is very often treated as a 'black box'. However, the way that things are implemented does actually matter, as the underlying infrastructure tends to show up in non-functional aspects, particularly in timing, for instance during network delays.

Both temporal factors and implementation infrastructures are therefore very important issues in Computer Supported Collaborative Work (CSCW). This thesis will partly consider the issues surrounding temporal properties and collaboration but the principal focus is on the underlying infrastructure that enables the construction of temporally coherent collaborative applications.

This work does not attempt to overcome the problems of communication delays by constructing an effective network protocol. It accepts that delays are likely to occur even over fast networks within a collaborative context and it uses this fact to drive the development of an underlying infrastructure that provides remote users with satisfactory temporal behaviour at the interface.

1.1 Background to the problem

Although the temporal properties of interaction are theoretically essential, they have been poorly investigated with the exception of a few studies (Dix, 1987), (Dix, 1992a), (Dix, 1994a), (Gray et al., 1994). There is also a sociological tradition of studying temporal phenomena that has recently been used in some ethnographic studies (Hudson et al., 2002), (Reddy and Dourish, 2002). Temporal properties in system design have traditionally been associated with the system response time (Miller, 1968), (Card et al., 1991), (Nielsen, 1993). However, the response time is not the only temporal property of interactive systems. This research uses two additional properties as its foundation for assessing temporal problems that users perceive at the interface.

The first lies in the interface behaviour. The user interface can be expressed though Status-Event analysis (Dix, 1991), (Abowd and Dix, 1994), (Dix and Abowd, 1996a) in terms of events and status behaviour. Status-Event analysis has been developed for tackling various user interface issues. Temporal problems at the interface are said to occur whenever any constraints between the status of the interface is broken. The idea of mediation between status is key to the understanding of delays in this research.

The second temporal property lies in the issues surrounding pace of interaction. The pace of interaction is the rate at which users interact with computer systems, the physical world and one another (Dix, 1992a), (Dix, 1994a), (Dix, 1995a). The pace of interaction is influenced by three factors: the pace of the communication channels, the pace of the shared task and the pace at which users operate. A mismatch between either one of these factors and the resulting pace of interaction generates delays. Unlike bandwidth, which gives a measure of the amount of information that is transmitted, pace indicates the frequency of communication.

Thinking about pace makes one concentrate on the timescale over which interaction occurs. This may take place over different lengths of time. When users interact with computer systems they normally expect the delay between their actions and the system feedback to be rather short. For example, in direct manipulation interfaces, the feedback needs to be in the order of 100ms. However, people across organisations often have to interact over a much longer timescale, which may range from hours to days. As a result, the interaction is likely to suffer from frequent interruptions, thus intensifying the temporal problems that people may have to face.

The emergence of the Web as an interactive environment has increased the significance of the temporal properties of interaction. The Web provides an information infrastructure that is universally accepted and this influences our everyday interaction through the Internet. The focus of this interaction is mainly on the communication infrastructure rather than the devices that access it; hence we tend to make inherent assumptions about the architecture of the infrastructure.

Over the last few years, various techniques have been developed for the analysis of CSCW and groupware (Benford and Fahlén, 1993), (Dix, 1994b), (Dix, 1994a). Also, there are many existing architectures for single-user interfaces and multi-user collaborative interfaces (Pfaff and Hagen, 1985), (Bentley et al., 1994), (Hill et al., 1994). However, most of these architectures are based on assumptions that will be broken once the software is no longer running on a single machine or even on a local network. However, these existing architectures form an essential starting point for this work.

Collaborative users require two important temporal requirements – feedback and feedthrough. The provision of feedthrough is more challenging in a collaborative application that executes over a distributed environment such as the Web. For example, rapid user interface feedback on the Web can be promoted by running code locally as downloaded Java applets; however local data updates may conflict with the needs of feedthrough. Feedthrough is an essential feature of cooperative interfaces but there is often little support for it, from either existing applications or the Web protocol itself.

Feedthrough allows participants to see an up-to-date version of the shared task while preventing inconsistent updates. These concerns have led to some considerable work on algorithms for synchronous editing and for merging versions of asynchronously edited material. Feedthrough also promotes the awareness between group members. This has always remained an informal interest in CSCW, but some formal analysis of 'awareness'

models (Benford et al., 1993), (Rodden, 1996) have emerged largely due to work on Collaborative Virtual Environments (CVE).

The analytic focus of this research is driven by the need for optimising temporal performance. The motivation lies in the timeliness of information, as informed by the temporal requirements of collaborative users and the design considerations of the underlying system architecture.

1.2 Objectives of the work

The primary objective of the work reported in this thesis was to develop the existing analysis of temporal problems and use this analysis to drive the development of software architectures for widely distributed groupware systems. This objective can be broken down into the following sub-goals:

- To develop an architectural framework that enables the construction of collaborative applications that satisfy appropriate temporal properties.

- To demonstrate the feasibility of the conceptual framework by using it as a basis for developing an exemplar that provides collaborative users with a temporal behaviour that meets their pace of interaction.

- To evaluate the effectiveness of the approach embodied by the model.

A major goal of this research was to support the construction of distributed collaborative applications that effectively manage the temporal behaviour at the interface level. The architectural requirements of this research are therefore driven by the desired temporal performance of the user interface. The next section shows how these goals have been addressed.

1.3 Approach of the work

The issues surrounding the temporal properties of interaction are first explored. Interaction covers a wide time scale from short-term discrete activities to long-term human processes in organisations. Although the temporal properties addressed in this thesis are driven by user needs on relatively short periods of interaction by focussing on architecture and interface issues (figure 1.2), the processes involved in collaborative long-term interaction were also investigated. The findings of a case study carried out to investigate the temporal problems that arise during long-term interaction are presented in the Appendix.

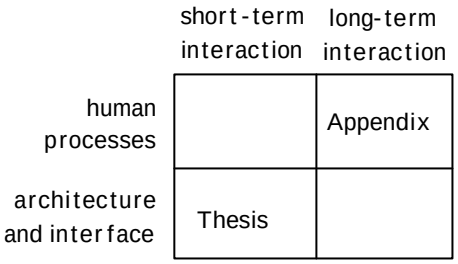


Figure 1.2 Temporal context

Software architecture is about dividing systems into components to perform certain functionalities and then linking the components together in such a way that they can communicate effectively. Although it may at first seem that software architectures are related to the internals of system design and not a necessary concern for the user interface, internal details manifest themselves at the surface. As a result, the consideration of the physical architecture becomes unavoidable. Architectural decisions directly influence the behaviour of the user interface and the most significant behavioural implication on a distributed platform is often the temporal impact.

Interface requirements and architectural models for both single-user applications and multi-user collaborative applications are analysed based on the existing body of literature. This architectural analysis is then extended to the Web, an environment that is predominantly subjected to delays. However, the Web also offers immense potential for the development of distributed collaborative applications, despite the fact that its protocol were originally designed and used largely for accessing anonymous static or slowly changing information.

The Web forces the concern between the location of the data and that of the control, thus generating various alternatives for the placement of architectural components. Location decision is decisive in determining the users rates of feedback and feedthrough. In order to clarify the various architectural options and their effects on the temporal properties offered by the interface, a framework for analysing cooperative architectures on the Web is presented.

Feedthrough is an intrinsic temporal limitation in collaborative applications in general. Furthermore, the needs for feedthrough on the Web conflict with those of feedback. However, feedthrough is crucial for maintaining collaboration and promoting awareness. There are two key requirements for feedthrough – firstly, the ability to access and update shared data, and secondly knowing when that data has been updated. The former lies behind the design of shared data repositories, either bespoke systems designed for CSCW or off-the-shelf databases and shared object stores. The latter requires notification mechanisms, the key element in informing people about status and change.

In order to investigate the different ways in which notification servers can be implemented, a framework for the design space of notification services is presented. The design space also generates a taxonomy for notification servers. The Status-Event analytic framework (Section 1.1) is used as a foundation to explore the ways in which the notification server can become aware of changes to the data and how it in turn, makes this available to the client

applications. The notification server can thus mediate feedthrough information between end-user clients.

Users involved in cooperative work often have to interact with a large number of shared objects. It may not always be possible to provide a fast rate of feedthrough for each object, as there is not enough network and computational resources available. Even if the network was infinitely fast and there was an infinite amount of memory, a maximum rate of feedthrough will generate further network congestion. The extra computational load implies delays for all the objects including the ones that are of higher interests to the users. Furthermore, from a cognitive viewpoint, collaborative users may find it too distracting to cope with a fast rate of updates.

This research therefore proposes that the notification server, through its central mediating role between end-user clients, is ideally placed to provide collaborative users with an effective level of feedthrough by matching the supplied pace of updates with the users required pace of updates. This matching has been called *impedance matching*. Feedthrough demands can be reduced by subsequently reducing the pace of updates (pace impedance) and the volume of updates (volume impedance).

The principles of notification server design and the issues surrounding impedance matching have been employed to develop the Getting-to-Know (GtK) purpose-built separable notification server. GtK is based on a distributed object infrastructure and it is largely an example to show how a controlled pace of feedthrough can be achieved in practice. GtK only supports pace impedance matching given the interests on pace issues in this research.

In order to demonstrate the practicality of the GtK notification server further, an example real-time Web conferencing application has been constructed using the GtK infrastructure. The application offers functionalities that are common in most Web-based chat systems. However, its novel feature lies in its ability to enable collaborative users to interact with multiple conferences simultaneously while adjusting the pace of feedthrough to match their rates of interests.

The main stance of this research lies on an architectural framework that manages the paced delivery of information at the user interface. It is often problematic to evaluate a framework embodied in code. The demonstration via the real-time Web conferencing exemplar acts as a technical evaluation of the GtK framework. Also, an architectural evaluation has been carried out to assess the benefits and limitations of the GtK framework.

1.4 Novel characteristics of the work

The work reported in this thesis represents a novel integration of user interface and architectural issues to provide distributed collaborative users with effective temporal behaviour. Software architecture is very often driven by implementation reasons rather than the desired user-level behaviour. In contrast, this work lays particular emphasis on a user-oriented approach in establishing the requirements of a collaborative architecture.

The method of providing collaborative users with a controlled pace of feedthrough through impedance matching is innovative in CSCW. It emerges from the extensive analytic focus of this research, which also demonstrates that a careful placement of architectural components can facilitate explicit notification mechanisms that match the users' pace of interaction.

This work also generated a novel method of analysing work processes through the 4Rs framework, which emerged from the study into long-term interaction. The 4Rs (Request, Receipt, Response, Release) recurrent pattern of activities is a fundamental unit of long-term work, as the same sequence repeats itself with similar triggers and similar failure modes. The 4Rs framework and the triggers for activities can be applied to assess the reliability of individual parts of a work process during system design.

1.5 Contributions to the research area

There have been a number of publications that this work has generated both in the HCI and CSCW research community. The interests of this research were presented to a panel of experienced researchers and practitioners at a doctoral colloquium session in a past CSCW conference (Ramduny, 1996).

The case study into long-term interaction led to a technical report (Dix et al., 1995), a short conference paper (Dix et al., 1996) and a journal paper (Dix et al., 1998). The trigger analysis technique that emerged from this study has also been proposed as a technique for task decomposition (Dix et al., 2003) that can be applied in conjunction with other task analysis or workflow methods.

The architectural framework for developing Web-based collaborative applications was published as a full conference paper (Ramduny and Dix, 1997a) and a poster (Ramduny and Dix, 1997b). The analytic technique was employed in a later research project[1] to investigate software architecture options for mobile user-interfaces and the findings were published as a journal paper (Dix et al., 2000).

[1] Interfaces and Infrastructure for Mobile Multimedia Applications research project – as part of the EPSRC MNA programme, GR/L64140 & GR/L64157

The structured analysis of the design space of notification servers also led to a full conference paper (Ramduny et al., 1998). This is an important contribution to the CSCW discipline as the last few years have seen the beginnings of a literature of notification servers in their own rights.

Finally, the issues surrounding impedance matching for providing effective user-level feedthrough were discussed at a day conference (Ramduny, 1999) and the findings were recently published as a full conference paper (Ramduny and Dix, 2002).

1.6 Structure of the thesis

Figure 1.3 shows the overall structure of the thesis. A series of analytical studies are presented in Chapters 2 – 7 which provide an understanding of the nature of temporal problems, infrastructure issues and design considerations. These analytical studies lead to the GtK framework that provides pace impedance matching. Chapters 8 – 10 describe the development of the GtK infrastructure, its use through the GtK notification server and evaluation via a real-time Web conferencing exemplar.

Chapter 2 starts by investigating the issues surrounding the temporal properties of interaction. It examines the impact of delays and interruptions on the interactive process and explores the temporal problems that users face on the Web. The foundations of interface behaviour and the issues of pace of interaction are applied to analyse the temporal problems that users perceive at the interface. The results of a case study that was carried out to check the completeness of the existing analysis on the pace of interaction and to identify the temporal problems faced during long-term interaction are discussed in the Appendix.

Chapter 3 considers the interface and architectural concerns involved in designing single-user applications. The analysis is based on examining architectural and temporal requirements for user interfaces, exploring mature architectural models, reviewing some of the tools that assist in the design and development of user interfaces and the various design paradigms employed in architectural and interface development. Similar issues for multi-user collaborative applications are discussed in Chapter 4.

Chapter 5 extends the investigative approach to the Web platform. It provides a systematic analysis that examines the behavioural issues and the architectural components that are necessary for Web-based collaborative applications. The placement options for the architectural components are also considered and the issues surrounding code and data mobility are explored. The findings of the analysis narrows down the behavioural focus of this work.

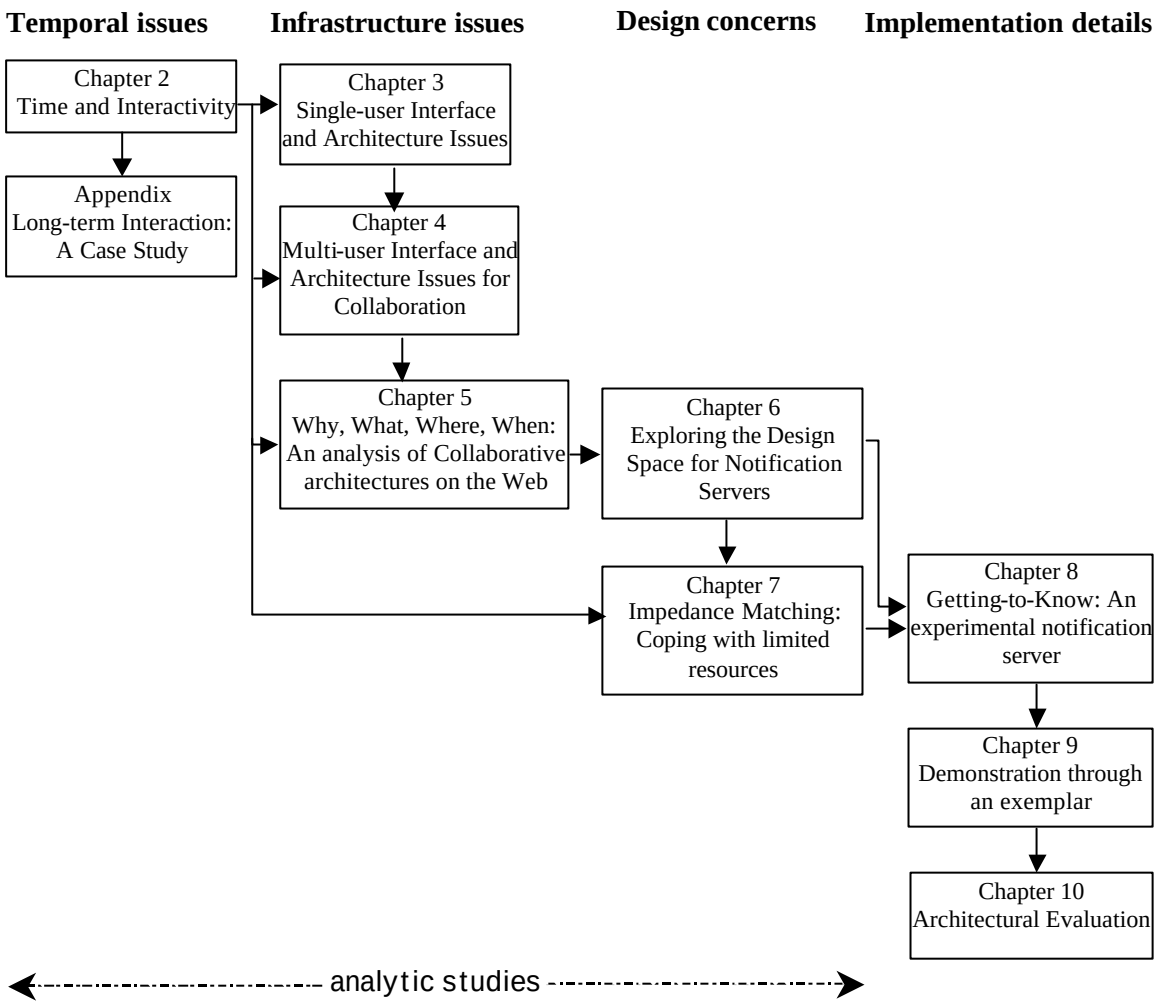


Figure 1.3 Thesis structure

Chapter 6 explores and clarifies the design space for notification servers by adopting an analytical approach similar to that applied in Chapter 4. A taxonomy of the design space for notification servers is also presented. Status–Event analysis (discussed in Chapter 2) is used as the foundation for examining issues of change propagation. The analysis also confirms the important role of the notification server as a mediator of updates between collaborative users in a distributed setting.

Chapter 7 proposes impedance matching as a method for providing collaborative users with an effective user-level behaviour. Due to its mediator role, the notification server is ideally placed for supporting impedance matching by controlling the frequency of notification to match the users pace of interaction. The issues surrounding impedance matching and the related implementation details are thoroughly examined.

Chapter 8 describes the Getting-to-Know (GtK) separable notification server that has been built on a distributed layered architecture to support pace impedance matching. The protocol employed for passing messages between different communication objects within the basic layered infrastructure are examined and the main functions of the GtK notification server are discussed.

Chapter 9 shows how an example real-time Web conferencing application has been constructed on the GtK framework in order to explore the practicality of the GtK notification server as an impedance matcher.

Chapter 10 complements the technical critique in Chapter 9 through an architectural evaluation of the GtK framework. The GtK framework is measured against some important architectural parameters such as the ease of flexibility, possibility for migration, support for dynamic mobility, event management and interaction with existing data.

Finally, Chapter 11 reflects on this work and highlights the broader research themes that this work has already contributed to.

Chapter 2 Time and Interactivity

Time plays an important role in computer systems in general and even more so in establishing the quality of human-computer interaction. Early approaches developed to analyse interaction such as GOMS (Card et al., 1983), TAKD (Diaper, 1989) and production rules (Newell and Simon, 1972) mainly concentrated on identifying operator tasks and examining traces of interaction. These cognitive models largely ignored the temporal properties of interaction because the main focus then was on single-user interaction with stand-alone systems.

Over the last few years, the development of groupware applications like conferencing systems and shared text editors has shifted the focus to multi-user collaborative systems. In addition, there has been a significant growth of distributed information sources using Internet facilities. In particular, the rapid growth of the World Wide Web has promoted a new category of applications to be developed on a distributed platform – Web-based collaborative applications.

Unlike single-user applications, the requirements for cooperative applications are far more complex, both in computational terms and in meeting the needs of the user. This has undoubtedly increased the importance of the temporal properties of interaction. Group users not only need to see the response of their own actions (feedback) within a reasonable time limit, but they must also be able to see the effect of others actions (feedthrough) promptly to allow successful collaboration through the artefact.

The unreliability of timely responses can lead to user frustration and application errors. In the worse case it can cause a complete breakdown in the work process. Furthermore, interaction over a network increases the likelihood of delays between the transmission and reception of users actions, thus affecting the system's response. Remote site failures, high network traffic and slow bandwidth intensify user frustration and error.

Although it is desirable that users perceive system responses almost instantaneously, this is not always possible due to processing time, network delays or delays in the pace of interaction between group users. Clearly, the absence of timely feedback and feedthrough will have a negative impact on group interaction. A thorough understanding of temporal issues will inform system design and development. This chapter investigates the issues surrounding the temporal properties of interaction.

Section 2.1 starts with a brief review of the human memory and looks at the effects of delays and interruptions as informed by cognitive psychology. Section 2.2 analyses the role that time plays on the interactive process. It first considers the issues of response time and then examines the impact of delays during interaction. The need for feedback is also established. A similar analysis is then applied in Section 2.3 to explore the temporal problems that users face on the Web. Finally, Section 2.4 uses the foundations of interface

behaviour and pace of interaction to analyse the temporal problems that users perceive at the interface.

2.1 Background

The cognitive psychology literature provides a good foundation for understanding the temporal properties of interaction from a user-centred perspective. These studies examine how the human memory functions and consider the impact of delays and interruptions on our short-term and long-term memory. With the widespread use of computers, usability issues have become a major concern and usability guidelines have for long been informed by cognitive psychology.

Indeed, there is a very close link between human-computer interaction (HCI) and cognitive psychology (Norman, 1988). Cognitive modelling is used in HCI to get a better understanding of how people interact with computer systems and identify aspects of the system that are easy or difficult to use and/or learn. The areas where people are most likely to make persistent errors can also be anticipated.

This section will first consider the aspects related to the human mind and the effects of interruptions, before looking at some common cognitive models that have been developed to represent the way the user of a computer system thinks.

2.1.1 The Human memory

The human mind is an information-processing system (Card et al., 1983) that can be divided into three interacting sub-systems consisting of memories and processors:

(a) perceptual system – this carries sensations of the physical world detected by the body's sensory system into internal representations of the mind by using integrated sensory systems. Our visual system is a good example of this (Card et al., 1983). Although the eye receives the visual scene over a wide angle, detail is only obtained over a narrow region, the fovea. The rest of the retina provides peripheral vision for orientation and whenever the target is more that 30 degrees away from the fovea, head movements occur to reduce the angular distance. The central vision, the peripheral vision, eye movements and head movements function altogether as an automatic integrated system to provide a persistent representation of the visual scene.

(b) motor system – this translates our thoughts into action by activating patterns of voluntary muscles. The arm-hand-finger system and the head-eye system are the two most important sets of effectors in computers users (Card et al., 1983).

(c) cognitive system – this connects inputs from the perceptual system to the correct outputs of the motor system. The cognitive system has two important memories, a working memory and a long-term memory.

2.1.1.1 Working memory

Working memory or short-term memory as it is also known, only holds information that is under current consideration and the representations produced by the perceptual system. It consists of a subset of activated elements from long-term memory, called *chunks*. Chunks can be related to other chunks and thus be organised in large units.

Short-term memory can be accessed very rapidly but it also decays at a fast rate. The chunks of information can only be held temporarily for a short amount of time, usually in the order of 200 milliseconds (Dix et al., 1993). Furthermore, short-term memory has a limited capacity for retaining information. The classic paper (Miller, 1956) stated that in general, people have the capacity to memorise approximately seven chunks of information at a time and that information can be held in short-term memory for about 15 to 30 seconds.

2.1.1.2 Long-term memory

Long-term memory stores knowledge for future use and unlike short-term memory, it has unlimited capacity for storing information. However, retrieving information from long-term memory usually takes longer, in the range of a tenth of a second and its success usually depends on whether associations between chunks can be found (Card et al., 1983).

People tend to cope with complex problems by chunking them down to simple components. The size of a chunk depends on an individual's knowledge, experience and familiarity with the material (Shneiderman, 1992). Furthermore, there is little decay with long-term memory as recalling information after minutes takes just as long as retrieving it after hours or days.

2.1.1.3 Effect of interruptions

Both long-term memory and short-term memory are highly volatile but the latter is almost instantly affected during interruptions. It is very difficult to recall recently stored information when people are interrupted. If there are long delays, the memory may need to be refreshed. However, people may still resume their work after an interruption if they proceed immediately and record their solution in short-term memory. If instead they record their solution in long-term memory, such as on a piece of paper or on a complex device, the probability for errors increase and the pace of work may slow down considerably (Shneiderman, 1992).

2.1.2 Cognitive models

Cognitive models attempt to represent users' interaction with an interface by taking into account some aspect of the users' understanding, knowledge, intentions or processing (Dix et al., 1993). The cognitive aspect of HCI focuses on the cognitive capacities of users in general and also how these affect the users' ability to carry out specific tasks with computer systems. Very often, this is explained in terms of mental processes expressed in computational terms, as shown by two well-known cognitive models discussed below.

2.1.2.1 GOMS Model

The GOMS model was one of the first cognitive models that described how users perform and coordinate tasks. A GOMS analysis involves describing the task structure and decisions made by users in terms of Goals, Operators, Methods and Selection rules (GOMS) (Card et al., 1983). A goal is something the user wants to achieve. Operators are low-level actions (perceptual, motor or cognitive acts) the user can perform. Methods are the procedures (sequence of operators) required to achieve the goal. Selections are the choices that the user can make between alternative methods of achieving a goal.

A typical GOMS analysis of a particular task involves decomposing an overall goal into sub-goals, each of which can in turn be decomposed into further sub-goals until ultimately these are reduced to basic operators. The actions required to complete the task are arranged into a hierarchical network of goals, sub-goals and operators. GOMS is well suited for analysing routine tasks where the users know all the relevant information about the system they are working with (in other words, expert users) and the tasks can be described into procedures.

The GOMS analysis has been applied to measure performance. The stacking depth of a goal structure can be used to estimate short-term memory requirements. By calculating the time it takes to perform each basic operator and then aggregating the operator times for all operators involved in the task, the total time for completing the tasks can be predicted. Although the GOMS model makes total time predictions, it is not appropriate in situations where errors and interruptions occur (Card et al., 1983).

2.1.2.2 Keystroke-Level Model

Like GOMS, the Keystroke-Level model (Card et al., 1983) only predicts error-free expert behaviour. But unlike GOMS, the Keystroke-Level model needs the method as input (it has no goals or method selection rules) and it only predicts the time to execute a task. User performance is based on key tasks during an interaction, such as the execution of simple commands. The Keystroke-Level model assumes that users first divide complex tasks into subtasks before they are mapped into physical actions.

The Keystroke-Level model decomposes unit tasks into four different physical-motor operators, one mental operator and a system response operator. The total time taken to execute a particular unit task is calculated by adding the time for each keystroke of the various operators. If the user has to wait for a response from the system then an appropriate time is added, which is measured by observing the system; otherwise the system response is assumed to be zero.

Experiments have shown that the Keystroke-Level model can predict performance fairly accurately but the range of applications it covers is limited. Although it is very useful for predicting micro-interactions, it does not do so well in large-scale dialog. Furthermore, the results depend heavily on the approximations made initially. The Keystroke-Level Model is considered to be a very low-level GOMS model, which has been simplified to produce a usable version (Card et al., 1983).

Both GOMS and Keystroke-Level models analyse interaction based on identifying operator tasks and examining traces of interaction. The temporal properties are largely ignored in these cognitive models perhaps because at the time they were developed, the interface requirements mainly involved single-users interacting with software that execute on a single machine. The temporal nature in such a mode of interaction is primarily related to the response time following users actions. But even at the single-user level, delays in receiving feedback have a negative effect on interaction and disrupt our mental processing ability. The next section analyses the impact of time and delays on the interactive process.

2.2 Time and the interactive process

The impact of delays is typified by Norman's interaction cycle (Norman, 1984), (Norman, 1986), (Norman, 1988) (figure 2.1). It describes user activity as consisting of four different stages – *intention*, *selection*, *execution* and *evaluation*. When users interact with a computer system during problem solving, they usually have a goal, they formulate certain actions to further that goal (plan), execute the actions and then evaluate the results of those actions against the expected outcome and the goal.

This model only works if one assumes that the results of the users actions are immediately available. If the delay between executing the actions and observing the results is greater than short-term memory times, the evaluation stage becomes far more difficult. This problem is referred to as the 'broken loop of interaction' and users may be forced to either re-formulate their plans or continue to wait for a response.

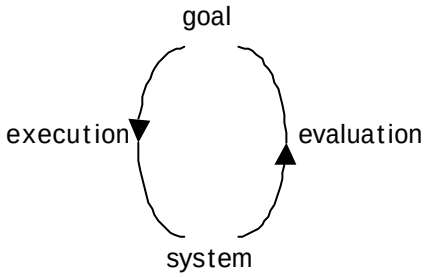


Figure 2.1 Norman's interaction cycle

When delays are predictable or expected, the interaction cycle will not necessarily be affected as users can incorporate known delays into their plans. But when delays are unpredictable, users may forget part of the plan and they may be forced to review the plan continually, thus causing a breakdown in the interactive process. Furthermore, when unexpected delays impede a task progress, many people become frustrated, annoyed and eventually angry (Shneiderman, 1992).

2.2.1 Response time

The primary way in which time is of relevance to interactive systems is through response time. This is defined in terms of the length of time (number of seconds) between a user initiating an action and the computer system displaying the results back on the screen.

Traditional human factors research into response times suggests three time levels that provide different effects on interaction and for almost thirty years, the same norm has been observed (Miller, 1968), (Card et al., 1991), (Nielsen, 1993):

0.1 second – this is the required limit that allows users to feel that the system is reacting instantaneously.

1.0 second – this limit is necessary to avoid interruptions in the users flow of thoughts, although they will notice the delay. No special feedback is usually required for delays in the range of 0.1 – 1.0 second, but users may lose the feeling of operating directly on the data.

10 seconds – this limit is essential for keeping users attention focussed on the dialogue. If the delay is longer, users will perform other tasks while waiting for the results of their actions. Therefore, they should be given some form of feedback to indicate when the computer expects to complete the task.

The above limits do not specify how the interaction process is affected if the delay in receiving a response lies between 1.0 – 10 seconds, a range where many responses actually fall. Also, they do not deal with the factors that can alter the way in which interaction is affected by response times. For instance, some studies have shown that novices prefer and are more productive with a slower response time (Shneiderman, 1992) while others have shown that novices, like expert users, tend to make more errors with longer response times (Long, 1976), (Kuhmann et al., 1987).

Furthermore, user expectations may vary depending on their previous experience and the task at hand. Users can also change their interaction mode with the artefact as they become more familiar with it or as their perception and skills change (Thomas, 1998) but more importantly, users may change their work strategies to adapt to different response times. It is therefore very difficult to assess whether the timing of an interactive activity is too fast or too slow as there are too many user, task and environment variables at play to determine any generally acceptable rate of interface response (Shneiderman, 1992).

Although shorter response times are more beneficial to users, it is possible for the computer to react so fast that the user cannot keep up with. For example, a scrolling list may move so fast that the user cannot stop it in time for the required element to stay within the available window. The fact that computer can be too fast suggests that user-interface changes should be timed according to the real-time clock and not as an indirect effect of the computer's execution speed (Nielsen, 1993). A fast interaction sequence can inhibit users from formulating a solution plan correctly (Shneiderman, 1992). Conversely, users can pick up the pace of a rapid interaction sequence and consequently they may learn less, read with lower comprehension and commit more errors (Shneiderman, 1992).

It is generally agreed that response times should be as fast as possible, preferably within 0.1 second, to limit the effect of delays and maintain users confidence and satisfaction with the system. But in practice, this is not always possible. It is however essential to ensure that

the interface stays usable and this can only happen if the response time is fast enough to match the task at hand.

2.2.2 Impact of delays

A number of studies (Dix, 1987), (Johnson and Gray, 1995), (Nielsen, 1993), (Smith and Mosier, 1986), (Shneiderman, 1992) have considered the effects of delays on user performance and on the behaviour of single-user applications. A delay of more than a fraction of a second in mouse-based interfaces has been found to reduce the success of the interaction. With the emergence of timesharing systems in the early 1960's, other studies (Miller, 1968), (Smith, 1983) investigated the effects of system delay on multi-user performance. Delays of the order of 100 milliseconds have been found to be disruptive within some collaborative virtual environment applications (Macedonia et al., 1994).

The increasing popularity of the Web, both as an interaction and a development platform, has widened the number and types of users and tasks. The changeable nature of the users and the tasks makes it even more difficult to have a single model that relates response time with user performance. Most research assumes that user productivity increases with faster system response times. For example, a study showed that user performance is systematically affected by system delays and users tend to choose task strategies that best suit a given system delay (Teal and Rudnicky, 1992). However, the task that the users were given in the study was relatively straightforward data entry. It may therefore be more difficult to characterise the relationship between response time and user strategy when users are engaged in more complex tasks.

It is generally agreed in the research literature that delays in the system response increase users' frustration, as they are left wondering whether the system is still working or not. If the users do eventually receive a response after a long delay, their attention may have wandered and they may forget which action the machine is responding to. The absence of a system response can therefore be very disconcerting to the user and in the worst case, she may suspect a complete system failure, with consequent disruption and/or termination of the interaction sequence.

Although rapid response times should be regarded as an explicit design goal, it is important to ensure that the user interface matches the rate at which the computer operates. For some frequent user actions, such as function keys or menu selections, a few seconds delay may prove to be intolerable. However, users may accept a relatively slow processing time for instance, during a repetitive form filling dialogue (Smith and Mosier, 1986).

Some experts argue that the consistency of the system response time may be more important in preserving user orientation than the absolute value of the delay. They even suggest that designers should delay fast responses deliberately in order to make them more consistent with occasional slow responses, hence allowing users to adapt to slow response times. A few studies (Shneiderman, 1992), (Conn, 1995) have showed that delays might be acceptable if users are accustomed to them.

However, most studies agree that in order to improve the usability of a user interface, it is more effective to make all responses uniformly fast. In cases where this is not possible, some form of feedback should be used (Smith and Mosier, 1986), (Myers, 1989). As a result, a slow response would become predictable to the user even if it were inconsistent with other responses.

2.2.3 Need for feedback

Our innate ability to act and communicate with each other depends heavily on the feedback we get from the environment and from one another (Dix, 1995a). Studies investigating the effect of delays on user performance (Johnson and Gray, 1995), (Nielsen, 1993), (Dix, 1994a), (Teal and Rudnicky, 1992) emphasise the need for feedback during delays. Feedback information is very important for maintaining users' orientation when they interact with the system. It allows users to know where they are, what they have done and whether the task was successful or not. Feedback is vital in situations where the response times are likely to be highly variable, such as over a network.

When a user is faced with some delay, it is usually hard to ascertain the source of that delay. In traditional interface design, the underlying system architecture and computation tend to be hidden from the users. Although in some cases, information abstraction and hiding enable the design of the user interface to be focussed on user-centred requirements, in other situations, it may have a negative effect. For instance, when a user clicks on a hyperlink or a button on a Web browser either to move to a different page or to download a file, the response time for such an action to complete may vary, depending on the size or location of the file. If the user is not informed of the location or file size prior to requesting the information, it is difficult to estimate the likely response time.

Another problem during interface design is that users cannot distinguish between actions that may lead to different system behaviours and hence different response times. For example, when a user clicks on a button on a menu bar, there is no indication of the complexity of the underlying computation or the expected response time. At the same time, the user may be faced with inconsistent response times when clicking on several buttons that look similar. The difference in the functionality and computation may be obvious to an expert user but less so to a novice user. Feedback is therefore required at two levels – the interface level, to register the user's action (e.g. a button press) and the application level, to show the effect of the action (e.g. a window pops up).

The provision of feedback information not only depends on the length of the delay in receiving a response but it also relies on the nature of the task. So, if a transaction usually processes immediately, delays of the order of a few seconds can be disturbing. Consequently, users should be given some intermediate feedback. Similarly, in transactions where the output must be deferred awaiting the results of a computer search and/or calculation, the expected delay should be indicated to the user. If the interaction is over a network, short lag times may not require an immediate interface feedback, however anything above a few hundred milliseconds are considered to be unacceptable.

2.2.4 Types of feedback

Different types of feedback have been recommended in the research literature to help users cope with inconsistent response times and lengthy delays.

2.2.4.1 Alert box

The alert box is one of the most frequent forms of feedback that provides users with the necessary assurance that everything is working well. It is usually an interim message that pops up on the screen to let the user know that processing has been initiated or a signal that appears while the system is processing the input.

2.2.4.2 Progress Indicator

A progress indicator or percent-done indicator is a form of continuous feedback that is recommended when the computer cannot provide a fairly immediate response or for operations that take longer than 10 seconds (Myers, 1985). Besides reassuring users that the system has not crashed and that there is an ongoing activity, progress indicators also inform users approximately how long they may have to wait. This allows users to plan their time more effectively and perhaps perform other tasks during long waits.

When the processing load cannot be estimated in advance, a running progress feedback can be used to reassure the user by displaying the absolute amount of work done. For example, when a system is performing a search on an unknown number of remote databases, a running progress feedback would be in the form of a list of name of each database as it is processed (Nielsen, 1993). However, a progress indicator is unsuitable for operations that execute reasonably fast, between 2 to 10 seconds (Nielsen, 1993) as users will be unable to keep pace with rapidly flashing changes on the screen. A less conspicuous progress feedback can therefore be used, for example by combining a "busy" cursor with a rapidly changing number in a small field in the bottom of the screen to indicate how much work has been done (Nielsen, 1993).

If a running progress indicator is not suitable, a less specific progress indicator can be used, such as a spinning ball, a busy bee flying over the screen, dots printed on a status line or any other mechanism that shows that the system is working but not necessarily what it is doing (Myers, 1985). Non-obtrusive auditory signals like a chime can also be used to notify users when the display output is complete.

Feedback information is vital but it has to be carefully presented to the users to ensure that they are not dealing with seemingly inconsistent interface behaviour. However, some systems still fail to provide the necessary feedback and in such cases, users attempt to deal with delays themselves by adopting coping strategies (Dix, 1992a).

2.2.5 Coping strategies

Multi-tasking is a common coping strategy that users have applied for a long time during consistent real-time delays in the system response. Users perform multiple tasks in parallel to speed up their rate of completion. Web users also tend to run multiple browser sessions in parallel to cope with the boredom of delays.

A few studies (Cypher, 1986), (Miyata and Norman, 1986), (Conn, 1995) have even suggested that in the event of delays, an appropriate alternative task must be proffered by the interface, together with some support for users to resume the suspended task at an "appropriate" time. However, enforcing such a strategy at the interface level can add an extra burden on the task at hand and cause interference between the different tasks. Although people are adaptable, there are situations where such task management techniques would fail, especially during synchronous collaboration as users interact closely and their contributions undoubtedly overlap.

Most single-user systems are designed to perform certain defined tasks in a specified manner. Often, the data processing loads can be anticipated and integrated in user interface design to provide adequately fast response for all transactions.

However, when task performance requires data exchange and/or interaction with other users over a communication channel, then every participant must be provided with status information about each other (Smith and Mosier, 1986). The support for fast response times therefore becomes more problematic. For instance, in the Quake2 arena-like game, instead of waiting to know if there were enemies round the corner, users were found to turn corners and shoot even before receiving the feedback from their own actions (Knight and Munro, 1998).

The next section will now analyse the temporal problems that users may face when they interact over the Web, an environment that is predominantly subjected to delays. Some potential solutions will also be discussed.

2.3 Interaction over the Web

A number of factors contribute to delays over the Web – network problems due to high network traffic, slow bandwidth or remote site failures, low processing power of host and client machines and poorly designed Web site interfaces. Although there are many high-speed computers appearing on the market and the network capacity is increasing daily, the problems of delays still remain. This tends to affect users' interaction in Web applications in general, but even more so in collaborative Web-based applications.

The length of delays experienced on the Web varies and unpredictable delays have a worse effect on interaction than consistent response times (Section 2.2.2). According to a pilot study (Byrne and Picking, 1997), Web users are critical of delays and regard time-related factors to be of importance to Web usability. The Web is therefore an ideal environment to

2 http://www.quake.com/quake2/index.html

study the effect of response time and the role that time plays on the interactive process. Note that, issues directly related to Web site design will not be considered here as the discussion centres on the Web infrastructure.

2.3.1 Problem areas

Traditional interface design tends to hide the underlying system architecture and computations from the users (Section 2.2.3). Web interface design seems to adopt the same policy, thus reducing its usability. For example, when a user visits a Web site, it is impossible to tell how long the browser will take to load the whole page. This obviously depends on the size of the page, any associated graphics or computations that must be carried out on the local machine and the network speed. However, users are not usually provided with such a level of detailed information, which will allow them to make informed decisions.

2.3.1.1 Response time

Like traditional human factors research into response times (Section 2.2.1), research on Web systems has also shown that users need response times of less than 1.0 second when moving from one page to another if they are to navigate freely through an information space (Nielsen, 1995). Web users are not currently getting sub-second response times; hence they get frustrated (Nielsen, 1997). A response time of no more than 10 seconds has been recommended as the limit for keeping users attention focussed while waiting for Web pages to download.

Often, the Web browser affords users an understanding of the progress in the computer activity whilst they are waiting, through a progress indicator. Although this enable users to tolerate delays, browsers should provide useful progress bars that communicate what percentage of the entire download for a page has been completed (Nielsen, 1997).

2.3.1.2 Network latency

On the Web, the latency of the network is the most obvious cause for delays. A faster network connection only tend to increase the Web performance by a small factor. For instance, upgrading from a dialup modem to an ISDN line only doubles the performance (Nielsen, 1997). Even if a high bandwidth connection to the Internet is used at both user-end and server-end, the response time will increase during network bottlenecks, especially for cross-continent connections and for use at peak hours. In extreme cases, the communication may be broken for longer periods if not completely, for instance during mobile work (Dix, 1995a).

2.3.1.3 Collaborative interaction

The nature of collaborative work itself introduces further delays. Group users do not only have to wait for feedback of their own actions but they must also wait for the effects of others actions – feedthrough. User feedback and feedthrough may improve by using a

faster network connection, but if the cooperative task requires many short network interchanges, additional delays will occur due to buffering and processing at remote and local sites.

Irrespective of the source, delays disrupt user interaction in general, and in particular they affect the nature of the work process during collaborative activities over the Web.

2.3.2 Coping Strategies

Users also adopt coping strategies on the Web to deal with delays when they are seeking for information, but the strategies tend to vary depending on the users knowledge.

A study (McManus, 1997) showed that when users were aware of the location of the Web page they were trying to reach and they had some knowledge of the hardware and/or browser being used, they tried to reach the desired information as efficiently as possible by performing some of the following actions:

- multi-threading
- download pages to the local machines for browsing at a later stage
- expand the cache to allow quicker access to pages viewed earlier in the session
- deactivate automatic image loading

However, when users had little knowledge of the information they were looking for, they minimised the time spent locating that information by carrying out some of the following actions:

- use a site or author they trust and follow their links
- avoid sites that contain a large number of graphics or frames
- use search engines
- use personal information feedback or agent
- use FTP

Another interesting observation from the study (McManus, 1997) was the fact that users actions varied depending on the granularity of the interaction. For instance, when users' interaction were over a long time scale they would adopt measures such as, download pages to the local machine, expand the cache or use personal agents. But in short time scale interaction, users would rather deactivate automatic image loading or avoid sites that contained lots of graphics and frames.

The underlying Web infrastructure does not assist users when they adopt coping strategies. Users could be provided with some help to overcome the problems of delays in some cases. However, it is not always desirable to support all the strategies that users undertake, as this adds an extra burden on the task at hand and may interfere with the different tasks that the users are performing (Section 2.2.5).

2.3.3 Potential solutions

A number of suggestions have been made in the research literature to improve usability on the Web despite its intrinsic delays. Because it is not always possible to control the occurrence of delays, the impact of delays can be reduced by providing users with a greater control over temporal issues, such as Web page loading times. For example, the browser may open some form of dialogue that queries the course of action the user wishes to take, thus creating a sense of rhythm during the interaction (Kutar, 2001).

Other research have suggested that users should be able to decide beforehand whether the value of the information they are trying to download outweighs the cost in retrieving it. For example, small chunks of 'meta-data' about a link can be downloaded and the size and type of information at the link location can be provided to the users via a pop-up right button menu (Bentley, 1997). Similarly, the browser could render images as thumbnails to allow users to evaluate the cost and benefit of viewing those images.

A slow but consistent interface with a regular response time may be preferable to an inconsistent interface with occasional fast responses (Section 2.2.2). However, the notion of slowing down our ever fast computing power is seen as an outrage in a world where speed is increasingly more important (Gleick, 2000).

2.4 Temporal properties of interactive systems

The speed of the response time is not the sole temporal factor of interactive systems. This section applies the foundations of interface behaviour and pace of interaction to analyse the temporal problems that users perceive at the interface.

2.4.1 Interface behaviour

An understanding of the user interface is very important. Temporal problems at the interface are said to occur whenever any constraints between the status of the interface are broken (Abowd and Dix, 1994), (Dix and Abowd, 1996b). This is most likely to occur when fast computation and communication is not available, thus creating a lag between the source of a change and its display, which may eventually lead to parts of the interface to become inconsistent. The user interface can be specified in terms of events and status behaviour (Dix, 1991), (Dix, 1992b), (Dix et al., 1993).

2.4.1.1 Events, status and agents

Events are things in the interface, such as a keystroke or a mouse movement, that occur at a particular time. Events carry a time-scale that is inherited from the task that prompted them. Status, on the other hand, always has a value associated with some interface object, for example, the screen contents or the mouse position (Dix and Abowd, 1996b). Agents are responsible for communicating events to other agents and they do so by changing the value of the status.

Potential temporal problems can be exposed by investigating the status and event occurrences in the interface elements and analysing the way in which agents mediate events through the status.

2.4.1.2 Mediating status

Consider an email delivery system. The file system (status) acts as the mediator between the agent that actually receives the email (e.g. sendmail) and the user's email agent or client (e.g. Microsoft Outlook). Similarly, the email agent highlights an icon (status) to notify the user (another agent) of the arrival of a new e-mail. So in an email system, there are a few occurrences where an agent informs another agent of an event by changing a mediating status (Dix et al., 1993), (Dix, 1992b).

Delays frequently occur when agents are affected by events. If an event involves some form of communication between agents such as sending a message, then it is likely to be a time consuming activity, which may probably be unreliable (Dix, 1991). Furthermore, a lag between the status changing its value (the status change event) and the change being noticed (the perceived event) also causes delays. In order to achieve an acceptable temporal behaviour, it has been recommended that the lag between the actual events and the perceived event should be short relative to the pace of the task at hand (Dix, 1992a).

The idea of mediation between status is key to the understanding of delays at the interface. Its primary contribution to this research lies in the analysis of notification mechanisms in Chapter 6.

2.4.2 Pace of Interaction

The pace of interaction is defined as the rate at which users transmit information when they perform an action and receive a feedback through a communication channel (Dix, 1992a). Pace is a useful measure of the rate of interaction and it is different from response time. The issue of pace is based on the notion that during interaction, the channels of communication between the user and the computer are more often used intermittently and not at a constant rate. Pace is therefore a better measure of communication than bandwidth, which assumes continuous transmission.

The notion of pace can be both measured and quantified. It can be used to provide an understanding of how individuals interact with some data and also how collaborative users interact with computer systems, the physical world and other group members (Dix, 1994a).

The pace of interaction (Dix, 1992a) is influenced by three main factors (figure 2.2):

(a) the intrinsic pace of the communication channel(s)

(b) the pace of the tasks (collaborative) and

(c) the pace at which user(s) operate(s)

Each of these factors will now be considered in turn.

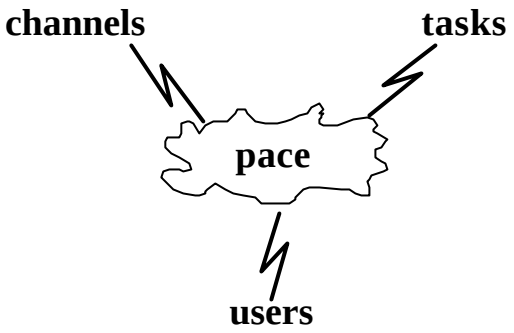


Figure 2.2 Factors influencing pace of interaction

2.4.2.1 Pace of communication channel

When users interact with each other or with the computer, information is very often transmitted through the communication channel(s) in chunks, with some periods of inactivity in between. Consequently, the intrinsic pace of the communication channel is more appropriate to measure the rate at which chunks of information are transmitted instead of the bandwidth.

Bandwidth measures the amount of information that is transmitted whereas pace measures the frequency of communication through the channel. For example, when analysing the rate at which two individuals exchange emails, the bandwidth will only give a measure of the average number of messages transmitted over a given period, but the pace would indicate the rate at which individual messages are produced. In addition to pace, issues such as the time taken to compose a message, the lag between transmission and reception and the time taken to perceive a message should also be taken into account. However, these properties are in some ways inherent in the pace of the channel.

2.4.2.2 Pace of task

Each task has an associated pace which should match the pace of interaction. For example, during collaborative interaction, a large proportion of cooperative work takes place through the artefact (Dix, 1992a) and different users have control of different (or shared) aspects of the artefact. Collaborative users therefore expect to receive both feedback from their own actions and feedthrough of others actions through the changes in the shared artefact (Dix, 1994b). The pace of the collaborative interaction is thus closely linked to the task at hand.

2.4.2.3 Pace of users

There is no clearly defined limit on the pace of activities that individuals can perform. Our mental and physical capabilities are more suited to a certain pace of activity than others. Also, our 'natural' pace varies for different kinds of tasks. For instance, hand-eye coordination tasks have a pace of around 100 ms and this puts a limit on our reaction time. So, computer feedback for hand-eye coordination tasks must be within this timescale.

Furthermore, short-term memory fades over a similar timescale unless refreshed by constant rehearsal.

2.4.2.4 Delays

Delays occur whenever there is a mismatch between any of the above factors affecting pace and the resulting pace of interaction (Dix, 1994a). The most obvious cause of delay is when the communication channel is too slow due to network latency. For example, if a large amount of data is transmitted, the bandwidth determines the pace of interaction, but if the task requires a high number of small network exchanges then, even a low latency network can appear to be slow. Similar problems arise when the intrinsic pace of the channel is too fast for the task at hand.

The context of the interaction and the nature of the task in particular can determine whether a channel is too fast or too slow (Dix et al., 1998). It is sometimes possible to change the nature of the task by speeding it up or slowing it down to fit the channel. However, the pace of the task is typically less flexible, especially in collaborative tasks, due to physical and computational constraints that limit the maximum or minimum pace. The pace at which communication takes place through the shared artefact will therefore affect the task being carried out and delays will arise whenever there is a mismatch between the pace of the communication channel and the pace of the cooperative task.

Collaborative users are more likely to be faced with a mismatch in feedback. This occurs when the pace of each individual interaction with the artefact is greater than the feedthrough of other users' actions. So, if the users do not notice others feedthrough or even when their own feedback is required, their interaction with one another will slow down considerably. The nature of the interface can however determine the effective pace for collaborative interaction.

2.4.2.5 Coping with delays

A way round the problem of mismatch is to adopt some technical or social solutions (Dix, 1994a). For example, if the pace of the task is greater than that allowed by the communication channel, a potential technical fix is to keep local copies of the shared information in order to minimise network transactions and allow users to perform tasks by using the locally available data. However, if the pace of interaction is greater than that of the task, then the task has to be restructured via some social protocol.

Some studies (Dix, 1992a), (Miyata and Norman, 1986) have also shown that cooperative users, like single users, may adopt coping strategies when the pace of the channel is too fast or too slow. For instance, cooperative users may change the nature of the task by increasing the amount of information sent through each chunk of communication if the pace of the channel is too slow. Also, they may delegate certain aspects of the tasks to other users and establish roles. Coping strategies therefore enable users' actions to become more predictable to one another, thus reducing the need for continuous feedback.

Users can also be made explicitly aware of delays in order to encourage the adoption of natural coping strategies. A recent work (Vaghi, 2002) in Collaborative Virtual Environment (CVE) investigated some techniques for dealing with a poor rate of feedthrough in an example multi-user VR pong game, where delays in the ball's movement made the game unplayable. A proposed solution showed ghostly versions of where the system predicted the ball would be, assuming no users hit the ball. This was found to be an effective mechanism in supporting players in their future moves.

2.4.2.6 Time granularity

In order to understand the impact of temporal issues on the pace of interaction, it is important to consider the time scale over which interaction takes place.

Fine-grained levels of time granularity occur in direct manipulation interfaces where the pace of interaction is of the order of 100ms. Coarse-grained levels of time granularity involve a pace of interaction of the order of minutes or hours, such as in batch processing. Another example is the rate of messages turnaround by an email system over a period of hours, days or weeks.

Both fine-grained and coarse-grained levels of granularity make demands on our memory. A study of the use of electronic paper diaries (Payne, 1993) showed that a varied time granularity is relevant both in terms of the user's interaction with the system and the system's functionality. The way in which electronic diaries uses time granularity has a strong impact on usability.

At fine-grained levels of time granularity, issues such as status-status mapping are encountered at the interface (Section 2.4.1). Coarser levels of time granularity are a potential source of difficulty as variations in users actions are more likely to be seen (Thomas, 1998). Users tend to change their course of actions in the long term. This is usually motivated by either a mandatory action, which require users to perform a completely new action, or by the user perceiving some benefit, such as time saving.

Different time granularities have different effects on the interactive process. Most work in human-computer interaction (HCI) focuses on tasks that take a few seconds or minutes to accomplish and where individual actions receive almost instantaneous feedback. However, a significant proportion of collaborative activities occur over weeks or months and users normally have to wait for hours or days before getting some form of response to their actions. The slow pace of interaction implies that the tight cycle of action and feedback is broken.

Although this work is driven by users' needs during relatively short periods of interaction, an analysis of temporal problems during long-term human processes was also carried out by applying the issues surrounding pace of interaction. The results of this empirical case study are described in the Appendix. The study was undertaken to expose the problems of delays and interruptions during long-term collaborative interaction, thus uncovering potential design irregularities.

2.5 Summary

This chapter has investigated the importance of time during interaction. The brief review on cognitive psychology provided an insight into the human memory and looked at the effects of delays and interruptions on our short-term and long-term memory. Delays and interruptions constitute inappropriate timing and may cause a failure in the users immediate expectation, thus giving rise to usability problems.

Traditional human factors research identifies three response time limits that affect interaction. These limits however, do not consider the factors that can alter the way in which interaction is affected by response time. Consequently, issues such as the users' level of experience and familiarity with the system or even the speed of the interaction sequence are not taken into account. Whilst it is generally agreed that the response time should be as fast as possible, it is even more important to maintain a consistent interface by matching the response time with the task at hand.

The impact of delays on the interactive process and the resulting influence of usability were also discussed. Untimely responses can render the task at hand more complex and may eventually cause a breakdown in the work process. So, when responses cannot be made uniformly fast, it is vital to provide users with some form of feedback to assist them in their tasks. Often in situations where feedback is inexistent or badly implemented, users tend to adopt their own strategies to cope with the frustrations of delays.

The Web was then used as an example to analyse the problems that users may face during interaction. The Web environment is largely subjected to delays due to network latency. However, with Web-based collaborative applications, there are additional delays that are introduced by the nature of collaborative work. Collaborative users require both feedback of their own actions and feedthrough of the effects of others actions. Delays therefore intensify user frustration and errors and can easily disrupt group interaction.

The foundations of interface behaviour and pace of interaction were applied to analyse the temporal problems that users perceive at the interface. The interface behaviour deals with issues such as status, events and agents. Temporal problems at the interface is said to occur whenever any constraints between the status of the interface are broken. Pace is a measure of the rate of interaction and it includes factors like the pace of the communication channel, the pace of the task and the pace at which users operate. A mismatch between any of these factors and the resulting pace of interaction will inevitably cause delays. Both the interface behaviour and the pace of interaction play an important role in the understanding of temporal problems in this research.

The most obvious cause of delay during collaborative work arises when there is a mismatch between the pace of the communication channel and the pace of the cooperative task. Furthermore, collaborative users will more likely face a mismatch in feedback. This happens when the pace of each individual interaction with the artefact is greater than the feedthrough of others actions. Users' interaction will slow down significantly if they do not

notice each other's feedthrough or even when their own feedback is required. The nature of the interface can however determine the effective pace for cooperative interaction.

The pace of interaction spans over different time granularities, from hundreds of milliseconds in direct manipulation interfaces, to minutes or hours in office-based environments. Long-term interaction poses different problems to high paced interaction. Although this thesis focuses on technological and architectural issues dealing with short-term interaction, a broader analysis of temporal systems would be incomplete without addressing the concerns of long-term interaction. An empirical study was thus carried out to analyse how users' interaction may be affected over long periods of time. The findings of this study are presented in the Appendix.

Chapter 3 Single-user Interface and Architecture Issues

This research aims to develop an architectural framework that satisfies appropriate temporal properties for distributed systems, particularly Web-based collaborative systems. The temporal issues that arise during interaction were investigated in Chapter 2. We will now consider the interface and architectural concerns that are involved in designing a software application, starting with single-user interfaces in this chapter and followed by multi-user interfaces in Chapter 3.

An important issue in the design of a software application is its overall architecture, which includes the nature of the components of the application and the way the components communicate with each other. The user interface is one of the major architectural components as it is responsible for managing the interaction between the user and the application, by handling input from the user and sending output to the display.

The user interface deals with the hardware and the software that enable users to interact with the computer. It embodies elements that are related to both the user and the system and the methods of communicating information between them. The task of implementing a user interface is complex and code intensive. The user interface has to control a number of devices and their input streams, in addition to performing interaction tasks such as displaying data, parsing input and reporting errors (Myers, 1989).

A fundamental problem in designing a user interface lies with understanding the users and the tasks that need to be supported. Some studies have shown that users are extremely diverse and the wide range of functionalities that have to be satisfied tend to make the application inherently complex (Curtis et al., 1988), (Gillian and Breedin, 1990). Several tools have been developed over the years to speed up the process of building user interfaces and simplify the task of creating and maintaining interfaces.

This chapter gives an overview of interface and architectural issues for single-user applications based on the existing body of literature. Section 3.1 starts by looking at the important architectural and temporal requirements that a single-user interface should satisfy. Section 3.2 explores some mature architectural models for single-user interfaces, such as Seeheim, Arch/Slinky, Model-View-Controller and Presentation-Abstraction-Control. These architectural models are analysed further in Section 3.3, in terms of their conceptual and physical structure. Section 3.4 reviews some of the tools that assist in the design and development of user interfaces, including toolkits and interface development environments. Finally, Section 3.5 explores the various design paradigms employed in architectural and interface development.

3.1 Requirements

Although there are a number of requirements for a single-user interface, usability and performance are considered to be most desirable. Usability deals with how well the interface satisfies its functionality and is related to the consistency of the interface. Performance is instead primarily governed by the temporal properties embodied by the interface. The focus of this research is on the performance aspect of the user interface and this has a direct impact on the way that users interact with the application.

This section considers some of the main architectural and temporal requirements that an interface should satisfy, including issues such as separation, direct manipulation, rapid semantic feedback and consistency.

3.1.1 Separation

Separation is an important architectural requirement that involves separating the abstractions of the application from those of the interface (Edmonds, 1992). The abstract objects can communicate with each other but each object should not depend on the specific implementation details of the other. There are several advantages that can be gained by separating the application semantics from the user interface (presentation) components.

Firstly, separation promotes customisation – certain parts of the application may be changed without redesigning the whole interface. Secondly, it facilitates re-use – the interface can be altered without changing the underlying application code, thus allowing parts of the interface to be used for other applications with just some minor modifications. Finally, separation enhances portability – an application can easily run on multiple platforms with different interfaces.

Many current design paradigms such as object-oriented, distributed and model-based systems promote separable user-interfaces. However, separation is hard to achieve especially in direct manipulation interfaces where the interface is often tightly bound to the application. Moreover, separation conflicts with the needs of rapid semantic feedback, an essential temporal property. Once the application semantics and the presentation components are separated, they have to be linked in such a way that they can communicate effectively with each other (Edmonds, 1992).

3.1.2 Direct manipulation

Direct manipulation interfaces have increased in popularity over the years due to the naturalness of the physical metaphors employed. Direct manipulation interfaces allow users to interact directly with the objects displayed on the screen and manipulate them by using devices such as the mouse or the keyboard, thus enabling rapid, reversible incremental actions (Schneiderman, 1983). However, direct manipulation interfaces are more difficult to implement as they frequently involve elaborate graphics, many alternatives for a single command, several input devices and a mode-free interface which allow users to enter any command at virtually any time (Myers, 1989).

3.1.3 Feedback

From the user's point of view, the performance of the interface is very influential during interaction. An interface is said to exhibit acceptable temporal properties if it ensures that there is no perceived lag between the user's actions and the system's response. In other words, the response time between the user interface and the application should be almost instantaneous. Feedback is basically the response a user receives from the display after performing an action; for instance, a button is highlighted when the user clicks on the mouse. Feedback may depend on the semantics of the underlying application.

Direct manipulation interfaces promote rapid semantic feedback by providing almost instantaneous response to users actions. But these interfaces also require information to be exchanged extensively between the user interface and the application to provide semantic feedback. Such a level of communication does not favour dialog independence and consequently hinders the run-time separation between the application and the interface components.

3.1.4 Consistency

Consistency is another important architectural property that increases the predictability of an interface (Gram and Cockton, 1996). A consistent interface allows users to transfer knowledge from one context and apply it in new situations. Users can therefore anticipate what the system will do, thus encouraging the development of behaviour patterns. Consistency also increases the ease of use of a system and speeds up the user's learning process.

It is easier to produce a consistent interface by splitting a system into various components and letting each component handle a particular functionality. The components can then interact with each other by sharing their behaviour. If a user interface is instead bound to its architecture, some degree of interaction may still be possible between the components, but consistency is not necessarily guaranteed.

The discussion in this section has highlighted the pros and cons of each requirement and shown how the needs for some requirements conflict with others. Although all the above requirements are desirable, they are influenced by the architectural solution that is adopted.

3.2 Architectural models

Most of the architectural models for single-user applications support the partitioning of the application semantics and the user interface functionality. This section looks at the main architectural models that have been used over the years for building single-user applications.

3.2.1 Seeheim model

The earliest significant work which supported the separation between the application and the presentation was Newman's Reaction Handler (Newman, 1968). However, the first explicit architecture that was developed was the Seeheim model (Pfaff and Hagen, 1985).

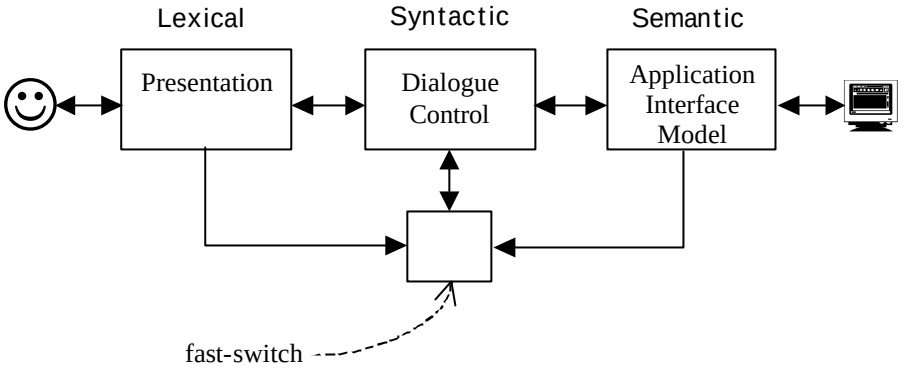


Figure 3.1 Logical components of Seeheim model

Figure 3.1 shows the logical components of the Seeheim model (Pfaff and Hagen, 1985). The functionalities of each component are:

- Presentation – this component is responsible for the external appearance of the user interface and accepts users input and generates output on the display screen.
- Application Interface Model – holds the data and defines the semantics of the application. It also provides a view of the application semantics at the interface by mapping a subset of the application entities onto the user interface code.
- Dialogue Control – this component is mainly responsible for mediating the interaction between the user and the application to provide semantic feedback. It manages the input sequence from the presentation component and the output sequence from the application interface model.

The Seeheim model has often been criticised for the linear nature of communication between the components. This is seen as a bottleneck for direct manipulation user interfaces where rapid semantic feedback is required. But some argue that this statement is only true if the architecture is implemented naively or if very fine-grained communications are required (Sawyer and Mariani, 1995).

However, the fast-switch represented by the lower box in figure 3.1 allows the application to bypass the dialogue component when its state is not affected by output events. The application can therefore communicate directly with the presentation component to provide rapid feedback. But unlike the other functional components, the fast-switch is less well defined and correspondingly more difficult to implement as an architectural feature.

3.2.2 Arch/Slinky model

The Arch/Slinky model (Gram and Cockton, 1996) recognises the fluidity of boundaries between the user interface and the application functionality, which constitute its two endpoints. In between these endpoints there are three additional component layers as illustrated in figure 3.2.

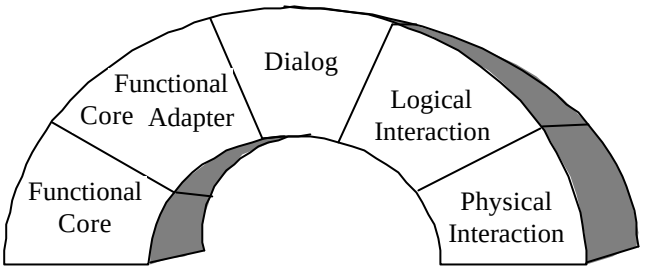


Figure 3.2 Arch/Slinky model

The interaction between the components and their functionalities are explained below.

- Functional Core – this component controls, manipulates and retrieves the application data. It uses the application data and operations to provide functionalities that are not directly associated with the user interface.
- Functional Core Adapter – acts as a mediator between the functional core and the dialog component. It augments the functionality of the functional core component to provide a service that is associated with the user interface and is thus related to the presentation of the information.
- Dialog – this component controls task sequencing between the user and the portion of the application domain that depends on the user to ensure consistency.
- Logical Interaction – mediates the interaction between the dialog and the physical interaction component. It controls users interactions without depending on the toolkit objects and includes descriptions of the data to be presented to the user and events to be generated by the user.
- Physical Interaction – implements the interaction with the end-user via hardware and software. It deals with input and output devices and is typically realised as a user-interface toolkit or an interface library.

In order to clarify the functionalities of the above component, let us consider an employee database as an example. The functional core component will be responsible for retrieving a set of employee names and salaries by gender from the database. The functional core adapter will instead allow a list of employee details to be viewed to display parts of the records. The logical interaction component may present the list of employees and salaries in a tabular form. Finally, the physical interaction component can present users with two radio buttons to allow them to select an employee with a particular salary.

3.2.3 Model-View-Controller

In addition to architectures that divide the entire system into a small number of large components, there are many agent-based or object-oriented user interface architectures. However, these either identify individual agents as belonging to one of the traditional layers or include a layering within each agent. One of the earliest object-oriented user interface architectures was the Model-View-Controller (MVC) (Lewis, 1995), which was initially implemented in the Smalltalk (Krasner and Pope, 1988) programming environment.

The MVC architecture separates the application object – the *model*, from the way it is represented to the user – the *view*, and the way the user controls it – the *controller*. The decoupling of these objects gives MVC a greater flexibility and promotes re-use, modularity and encapsulation. MVC also provides a powerful way to organise systems that support multiple presentation of the same information by using different views. Models, Views and Controllers form triads of cooperating objects that are fully aware of each other's existence. Each view-controller pair is associated with only one model but a particular model can have many view-controller pairs.

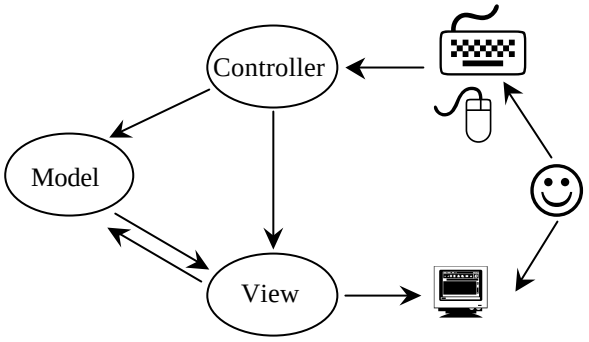


Figure 3.3 Model-View-Controller model

The link between the objects is built up in units by means of the MVC triad (figure 3.3). The functionalities of each object are described below.

- Model – implements the state and the behaviour of the application domain. It holds the data that is relevant to the application and acts upon it in ways defined only by the application thus enabling different user interfaces to use the same model functionality. The model is not aware of how the data is to be displayed or even what actions are used to manipulate it.
- View – requests data held in the model objects and presents the information to the user in a graphical and/or a textual mode. The different view objects have no bearing on the intrinsic behaviour of the model.
- Controller – provides an interface between the model, its associated view objects and the interactive user interface devices. It handles users input by tracking input devices movements and sends messages to the model.

A simple example of a *model* could be a clock object, whose intrinsic behaviour is to keep track of the time by updating an internal record of the time after each second. One *view* object can therefore display the time as an analogue clock while another can show it as a digital clock. When the clock is reset directly by typing the current time into the digital clock display, the *controller* object associated with the view will know that a new time has been entered and it will call the relevant method of the model object.

In order to produce an output, the view has to ask the model for the appropriate data and it can only do so if it has prior knowledge of the object whose data it has to display. This information is gathered through the view-controller link. If a controller is linked to several view objects and the user updates some data on a particular view, the controller must know exactly which view will be affected to perform the necessary changes on the model. However, in some cases, the controller may interact directly with the view without going through the model. For example, the view may consist of a list of information and the controller can make a request for the data to be displayed in alphabetical order. For such a simple re-formatting operation, the data need not be updated on the model.

It is sometimes impossible to partition the functionalities between models, views and controllers. Consequently, 'view-like' functionality might leak into models, for instance, where the model has to know about the screen layout. Similarly 'model-like' functionality could leak into controllers, for example, where it is more convenient to deal with mouse clicks.

3.2.4 Presentation-Abstraction-Control

The Presentation-Abstraction-Control (PAC) model (Coutaz, 1987) is an agent-based user interface architecture that is partially built on the Seeheim model. The user interface in PAC is structured on agents or interactive objects at the top level and each interactive object is decomposed into semantic, syntactic and pragmatic chunks (figure 3.4).

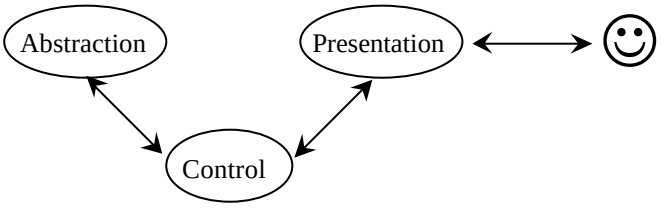


Figure 3.4 Presentation-Abstraction-Control model

The functionalities of each component in PAC are described below.

- Presentation – defines the syntax of the application, in other words the input and the output behaviour of the application as perceived by the user.
- Abstraction – represents the application semantics and implements the applications' functionalities.
- Control – maintains the dialogue and the consistency between the abstraction and the presentation components. It manages the overall interaction between the user and the application.

The PAC interface is constructed as a hierarchy of interactive objects or agents. The link to the application is made via recursive calls from a PAC object to another at each level of abstraction of the user interface, where each presentation component of an interactive

object maps onto another interactive object. The whole interactive application can be treated as being a PAC entity.

The notion of interactive objects makes PAC conducive to an object-oriented approach and thus offers several advantages. For instance, an interactive object can be customised without changing its presentation or its related abstract interface(s). Similarly, the interfaces can be altered independently without causing any side effects. The interactive objects may be regarded as active entities that communicate with each other through some form of inheritance mechanism which enables concurrent multiple input and output.

3.3 Analysing architectural models

An architectural model can be viewed in two ways – either as a conceptual (or logical) architecture or as a physical architecture. This section will discuss the similarities and differences between the Seeheim, Arch/Slinky, MVC and PAC models from both conceptual and physical perspectives.

3.3.1 Conceptual architecture

A conceptual or logical model prompts us to think about user interface development issues in general. Table 3.1 summarises the main functionalities offered by the different architectural models for building single-user applications.

The Seeheim model is often criticised for relying on the user interface functionality to be decoupled from the application functionality. Like Seeheim, Arch/Slinky considers the system as a whole and partitions it into distinct application data and user interface components. However, the Arch/Slinky model recognises the fluidity of boundaries by maintaining the central role of layering and separability (in fact adding additional layers), but it also accepts that the precise placement of these layers into coded modules may vary between systems and even between parts of the same system.

Functionalities	*Seeheim*	*Arch/Slinky*	*MVC*	*PAC*
separate user interface and application functionality	✓	✓	✓	✓
layer presentation/semantics	✓	✓	✓	✓
share application and interface data		✓		
view system as whole	✓	✓		
view system as multiple interactive components			✓	✓
share same view on structure within framework	✓			✓
enable independent representations			✓	
build large interactive systems from smaller components			✓	✓

Table 3.1 Summary of functionalities offered by architectural models

The Arch/Slinky model can therefore demonstrate that a separable user interface is not at all ignorant of the functions of the system. It addresses the problem of semantic feedback by sharing application data with the interface while still providing the advantages of modularisation (Szekely, 1987). However, much more research is needed to justify the efficiency of this method (Myers, 1989). Although the suitability of the Seeheim model as a run-time architecture has been questioned, it still serves as a useful conceptual model that provides a decomposition of roles.

The layered presentation/semantics distinctions can also be found in MVC and PAC. The PAC and MVC models are however atomic based, in that they view the system as consisting of multiple interactive components, each having its own intrinsic behaviour. The focus is on individual parts of an interface, for example the interaction with a particular item of data rather than the global separation of the application into components for each level. They identify individual objects as belonging to one of the traditional layers or include a layering within each object.

The MVC model recognises the independence between the representation components and hence bears no similarities with the Seeheim model. Instead, PAC like Seeheim, shares the same view on the structure within the framework. The user interface structure in PAC allows many PAC objects to exist within a single framework, while this is not the case with MVC. However, unlike Seeheim, PAC implements the dialogue in a distributed fashion and is thus a multi-agent model. Likewise, MVC addresses the issue of building large and complex interactive systems from smaller components, but neither Seeheim, nor Arch/Slinky support this feature.

3.3.2 Physical architecture

A physical model pushes us to view the architecture as components of a system with named roles and communicating along specified paths. This is particularly obvious when all the components are placed on the same machine. Table 3.2 summarises the functions of the different physical components of the architectural models.

Functions	*Seeheim*	*Arch/Slinky*	*MVC*	*PAC*
define application semantics	Application Interface Model	Functional Core Adapter	Model	Abstraction
manage control	Dialogue Control	Dialog	Controller	Control
accept input	Presentation	Logical Interaction + Physical Interaction	Controller	Presentation
generate output	Presentation	Logical Interaction + Physical Interaction	View	Presentation
control, manipulate, and retrieve application data		Functional Core		

Table 3.2 Mapping of components between architectural models

There are some obvious similarities between the components in the Arch/Slinky and the Seeheim model. The *application interface model* in Seeheim takes the role of the *functional core adapter* in Arch/Slinky, while the *dialog control* component maps directly onto the *dialog* component. In addition, the *presentation* component in Seeheim encompasses the functionalities of both the *logical interaction* and the *physical interaction* of Arch/Slinky. However, there is no functional equivalent for Arch/Slinky's *functional core* component in the Seeheim model. The *abstraction* component in PAC roughly corresponds to the *application interface model* in Seeheim.

Although both PAC and MVC models are based on an object-oriented paradigm, there are some significant distinctions between them. In MVC for instance, the input and output events are handled separately by the *controller* and *view* components respectively, whereas in PAC, the *presentation* component manages both the input and output. A change in user input does not necessarily imply a change in the output, but this often is the case at finer grained level of interaction. Consequently, PAC will more likely provide immediate feedback after each input event. However, in some MVC based systems, such as the Andrew Toolkit (Palay, 1988), the *view* and the *controller* are combined into a single object to reflect the tight coupling between input and output events in direct-manipulation interfaces, thus reducing the communication overhead between the components.

Another major difference lies with the control. The PAC model has a dedicated *control* component to manage the control, whereas the *controller* component in MVC also handles other tasks, such as users input events. A separate control component makes PAC more amenable to other kinds of events, hence allowing it to be easily applicable within a multi-user context. However, it is worth noting that the MVC model has been applied successfully on a variety of platforms and effectively instantiated as a design standard.

3.4 Interface development tools

In addition to architectural models for building single-user applications, there are a number of tools that are available to assist developers with the design and implementation of user interfaces. The conventional development tools range from windowing system at the lowest level to toolkits, User Interface Management Systems (UIMS) and User Interface Development Environments (UIDE). This section will provide an overview of each of these development tools.

3.4.1 Windowing systems

The windowing system was one of the earliest tools developed that used the concept of a window or a frame on screen for managing user interactions. The application processes user input as events and display output in a window, usually in graphics form. Some common windowing systems are X Window system for Unix (Scheifler and Gettys, 1986), the Apple Macintosh Toolbox (Computer, 1985) and Microsoft Windows (Microsoft, 1993).

X Window is a network-based system that uses a client-server architecture. An X application client can run on any machine in the network and the server resides on the user's workstation. The server is responsible for mediating user input and displaying output. It translates user input into X events and sends it to the relevant client. The client then interprets these events and informs the server on how to update the display layout.

Windowing systems in general provide a very low-level of abstraction for building applications (Myers, 1989). As a result, toolkits and UIDE are built on top of them to assist developers.

3.4.2 Toolkit

A user interface toolkit is a library of widgets or interaction objects, such as menus, scroll bars and buttons that can be called by an application program (Myers, 1989). Interaction objects have a predefined behaviour and their attributes can be tailored to meet the needs of the programmer. When a user activates a widget, callback procedures are executed. A toolkit usually consists of an application programming interface (API) and a run-time library.

Toolkits allow high quality applications to be built on top of them with less complexity and they enforce consistency across the interface by providing similar behaviour to a whole collection of widgets (Dix et al., 1993). Example toolkits include the Andrew Toolkit (Palay, 1988), Motif (Foundation, 1989) and TCL/TK (Ousterhout, 1994). Some toolkits also offer a direct manipulation graphical interface like Garnet (Myers et al., 1990), SUIT (Pausch et al., 1992) and Java toolkit (Microsystems, 1996).

Although toolkits are popular for building interfaces they do have certain limitations. The support for separation only applies to the level of each individual component and not to the whole architecture (Linton, 1993). Also, toolkits do not provide much support for designing interfaces or sequencing specifications and dialogue control. Furthermore, toolkits provide a limited range of interaction objects, thus restricting the scope of user interaction to those widgets supplied by the system. Finally, toolkits are often expensive to create and difficult to use by non-expert programmers, perhaps with the exception of SUIT, which was designed for novice GUI developers.

3.4.3 User Interface Management Systems

User Interface Management Systems (UIMS) add another level of services in interactive system design beyond the toolkit level. The partitioning of the application semantics and the user interface functionality was one of the main motivations behind the design of UIMS. This concern generated another set of issues related to the mode of connecting these two components together and the protocols of communication between them. The Seeheim model was the first explicit architecture of what constituted a UIMS.

A UIMS offers mechanisms for the run-time management of user interfaces (Hix, 1990). An application is typically separated into components and a runtime component is responsible for managing the interaction between them (Olsen Jr, 1992). A UIMS can be

integrated with tools that define user interfaces to provide an environment that allows rapid prototyping and execution of the applications' interactive components (Sawyer and Mariani, 1995).

3.4.4 User Interface Development Environments

User Interface Development Environments (UIDE) provide an integrated set of tools that address the design activities that precede the management of the run-time system (Dix et al., 1993). There are a number UIDE developed for single-user systems which support the process of creating a user interface and they typically include programming environments such as Visual Basic and Visual C++.

Although UIDE plays an important role in user interfaces in general, there has not been much support in this area for multi-user interfaces – the subject under consideration in the next chapter. However, there are some development tools for the Web environment that support the construction of applets, HTML web pages and CGI scripts such as File maker[3] and Visual Café[4]. The latter also executes servlets but there is no facility for linking the servlets with the CGI scripts. Given the limited coverage in multi-user interfaces, the issues surrounding UIDE will not be considered any further.

Interface development tools should ideally help developers convert interface specifications into an interactive system, while supporting all stages of the life cycle of system development including prototyping, implementation, testing, maintenance and system enhancement (Baecker et al., 1995). While some tools support some of these goals, none address all of them. The majority of the tools aim at increasing developers' productivity. They claim to reduce the development time in the life cycle of user interfaces and increase the quality of the interface by making it easier and more economical to create and maintain (Myers, 1995).

3.5 Design paradigms

From a software engineering viewpoint, it is advantageous to describe an application's functionality as being separate from its presentation. But in practice, this degree of separation is difficult to achieve. This section will now explore some of the ways in which architectural models and interface development tools have been implemented at the conceptual level.

3.5.1 Event-based

An event-based model considers input tokens as events. Input events are processed by event handlers, which modify the internal state of the system and call the relevant application

[3] http://www.filemaker.com/

[4] http://www.visualcafe.com/

routines before generating output events. Such systems are very efficient at managing multiple processes and they do support some semantic feedback.

However, the application only communicates with the presentation when there is a need for input. The dialogue control is therefore internal to the application and this forces the application to be aware of the presentation issues, thus rendering the application less generic. Moreover, the tight degree of control between the input and output events makes the task of modelling the overall flow of the dialogue more difficult, as small changes in one part of the program may affect many other parts (Dix et al., 1993).

3.5.2 Object-oriented

In an object-oriented model, the elements of a user interface are represented as objects. There are usually two kinds of objects: interactive objects, which implement the user interface and abstract objects, which implement the data underlying the interface. This keeps the user interface separate from the application code.

Objects provide a good abstraction mechanism that encapsulates both state information and operations. Moreover, an application can be easily extended by specialising high-level classes through the inheritance mechanism. The X Window System Manager uses an object-oriented programming style (Scheifler and Gettys, 1986). Toolkits that are amenable to an object-oriented approach provide instantiation and inheritance features, thus simplifying customisation.

Object-oriented systems can handle highly interactive direct manipulation interfaces as the application can alter the computational link between the input and the output to provide rapid semantic feedback. However, these systems are usually programming environments and may as a result be unusable to non-expert programmers (Barth, 1986), (Krasner and Pope, 1988).

3.5.3 Constraint-based

In a constraint-based model, the link between the presentation and the application is made more explicit via the use of constraints. Garnet (Myers et al., 1990) is an example single-user application that uses simple constraints for communicating between the user interface and the application. The control component in the PAC model also has an implicit notion of constraints between the values of the application and those of the presentation. In general, most toolkits tend to hardwire the input handling directly into each widget. But some toolkits such as Grow (Barth, 1986) and Garnet (Myers et al., 1990) add constraints on top of their object-oriented functionality to enable designers to specify the relationships among the objects, which are then maintained automatically by the system.

Constraints are enforced at run-time by reflecting the changes in one object onto others. For instance, in Garnet, the look (interface) and feel (application) is separated by encapsulating the behaviour of input devices in interactors (Myers, 1990). Consequently, different widget types can be implemented on top of the same interactor mechanism. This

approach is very similar to Smalltalk's MVC paradigm (Krasner and Pope, 1988), where the model is the prototype-instance object specification, the view is the graphical object system and the controller is the interactor. All three objects are interconnected by a constraint system.

Unlike event-based systems, constraint links produce an independent description of the dialogue controllers. The interface procedures only call the application when the user inputs a command and as a result, the control is external, usually in a separate dialogue component. However, in order to preserve the intended link between the application and the presentation, a great deal of information about each other must be represented in the external component and this may be an inefficient and cumbersome task.

3.5.4 Callback

In systems using callback procedures, the separation between the application and the presentation is maintained by adopting a notification-based programming and similar to constraint-based systems, the dialogue control resides externally from the application. So when a user performs some input action, the notifier or the interface invokes the appropriate application procedure to handle the event.

Toolkits are usually based on a procedural interface and all actions or events in the interface are handled by callback procedures. For instance, the standard X toolkit (OSF, 1995) uses callback procedures to implement the application interface. The callbacks are directional as the application objects have to register callback procedures with the notifier or the user interface objects (widgets). When a specified event occurs, the relevant callback procedure in the application is called. Callbacks tend to increase the dependency between the application and the interface (Myers, 1991).

The MVC architecture uses callbacks in the opposite direction. The model object allows the view and controller objects to register callbacks with it, hence the dialogue is not managed separately. When some aspect of the model is changed, the respective callback methods are notified. It is then up to the view and the controller objects to determine what changes have taken place, so they must have some knowledge of the model internal structure. Modularity is hard to achieve in MVC as any change in a component undoubtedly affects the others.

The use of callbacks to communicate between modules does generate some problems. A module should know in advance when it wants to be called by another module, and such a master-slave relationship is often difficult to identify and maintain. Furthermore, the communication is limited by the granularity of the callbacks and the conditions under which they are specified. In MVC for instance, the module that is called back often has to query the calling object and compare its current state with its previous state to determine any changes.

A slight variation of callback is to use shared memory, where the application program either polls the data to check for any changes or is automatically notified of any changes. But in

general, message passing or event-handling techniques are less efficient than callback procedures.

3.6 Summary

The focus of this chapter was on the architectural and interface issues for single-user applications. Software architecture plays a very important role in the construction of a user interface as the underlying architecture directly affects the behaviour or the look and feel of the user interface. A number of desirable requirements for single-user interfaces were identified; among which separation, direct manipulation and rapid semantic feedback have a major influence on the temporal properties of the interface.

Early architectural models such as Seeheim emphasise the distinction between the presentation, dialogue and functionality aspects of the interface. Whilst the Seeheim model has been developed in various ways, most notably in the Arch/Slinky framework, most user interface architectures preserve some notion of layering between the surface output and input devices and the deep application semantics. Object-based models such as Model-View-Controller and Presentation-Abstraction-Control make similar distinctions but they tend to focus on individual parts of the interface, instead of the global separation of the application into components.

The separation of the application and user interface functionality enhances features such as portability, reusability, customisation and adaptability. However, the decoupling of the user interface functionality from the application functionality is sometimes difficult to achieve, especially when rapid semantic feedback is required. Consequently, aspects of the user interface may 'leak' into the application and vice versa.

The architectural models were analysed further by looking at both conceptual and physical aspects. A conceptual (or logical) analysis helps us to think about user interface development issues whereas a physical analysis identifies whether there really are components of the system with the named roles and communicating along the paths specified in the architecture. This breakdown is clearly visible in single-user interfaces as all the components are placed on the same machine.

In addition to architectural models, a number of interface development tools are available for building single-user applications. Windowing systems provide a very low-level of abstraction for building applications. Consequently, toolkits and User Interface Development Environments are built on top of them to assist developers in their tasks. Toolkits enforce consistency across the interface by defining the interface as a collection of widgets. However, they do not provide enough support for designing interfaces or sequencing specifications and dialogue control. User Interface Management Systems instead goes beyond the toolkit level by offering mechanisms for the run-time management of user interfaces.

Finally, some common design paradigms employed by architectural models and interface development tools in general were explored. Although, it is advantageous to separate an

application's functionality from its presentation, such a degree of separation is more difficult to achieve in systems where the user interface elements are event-based and less so in object-oriented models. In constraint-based models, the link between the presentation and the application are handled through constraints but the control component is external. Consequently, a large amount of information about the application and the presentation has to be represented in the external component. Callback-based systems adopt a notification-based method to invoke the appropriate application procedure to handle events, but like constraint-based systems, the dialogue control is external to the application.

The next chapter will consider similar architectural and interface concerns to those discussed here, but these will be applied to a multi-user context. In a multi-user distributed environment, the software no longer runs on a single machine. One can no longer fudge the boundary and communications between the application and the user interface components as they are enshrined in the physical location and network connectivity.

Chapter 4 Multi-user Interface and Architecture Issues for Collaboration

The user interface was mainly viewed from a single-user perspective until the advent of personal computers and local area networks. Client-server architectures became more popular and they presented a whole set of challenges in the design and implementation of user interfaces. Some of the crucial issues that had to be addressed included the need to support multiple user interfaces while ensuring a consistent interface for each client, and providing concurrency and access control to prevent users from performing conflicting actions.

In distributed interfaces, the application and the interface do not use shared memory. Instead, the application may be running remotely at the server-end while the user interface is executing locally at the client-end. When users are spread across a number of locations, the underlying application should support interfaces that run on a number of workstations across a distributed environment.

The emergence of Computer Supported Cooperative Work (CSCW) has pushed collaborative interfaces into focus. The support for collaborative work has seen the development of groupware that present a number of interfaces for simultaneous interaction by multiple users. Groupware is described as software that supports and augments group work (Baecker et al., 1995). Groupware systems have been developed for several areas, from databases, graphics applications, multimedia systems to conversation boards (Brink and Gomez, 1992) and computer games (Rohall et al., 1992).

Collaborative interfaces are difficult to implement because they not only have to handle interaction tasks associated with single-user interfaces but they must also perform different collaboration tasks. On a technical level, collaborative applications have to manage distributed processes, maintain a robust inter-process communication and a persistent object store. Furthermore, groupware should support some fundamental human factor requirements for promoting effective groupwork, such as setting up and breaking connections dynamically with remote users, managing input and output from multiple users, informing users input and coordinating users interactions (Dewan and Choudary, 1992).

This chapter gives an overview of multi-user interface and architectural issues for collaborative applications based on the existing research literature. Section 4.1 explores the architectural and temporal requirements for multi-user interfaces designed for supporting collaborative work. Section 4.2 examines some common architectural models. Early distributed systems used to implement multi-user applications in a very transparent fashion. However, the need for collaborative applications has driven a number of development tools that enable the sharing of data and coordinate interaction across different interfaces. Section 4.3 reviews some of these tools. Finally, Section 4.4 considers some design paradigms applied in multi-user architectural models and development tools.

4.1 Requirements

Many existing guidelines for user interface design have been developed from a single user perspective. There are very few requirements that exclusively address the needs for multi-user interfaces (Dewan, 1992) and a large number of them merely extend the primitives of their single-user interface counterparts (Dewan and Choudhary, 1991).

Multi-user interfaces designed for supporting collaborative work must facilitate the sharing of application information to promote collaboration among group users. Furthermore, these interfaces should cope with network related problems that may arise with distributed interaction, such as delays, disconnections and network failures. If network problems are likely to occur, the application should also provide enough support to maintain data consistency.

This section considers some of the main temporal and architectural requirements for multi-user collaborative interfaces. Like single-user interfaces, separation and feedback are important issues in multi-user interfaces. However, the need for supporting collaborative work introduces additional requirements of feedthrough, awareness, sharing and control.

4.1.1 Separation

The separation of the application semantics from the user interface functionality is a desirable architectural feature in single-user applications (Section 3.1) as it offers many advantages including portability, reusability, customisation and adaptability. However, the logical separation is sometimes ignored in single-user interfaces in order to reduce the complexity and speed up the development of the application. Also, there is so little in the specification of single-user applications that explains how the separation should be made (Patterson, 1991).

Unlike single-user interfaces, the logical separation is a necessity in collaborative multi-user interfaces as it simplifies the process of visualising the interface in different ways (Patterson, 1991). It is also easier to identify which elements should be part of the application component and which elements should be part of the interface component. For example, the need for multiple views of the shared interface implies that the shared information should be embedded in the underlying application model in order to make it available to all group users.

In addition to the logical separation, the physical separation of the architecture is also desirable for supporting end-user adaptation in a distributed environment. Physical separation provides some degree of fault tolerance, as a failure in a particular user interface process will not affect other user interface or application processes (Bentley, 1994).

4.1.2 Feedback

Like single-user interfaces (Section 3.1.3), the requirement for rapid semantic feedback is an important temporal property in multi-user collaborative interfaces. As well as interacting

with each other, collaborative participants interact individually with the system. Therefore, they require a rapid response following their actions.

Most operations in single-user applications require a feedback within 0.1 seconds (Section 2.2.1) to enable users to feel that the system is reacting instantaneously. This time limit is relatively easy to achieve when all the components are running on a single machine. But in a distributed system, the application and the user interface components often reside on different machines. Consequently, the feedback loop involves transmission over a network and if there are problems such as latency or delays, it is more difficult to achieve acceptable response times.

4.1.3 Feedthrough

In addition to feedback, collaborative users also need to see the effect of one another's action. A large proportion of group interaction takes place via the shared objects. Feedthrough is the reflection of a user's actions on other participants screens (Dix et al., 1993) and it is a crucial temporal property that helps to promote collaboration. Rapid feedthrough is necessary for group work as the artefact is shared by a number of users.

However, the requirements of feedthrough are not so stringent as for those for feedback (Dix et al., 1993). Feedthrough depends on two major factors: the granularity of the updates and the propagation of those updates.

The extent to which a user's activities are represented on another user's screen depends on the degree of *coupling* between the two interfaces. During close collaboration, group members have to communicate frequently with each other before performing any updates to the shared information space. This requires a *tightly coupled* interface. But when collaboration is less direct and less frequent, a *loosely coupled* interface is sufficient.

So, in a tightly coupled cooperative activity such as group drawing, the associated explanation and gesturing involved in drawing an object is often more important than the end product itself (Tang, 1991). Consequently, a small granularity of updates needs to be broadcast to all the users after each action and rapid feedthrough is vital. In contrast, the rate of updates in loosely coupled applications can be reduced significantly – the updates can be *chunked* and broadcast all together at a later stage. Rapid feedback may be necessary for the user who initiated the action but feedthrough to other users can be slower.

4.1.4 Awareness

Traditional distributed systems have applied different types of transparency to hide information from the users, thus giving the illusion that each user is working in isolation. The aim of a collaborative multi-user interface is to facilitate the real-time presentation and manipulation of shared information in order to establish and maintain a common context (Bentley et al., 1994). Collaborative applications must therefore provide users with an awareness of each other's presence and activities to support group work effectively and for establishing successful collaboration (Ramduny, 1994).

There are various types of awareness that have been identified in the research literature. The three major forms of awareness that enhance group work are:

(a) the presence of group members and their availability for cooperative work

(b) the effects of group members' actions and

(c) how changes happen

Awareness of type (b) basically conveys the notion of feedthrough (Dix, 1997) as it deals with the changes that have occurred. Also, the pace of feedthrough is directly proportional to the rate of providing awareness of type (c); one can infer the reasons why changes occur by noticing the intermediate steps and the way changes happen. Both awareness of types (b) and (c) will be affected by network delays and lags.

4.1.5 Sharing

Different users need to be aware of one another's interaction to varying degrees depending on the activities that they are carrying out. The amount of sharing provided to the users depends on the *granularity* (amount of information to be shared) and the *levels* at which the objects are shared at the architectural level.

In general, the majority of groupware systems operate between the *fine-grained* and *coarse-grained* extremes (Dix et al., 1993). For example, fine-grained sharing allows users to edit the same sentence or even the same word within a particular sentence, whereas coarse-grained sharing only allows one user to edit a file at any time.

Objects can be shared at the following levels and each level corresponds to a particular degree of user interface coupling:

(a) *Presentation level* – each participant is presented with the same display of the same subset of the shared information and any update in the presentation is replicated to all other display screens (Bentley, 1994). Systems supporting presentation level sharing are *tightly coupled* and are they are also referred to as What-You-See-Is-What-I-See (WYSIWIS) systems. Example applications include meeting rooms (Begeman et al., 1986) and shared window systems (Lauwers and Lantz, 1990).

(b) *View level* – each user is presented with the same subset of an information space but the actual presentations may be different. For instance, one user may view the data in tabular form while another may view the same data as a graph and they can both interact simultaneously. The view is shared but not the presentation of the view, hence such systems are *semi coupled*.

(c) *Object level* – each user is presented with different, possibly overlapping subsets of the information space (Bentley, 1994). Such applications are *loosely coupled* and they are also called What-You-See-Is-Not-What-I-See (WYSINWIS) systems. An example groupware system is Grove (Ellis et al., 1990), (Ellis et al., 1991) that allows participants to edit different parts of the same document, so the object is shared but not the presentation.

4.1.6 Control

The degree of sharing offered by the interface depends on the types of control enforced at the architectural level. Traditional distributed systems viewed control as dealing with the problems of distribution and such problems were masked from the applications (Rodden and Blair, 1991). For instance, most distributed systems allowed users to know who could access which objects but they did not allow users to know who was accessing a particular object at a particular moment. The control decisions were thus embedded into the system and hidden from the users.

However, due to the dynamic requirements of CSCW applications, one of which is awareness, transparency is the wrong approach (Rodden and Blair, 1991). The common focus on work implies that collaborative participants have to access the same data. As a result, some form of control is required to manage the shared data and the shared objects.

One of the most common concurrency control mechanisms is locking. An explicit form of locking is floor control policies (Begeman et al., 1986), (Stefik et al., 1987a) where the floor control is the responsibility of the central conference agent and users have to make an explicit request for the floor. Users thus take turns in interacting with the application. In contrast, implicit locking is automatically applied when users attempt to access an object. Systems that adopt this mechanism offer no techniques for coordinating group interaction. A lock may be implicitly requested before a user's action and if no one else has the floor, the floor is implicitly granted. But if the floor is already taken, the user's action is blocked until the lock is released.

In some systems, additional protocols are built on top of the locking mechanisms, such as access rights or roles (Leland et al., 1988). Users perform certain tasks depending on the roles they are assigned. Unlike access rights that normally impose a restriction on users' functions, roles are more dynamic in nature. Finally, certain applications do not provide any locking mechanisms; they rely on participants using a social protocol to negotiate simultaneous access in a free-for-all situation. However, such systems must usually have some ways of detecting conflicts to restore consistency automatically or at least alert users when conflicts occur.

This section has shown that, in addition to the requirements of separation and rapid semantic feedback for single-user interfaces, cooperative multi-user interfaces are governed by the need to facilitate the sharing of application information to promote collaboration among group users. So, in order to meet the needs of group users, feedthrough, awareness, sharing and control are essential requirements. However, the properties of the supporting architecture delimit many of the features of the cooperative interface provided by an application.

4.2 Architectural models

Unlike single-user systems, which have seen a number of research characterising software architectures (Bass, 1993), (Myers, 1995), (Nigay and Coutaz, 1993) the contribution in

multi-user architectures has not been so prominent. Most of the work that have addressed groupware systems mainly consist of identifying specific architectures such as centralised and replicated window architectures and hybrid architectures. This section looks at the benefits and drawbacks of each of these architectures.

4.2.1 Centralised architecture

The centralised architecture consists of a client program that runs on each user workstation and a server that runs as a dedicated program on a central computer that holds all the application's data (figure 4.1). This arrangement is also known as the client-server architecture. The client manages the screen layout and accepts input from users while the server broadcasts output events by routing them through local client programs to all the users.

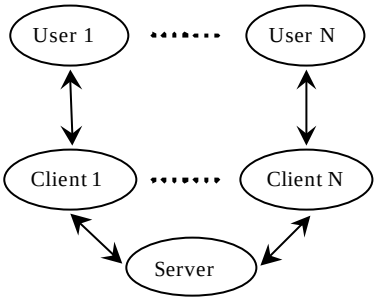


Figure 4.1 Centralised architecture

The client-server approach is very simple to implement, as it essentially comprises one program with several front ends. Because the application and its related data are held centrally, it is easier to manage concurrency control, data consistency and access management. Client-server architectures have been adopted in a number of systems, for example, MMConf conferencing system (Crowley et al., 1990) and shared X window system (Gust, 1988).

Presentation level or WYSIWIS sharing is easily supported in the centralised architecture, for it only involves the server replicating the output to all the clients. View level and object level sharing can also be supported, but the interaction and visualisation have to be embedded in the central server, thus inhibiting end-user interface tailoring. Another disadvantage with having a central server is that large amounts of output may potentially act as a bottleneck (Lantz, 1986) and in case of a failure due to a network breakdown or a delayed feedback, this may eventually lead to a deadlock state.

4.2.1.1 Rendezvous Abstraction-Link-View architecture

The Rendezvous system (Patterson et al., 1990) is based on a centralised architecture with an underlying Abstraction-Link-View (ALV) architectural model (figure 4.2). Rendezvous assumes that the state of a multi-user application is encapsulated in objects and each multi-user application consists of an abstraction and several, possibly different views. The sharing

and consistency between these objects are achieved by establishing constraints between the shared abstraction and views.

Figure 4.2 ALV architecture

The functionalities of the components of ALV are:

- Abstraction – this stores and provides access to the abstract information of the application in other words, the information that is common to all the views.
- View – this presents information to the users that enable them to modify the display. Although this information may be redundant, it is not replicated and besides, it allows views to update their displays quickly even when access to the abstraction is impaired by network delays or high loads.
- Link – this is a two-way *constraint* mechanism that maintains consistency and facilitates the communication between the view and the abstraction components. It ensures that the redundant information is kept consistent by mapping the underlying data with the presentation.

For example, in a pie chart or bar chart display, the *abstraction* component will store the numeric values of the information presented as pie slices or bars. The *view* will have the abstraction information in the form of slice angles or bar heights and the *link* will be responsible for transforming the bar heights to raw numbers and vice versa.

The *abstraction* and *view* objects in the ALV architecture correspond to the single-user MVC (Section 3.2.3) *model* and *view* components respectively. When an interactive system is decomposed into view (presentation) and abstraction (application) components, some information may belong on both sides. But with ALV, the constraint mechanism helps to keep the redundant information consistent. By implementing the communication constraints in the link object, the view and the abstraction components become independent of each other and are consequently easier to design and implement. Furthermore, this allows the re-use of the view and the abstraction components, thus simplifying rapid prototyping.

The centralised architecture of Rendezvous simplifies the process of synchronising the interfaces, however it suffers from performance drawbacks, as the abstraction and the associated view objects execute at a central location. Rendezvous does provide object, view and presentation level sharing but the centralised architecture does not allow the sharing policy to be visible; hence end-user tailoring is not supported.

4.2.2 Replicated architecture

In a replicated architecture, a separate copy or a replica of the application runs on each client workstation (figure 4.3). Each replica executes the application code and sends feedback locally. Groupware applications that adopt a replicated architecture tend to be highly interactive and require immediate response, for example GroupDraw (Greenberg et al., 1992a) and GroupDesign (Beaudouin-Lafon and Karsenty, 1992).

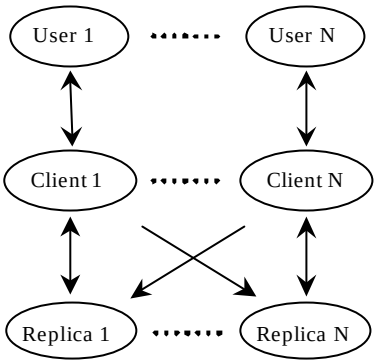


Figure 4.3 Replicated architecture

The replicated approach offers the advantages of a centralised architecture with the added benefits of performance, as the output of a workstation is produced by the local workstation itself (Dix et al., 1993). Because the clients can be managed locally, view level and object level sharing are easily supported. Also, it is relatively easy to provide end-user interface tailoring, as each replica can simply adapt its visualisation and interaction policies to the users preferences (Bentley et al., 1994).

The main problem with the replicated architecture lies with synchronisation and maintaining data consistency. Usually, the input from each workstation is sent to each replica to ensure synchronisation among the different replicas. The copies then communicate with each other to maintain data and interface consistency. However, when several users perform simultaneous actions some conflicts may arise. For instance, if a user deletes a selected object in a WYSIWIS group drawing program while another user is changing the selection to a different object, inconsistent interfaces can occur due to events arriving in a different order at each workstation. Such conflicting actions will lead to *race conditions*.

Complex synchronisation algorithms are required to deal with race conditions. A standard solution traditionally applied in distributed computing is to use a global clock to timestamp each event. If any inconsistencies arise, the events are rolled back and re-executed in temporal order. This approach is however inappropriate for multi-user interfaces where display screens are updated immediately after each user's input. Some groupware systems, such as GroupDraw and MMConf (Crowley et al., 1990), assume that race conditions rarely occur and they ignore the problems of synchronisation. Others like GroupDesign and Grove (Ellis et al., 1990) attempt to synchronise events at the expense of very complex and computationally expensive algorithms.

Alternative concurrency control mechanisms based on transforming updates to prevent rollback have also been developed. These mechanisms include locking and floor control and they can only prevent race conditions or tolerate race condition in situations where users can obtain locks and when rapid feedback is not the major concern (Dix et al., 1993). Real-time synchronous update will still demand special-purpose algorithms.

A further problem with replication occurs when latecomers join a shared session, for example, in a conferencing system. It is relatively straightforward to handle newcomers in a centralised approach, as new clients only have to contact the server and register their existence. The server then broadcasts the current state of the application to bring the new client up to date. But with a replicated architecture, a new replica has to contact all the other replicas to find out about the current state of the application and receive any updates. Therefore, the new replicas must be aware of the locations of all other current replicas or at least should be able to find them out.

Despite the difficulties with the replicated architecture, this approach has two key advantages over the centralised approach – performance and versatility.

4.2.3 Hybrid architecture

Very often, neither a pure centralised nor a pure replicated architecture fully meets the requirements of a system. It may therefore be more effective to adopt a hybrid architecture, where certain parts of the systems are centralised and others are replicated depending on the application's requirements. Suite (Dewan and Choudary, 1992), (Dewan, 1993) is an example of a distributed hybrid architecture that has a central semantic component, where the application resides and local user-interface components, represented by dialogue managers (figure 4.4).

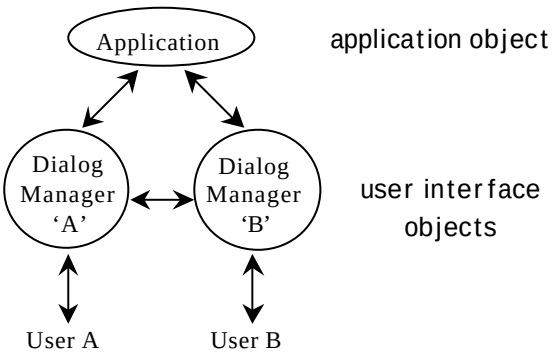


Figure 4.4 Suite hybrid architecture

Suite, like the Seeheim architectural model (Section 3.2.1), separates an interactive program into application objects and user interface objects. Each user interface object in Suite is further divided into multiple dialogue managers and each dialogue component manages the interaction between a particular user and the application.

The Suite architecture is designed to keep network traffic low. Although the semantic tasks are performed remotely in the central application component (which may involve some communication delays) there is no delay in users viewing the displays on their screens, as the dialogue managers format the computation results locally (Dewan, 1993). It has been argued that the communication delays at the application-end are usually less perceptible because computation size is typically small compared to its display (Dewan, 1993). The other assumption is that results are not usually generated frequently for users tend to execute long transactions before committing their input values.

The hybrid architecture is also supported by the MMM multi-user editor framework (Bier and Freeman, 1992), which is essentially based on a centralised architecture, but the data is replicated to support local insertion points, feedback and style setting. In contrast, the XGroupSketch multi-user drawing program (Greenberg et al., 1992b) is mainly replicated, but all the updates, communication and user-registration are handled via a central component. In this case, the replicas only need to know the location of the central component to broadcast the updates.

This section has provided an overview of the various architectural models that have been developed for implementing multi-user collaborative applications. Most systems tend to adopt either centralised or replicated window architecture. Each architecture meets the requirement of collaborative systems by supporting different levels of information sharing with different degrees of complexity as summarised in table 4.1. However, a hybrid approach is very often more effective to meet the needs of collaborative users, as certain parts of the systems are centralised and others are replicated depending on the application's requirements.

Features	*Centralised architecture*	*Replicated architecture*
implementation	easy	difficult
late user registration	easy	difficult
synchonisation/ data consistency	easy	difficult
rapid feedback	no	yes
presentation level sharing	yes	yes
view/object level sharing	no	yes
end-user interface tailoring	no	yes

Table 4.1 Centralised vs. Replicated architecture

4.3 Interface development tools

A considerable amount of research has been carried out in the real-time presentation of multi-user interfaces within a collaborative environment. Most of them have focussed on developing techniques, tools and facilities to coordinate the interaction across different interfaces and these are discussed below.

4.3.1 Shared window systems

Shared window systems allow existing single-user applications to execute in a multi-user environment with minimal changes. They are merely extensions of single-user window systems that support sharing. The effects of the user's actions are shared across a number of displays in a transparent fashion. Applications based on the shared windows approach are also known as *collaboration transparent* applications.

The NLS teleconferencing system (Engelbart, 1975) was one of the first shared applications developed that allowed each user to share their complete display screens. The sharing of parts of a display or individual windows came with the development of windowing systems, such as Vconf (Lantz, 1986) and Rapport (Ensor et al., 1988).

The logical structure of a shared window system is shown in figure 4.5. A central conference agent is responsible for managing the interaction between the users and the applications (Lauwers and Lantz, 1990). The central agent multiplexes output streams from the applications onto the users window systems (figure 4.5a) and demultiplexes the input streams from all users to the appropriate application (figure 4.5b).

A shared workspace is thus created and this allows each user to see the same view of every window associated with the shared applications. Shared window systems are usually built on top of an existing network windowing system, which is responsible for handling the communication between the applications and the displays via a network protocol. Examples of collaborative systems based on this approach are SharedX (Gust, 1988) and MMConf (Crowley et al., 1990).

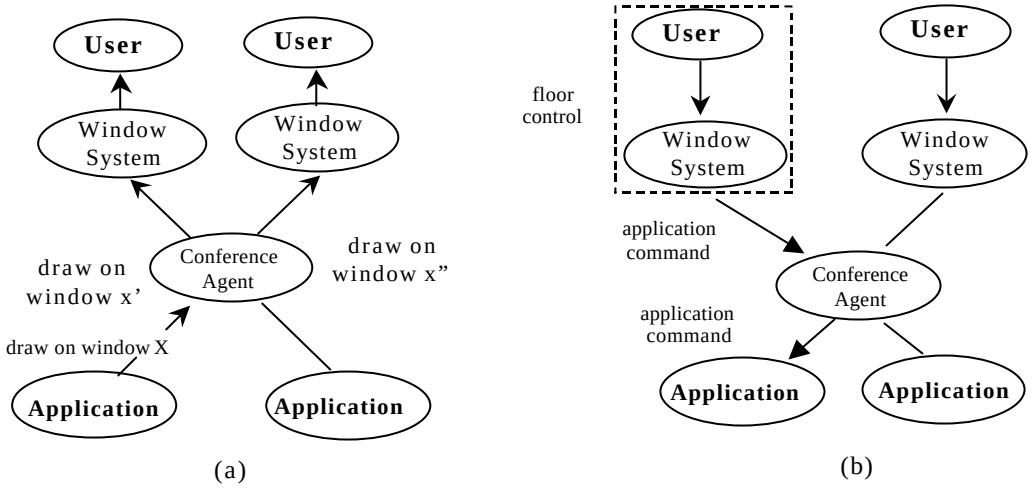


Figure 4.5 (a) Output and (b) Input structure of a shared window system

Because shared window systems deal with a single stream of output and input events, only one user can interact with the application at any given time. A suitable concurrency control mechanism is therefore required to coordinate the interaction between the participants. This is usually in the form of some floor control policy that is managed by the central conference agent. However, floor control policy is a contentious area of debate, as no single policy will suit the needs of all group members in all task contexts (Lauwers and Lantz, 1990). Floor control policies are usually based on some turn-taking protocols or on policies that range from allowing only one user to control the shared workspace at a time, to allowing anyone to generate input at any time to any window (open floor).

Most shared window systems are based on replicating the output via the sharing of the *presentation* and thus they do not have an explicit knowledge of the shared task. Such interfaces provide a common frame of reference to all participants by allowing users to refer to the same visual context (Lauwers and Lantz, 1990), (Dix et al., 1993). Earlier systems supported *strict* WYSIWIS, where all participants had the same window size and placement. This arrangement was not only found to be confusing and distracting for users but it was also overly restrictive for group members especially when they have widely differing roles and assume different tasks (Greenberg, 1990). Hence, the MMConf system (Crowley et al., 1990) adopted a more *relaxed* WYSIWIS approach and users were allowed to make decisions about the window layout locally.

Although shared window systems increase the power of interactive systems significantly by supporting collaboration tasks, they can only manage low-level interaction entities and provide support for near-WYSIWIS interaction (Dewan and Choudary, 1992). For instance, a shared X window system forces users to share scroll bars, as the notion of scroll bars is not defined at such a low level.

4.3.2 Shared object systems

Shared object systems enable the collaborative sharing of data and they are often referred to as *collaboration aware* systems. Collaboration awareness supports the development of specialised applications designed for simultaneous use by multiple users (Lauwers and Lantz, 1990). These applications provide facilities for managing information sharing explicitly between the participants and they present a number of different interfaces to the users. Example systems include the CoLab meeting room (Stefik et al., 1987b), Grove group editor (Ellis et al., 1990) and rIBIS real-time information system (Rein and C., 1991).

Collaborative aware systems like collaborative transparent applications, provide presentation level sharing. For example, in the CoLab collaborative meeting system (Stefik et al., 1987b), all clients execute the same application program and the state of the objects can be shared among the clients. The objects encapsulate both the semantic state and the user interface state. If a shared state object is modified, such as the scrollbar, this change is replicated to the other clients via broadcast methods, thus allowing the users to view the changes without encountering any network delays.

CoLab implements both a strict floor control policy to manage users interaction with the display and a relaxed WYISWIS approach to provide a coordinated interface for all participants. The latter allows participants to have the same presentation of the shared information, which can be arranged into *shared* and *private* windows as required (Stefik et al., 1987a).

Grove (Ellis et al., 1990) separates the users shared and private windows by applying the notion of views that links the users access rights with their presentation. For example, private views are only accessible to the owner, shared views are accessible to a group and public views are accessible to all the participants.

In general, collaboration aware systems often embed the management of each user's sharing within the application itself. As a result, the decisions regarding the display and the ways in which the information can be modified are embodied in the application itself. This inhibits the tailoring of the sharing policy and hinders multi-user interface prototyping (Bentley, 1994). Furthermore, the lack of a supporting infrastructure means that most collaboration aware applications have to be built from scratch. Hence, this approach has been less popular than collaboration transparency. However, due to flexibility reasons, collaboration aware arrangements are becoming more prominent (Greenberg, 1990).

Table 4.2 summarises the main features of collaboration transparent applications and collaboration aware systems.

Features	*Collaboration transparent*	*Collaboration aware*
easy to implement	✓	
support specialised synchronous multi-user application		✓
information sharing		✓
presentation level sharing	✓	✓

view/object level sharing		✓
private windows	✓	✓
shared windows		✓
flexibility		✓

Table 4.2 Collaboration transparency vs. Collaboration aware

4.3.3 Groupware toolkits

Conventional multi-user toolkits do not support many of the requirements of groupware, such as managing synchronisation, concurrency and communication on a technical level. In addition, they do not incorporate the fundamental human factor issues that are necessary for promoting effective group work.

Groupware toolkits instead provide the key components for common groupware needs, thereby reducing the development time and increasing the quality of multi-user applications (Roseman and Greenberg, 1992). They also allow the rapid prototyping of applications and enable various aspects of cooperation to be customised. Example toolkits for real-time distributed meetings include GroupKit (Roseman and Greenberg, 1992), Rendezvous (Patterson, 1991), (Hill et al., 1994), Suite (Dewan and Choudary, 1992) and MMConf (Crowley et al., 1990).

GroupKit (Roseman and Greenberg, 1992) allows the development of real-time conferencing applications for geographically distributed or face-to-face meetings. It is based on a replicated architecture that offers support for sharing, access control and floor control. The predefined classes of GroupKit can extend a single-user application developed with a single-user toolkit to a multi-user application and also implement multiple concurrency and access control policies.

MMConf (Crowley et al., 1990) also supports a replicated architecture, but each replica behaves as a distinct logical entity that manages its own state. Each replica can therefore determine whether an input event generated by its user should be broadcast to other replicas or not. As a result, operations with side effects can only execute on a single replica, thus reducing the occurrence of race conditions.

Groupware toolkits generally have a model of the user interface data structures such as windows and widgets. They enable the sharing of the user interface state by allowing multiple copies of interactive elements to be created but they do not manage the application state. So, even the conceptually simple idea of having multiple cursors on a display and annotating artefacts used for promoting gesture, awareness and note taking over a visual surface is difficult to achieve with toolkits (Hayne et al., 1993).

4.3.4 Multi-user User Interface Management Systems

A fundamental issue with single-user User Interface Management Systems (UIMS) is the separation of the application semantics from their screen representations (Section 3.4.3).

Separation is often difficult to achieve because current direct manipulation interfaces require extensive communication between the user interface and the application to provide rapid semantic feedback. However, in a multi-user UIMS context, separation is crucial in order to provide the necessary high-level abstractions to support the sharing of both applications and user interfaces.

The Rendezvous system (Patterson et al., 1990), (Patterson, 1991), (Hill et al., 1994) allows multiple users to build interactive systems on multiple workstations simultaneously, while promoting the run-time separation of the user interfaces from the applications. Figure 4.6 shows how the ALV architecture, discussed previously in Section 4.2.1.1, is implemented at run-time. It consists of a tree of objects with a central shared abstraction for handling user interaction and display management, and an associated view process that interprets input events and display directives for each user (Hill, 1992).

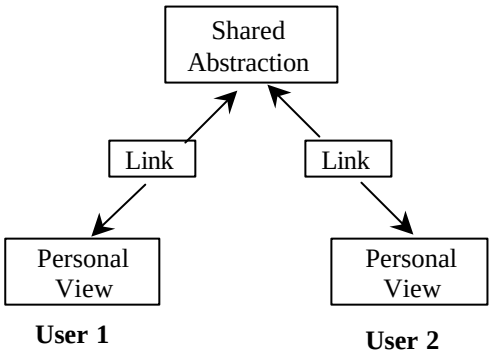


Figure 4.6 Run-time ALV architecture

So, for each multi-user program, Rendezvous creates a single abstract process that stores the abstract objects defined by that program. Similarly, a view process is created for each user, which stores all the view objects associated with that particular user. An abstract object can therefore be shared among a group of users by creating a view object for each user and using constraints to keep the abstract objects consistent with the view objects. In this manner, visualisation and interaction are separated from the information being shared, thus facilitating the alternative presentations of information.

4.3.5 Multi-user interface generator

A multi-user interface generator is defined as a multi-user UIMS that automatically generates views of abstract data from a high-level description of those views (Dewan, 1992). Like a multi-user UIMS, a multi-user interface generator manages the visualisation and manipulation of information outside the application. A separation between the application information (what is being shared) and the interface presentations (methods of sharing) is achieved by hiding the physical distribution of the components from the application developer. Consequently, the visualisation and interaction policies can be tailored independently of the application.

Examples of multi-user interface generators include Suite (Dewan and Choudary, 1992), (Dewan, 1993) and MEAD (Bentley, 1994). Although the default collaboration scheme offered by Suite is collaboration transparency, collaboration aware programs can be supported by using some powerful set of constructs or primitives that control different aspects of collaboration, for instance, the tailoring of users input and output, and user interface coupling (Dewan and Choudary, 1992). Users are also allowed to define different collaboration attributes with different users, such as sharing, communication and access attributes. MEAD instead, provides high-level tools for interface development and its supporting architecture manages prototype interfaces in execution, thus making it more suitable for rapid multi-user interface prototyping.

This section has given an overview of the various types of development tools for implementing multi-user collaborative interfaces. However, the success of any development tool depends on a number of factors. Firstly, the tool should be flexible and it should support the construction of a wide range of interfaces. Secondly, the effort required to implement or change the user interface should be minimal, so ideally the tool must be automated. Finally, the tool has to be efficient and this will depend on how well the user interface performs.

4.4 Design paradigms

Various mechanisms have been adopted to facilitate the communication between the user interface and the application components in collaborative multi-user architectures. This section will explore some of the ways in which architectural models and interface development tools have been implemented. These include the use of constraints, callbacks, and active values.

4.4.1 Constraints

A constraint system allows a set of source variables to be linked to a target variable, so whenever there is a change in any source variable, the value of the target variable is set to a specified function of the source variables (Hill, 1992). Unlike single-user constraint systems that only support a single constraint to be inbound on a value (one-way constraint), multi-

user systems require multiple constraints. Constraints embody dependencies between different values that must always be maintained.

As discussed previously in Section 4.2.1.1, the ALV architecture in Rendezvous uses constraints to maintain consistency across multi-user applications. The use of constraints enables the link component in ALV to be described separately from the abstraction and the view components; hence the view can ignore the abstraction and vice versa. Constraints can automatically retarget themselves as views and view objects can be easily created, restructured or deleted. The constraint method for communication does not seem to pose performance problems as graphic updates have shown to consume far more processor resources (Hill, 1992).

Rendezvous is designed to provide rapid feedback independent of the number of users, through the use of some form of redundancy like caching the user interface. Constraints are also employed in Rendezvous to maintain consistency between caches and "real" values.

4.4.2 Callbacks

Callbacks are basically procedures invoked in applications in response to user actions, for instance, when a user connects to an application or when changes to data structures have to be checked for semantic consistency. The single-user Suite system (Dewan, 1990) uses callbacks and so does the MVC architectural model (Section 3.5.4). In theory, MVC could support multi-user interfaces with multiple view-controller pairs but in practice, problems with concurrency and bandwidth requirements of the callback protocols are said to arise (Hill, 1992). A potential solution is to fully replicate all three components for each user instead of using multiple view-controller pairs for each model (Smith et al., 1989). However, this replicated approach can only be used when all the users have identical interfaces.

Constraints act as an effective communication mechanism and they have often been used in adhoc ways to link values and not objects. Callbacks can be used in its place to link objects but they are a poor communication mechanism. Callbacks are rather static and they force communication to be coded procedurally into all the relevant components. Consequently, codes have to be written for each interface to ensure that callbacks are registered and de-registered as required and this limits the scope of reusing the components.

4.4.3 Active values

Active values are variables that allow other objects to register functions with them (Hill, 1992). Whenever an active value changes, it calls the relevant registered functions. Active value systems behave like callback systems, as they are implemented through the callback mechanism. However, active value systems are at a higher level of abstraction – an active value can in fact be considered to be an instance of a callback. Both active values and callbacks react to events. With active values, this reaction can cause an update in the relevant variables, but callbacks may not have the same effect.

Multi-user Suite (Dewan, 1992), (Dewan, 1993) uses active values to extend single-user Suite (Dewan, 1990) which actually uses callbacks to multiple and possibly distributed, users. Users can change active values via an interaction variable – a user's local version of the active value of an object. An interaction variable is automatically created by the system when the user connects to a particular object. The users can subsequently modify the interaction variable by changing the interaction attributes associated with it. These attributes determine the properties of the variable, for instance, the format used to display the variable. The system then invokes the relevant callbacks to make the necessary changes at a lower level.

4.5 Summary

This chapter has highlighted the major architectural and interface concerns for collaborative applications. Like single-user systems, the notion of separation and feedback are important requirements. However, groupware systems need to facilitate effective collaboration among users. Collaborative users should therefore be able to manipulate the shared information in a timely fashion and they should also be aware of each other's activities with minimal delay. The requirements of feedthrough, awareness and sharing are critical in meeting the needs of collaborative users. Furthermore, it is essential to maintain data consistency between the displays of the shared information and the information itself. Some effective control mechanism is required for handling change propagation.

The supporting architecture usually governs the features of the interface. Some common architectural models for collaborative systems were reviewed. Such systems adopt either centralised or replicated window architectures and when these are not suitable, hybrid architectures are used, where certain parts of the systems are centralised and other parts are replicated. Unlike the replicated architecture, the centralised architecture is easy for implementation purposes, for adding and removing clients and for maintaining consistency. However, the replicated architecture does have the advantage of providing different levels of sharing, rapid feedback and end-user tailoring. Very often, a hybrid approach is more effective for meeting the needs of collaborative users.

The underlying architectures for collaborative systems such as Rendezvous and Suite were also investigated. Both architectures allow the coupling of semantic values without coupling their presentations. However, in Rendezvous, the view objects (presentation) for a user executes at a central site, in the same address space as the corresponding abstraction objects (application); whereas in Suite, the dialogue manager (presentation) executes in its own address space. Consequently, Rendezvous is not so flexible with the placement of its components.

A number of CSCW applications are based on the shared window and the shared object approach, and they are referred to as collaboration transparent and collaboration aware systems respectively. The tools used in interface development were also explored and they included groupware toolkits, multi-user user interface management systems and multi-user interface generators. These tools provide a much higher-level of abstraction than the shared window or the shared object approach.

Collaboration aware features are useful for enhancing awareness among users and for providing public and private views whereas collaboration transparent features are essential when any participant's interaction becomes disruptive to others or when a private workspace is necessary. Collaboration transparent applications are completely unaware of the presence of multiple users and their interactions but collaboration aware arrangements offer more flexibility. Often, a hybrid approach may be more suitable for developing cooperative multi-user interfaces as it encompasses both aspects of collaboration.

Finally, some common design paradigms applied in implementing multi-user architectural models and development tools were considered. Constraints were found to be an effective linking mechanism between the separate components of an architecture, as typified by the Rendezvous ALV architecture. Callbacks are not very efficient as a communication mechanism between separate components, but they have been used successfully to link objects together. Active values show similar behaviour as callbacks as they both react to events, but in the case of active values, this reaction in turn invokes another callback, which makes the relevant changes at a lower level.

The issues discussed in this chapter emphasise the need for separating collaborative architectures into various components to provide effective user-level behaviour. Even in stand-alone systems, a poor separation between the components can reduce the performance of the system and create unacceptable interface delays, as described in Chapter 3. The development of the Web has forced the concern between *where* the data is stored and *where* the control lies, thus generating various alternatives for the location of architectural components. The location decision of each component is decisive in achieving the temporal requirements of rapid feedback and feedthrough and this is the subject under consideration in the next chapter.

Chapter 5 Why, What, Where, When: An analysis of Collaborative Architectures on the Web

The Web is a ubiquitous platform-independent infrastructure that has a lightweight extensible centralised architecture, cross-platform browser implementations and an existing user base numbered in millions. With such an extensive set of functionalities, the Web offers immense potential for the development of CSCW applications that provide much richer support for collaboration. The Web can facilitate the development and implementation of remote collaborative applications, despite the limitations in the range of applications that can be directly supported.

Issues such as network bandwidth, reliability and performance have become critical with the increasing use of the Internet. They have a direct influence on the temporal behaviour offered by the interface. Although collaborative applications can be developed in an ad hoc fashion, it is widely recognised that for both single-user and multi-user interfaces, an appropriate software architecture is required as an aid for design, portability and maintenance (Bentley et al., 1994), (Hill et al., 1994), (Pfaff and Hagen, 1985). Interface and architectural issues surrounding single-user and multi-user applications were discussed in Chapters 3 and 4 respectively.

This chapter analyses some important architectural decisions that need to be considered when constructing collaborative applications for the Web. Like distributed systems, which allow data and code to be moved to achieve the desired behaviour, on the Web, Java applets can be downloaded to give rapid local semantic feedback. Architectural decisions on the Web however, do not solely lie in the choice of the physical location for each functional component. They also depend on *when* that component should reside in different places. The analysis presented here examines the reasons that determine the optimum placement for each component. Issues in this chapter have been discussed in (Ramduny and Dix, 1997a), (Ramduny and Dix, 1997b).

Section 5.1 starts with a brief overview of the Web architecture. It examines the limitations of the Web as a development platform for collaborative applications and considers some of the ways of removing the constraints of the basic Web architecture. Section 5.2 introduces the analytic focus of this chapter. Section 5.3 assesses *why* certain behavioural issues are critical for collaborative work and Section 5.4 investigates *what* components are necessary in collaborative interfaces. The decisions regarding *where* the components in a distributed architecture should be placed and their consequences are discussed in Section 5.5. Section 5.6 then examines *when* components in a networked environment should be moved to improve performance.

The issue of mobility surrounding data and code emerges from the analysis and this is explored further in Section 5.7 by focusing on the options available for the Web. Finally,

Section 5.8 sets out the main behavioural objective of this research and examines its influence on the architectural framework that will be developed to support collaborative applications on the Web.

5.1 Overview of the Web

The Web was originally intended to be a powerful tool for supporting 'active' forms of collaboration between collaborators in remote sites through the sharing of ideas surrounding a common project (Berners-Lee et al., 1994). However, over the years the development of Web browsers, servers and protocols have largely concentrated on more 'passive' forms of information browsing and the initial concept of an 'active' form of collaboration was set aside (Bentley et al., 1997b).

This section first considers the Web architecture, followed by its limitations as a development platform for collaborative applications and finally looks at some of the ways in which the Web functionalities can be improved to facilitate the construction of collaborative applications.

5.1.1 Architecture

The Web is based on a simple client-server architecture. Web browsers run at the client-end and interact with a central server component (figure 5.1). A browser identifies the information required by using the standard Uniform Resource Location (URL) naming scheme and requests information from the server through the standard HyperText Transfer Protocol (HTTP) (Fielding et al., 1997). The host server then sends an HTTP response back to the client.

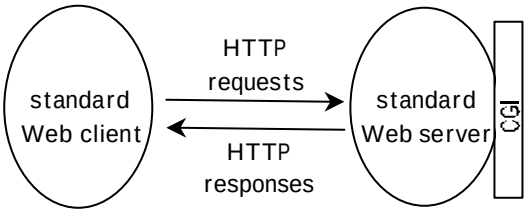


Figure 5.1 Web client-server architecture

The response consists of a header, which identifies the data type (MIME) and allows the browser to handle the format correctly, and a body, which contains the requested information. The client browser uses the header information to display the information in the right format, for instance in text or graphics form, by launching an external 'helper' application such as Microsoft Word to display a Word document or a browser plug-in such as QuickTime to view video files.

HTTP is a generic, stateless object-oriented protocol, based on a simple request-response model. The client browser uses the GET 'request' method to ask the host server for a specific service, like an HTML page or a video clip, and the POST 'response' method to

transmit HTML data to the server. A stateless protocol implies that no state is associated with a network connection. This has several advantages. Firstly, it increases the robustness and the efficiency of the connection, as a dropped connection will only affect a single request and a connection state does not have to be established each time a connection is created. Secondly, a stateless protocol eliminates the need for a resynchronisation operation to recover a connection state following an interruption. Finally, servers can process requests from client browsers independently without affecting any previous requests, thus enabling the development of lightweight server components.

State information is typically preserved by the client and is then passed on to the server as part of the HTTP request. The HTTP protocol is independent of the format of the data transmitted. The clients and servers are responsible for handling new data formats usually through some extension. Client browsers can handle different types of data by using helper applications whereas the functionality of Web servers can be extended through Application Programming Interfaces (API). Helper applications usually function like many database client-server applications and they can be in the form of additional protocols running independently or in parallel with the current Web protocol.

The Common Gateway Interface (CGI) [5] is the de-facto standard for interfacing external applications with information servers such as HTTP or Web servers. Unlike an HTML document, which is static in nature, a CGI program is executed in real-time and produces dynamic information. The CGI approach is independent of any particular server architecture and allows rapid development. In CGI scripts, the interface and the application sit on the server side to produce dynamic pages and the Web browser is used as the presentation manager at the client-end. Dynamic pages allow changing information to be displayed, but this can often cause server-end overload.

5.1.2 Limitations

The Web already offers global access to Web pages, which in some ways can be regarded as a kind of shared artefact. Although the Web allows users to search, browse, retrieve and publish information fairly easily, it does not currently offer features for sharing information in a more cooperative fashion, such as facilitating authors to produce a joint document (Bentley et al., 1997b). The main difficulties with supporting shared activities stem from the Web architecture itself, the existing protocols and browser limitations.

[5] http://hoohoo.ncsa.uiuc.edu/cgi/intro.html

5.1.2.1 Asymmetric nature

The Web is asymmetric in nature due to its intrinsic distributed features. As a result, it cannot support symmetric access to the shared artefacts. Updates only occur where the pages are stored and readers are simply allowed to view the pages. Equal access to the shared artefact is an essential requirement for enhancing collaboration (Section 4.1.5).

5.1.2.2 Lack of awareness

Awareness is another important requirement for assisting and promoting collaborative work (Section 4.1.4). However, awareness mechanisms are practically absent within the Web context. Changes to normal Web pages are only noticed when the page is visited. Despite their advantages, stateless servers answer for the lack of implicit notification services on the Web and thus render the HTTP protocol unsuitable for sending notification messages (Trevor et al., 1997), (Dix, 1997). Some Web-based applications do however provide explicit forms of notification, for instance they send emails to users when updates take place. Even then, the absence of any client-server notification increases the likelihood of the interface becoming inconsistent with the information held on the central server, unless users reload the page frequently.

5.1.2.3 Restrictive architectural arrangement

The Web does not fully support collaborative arrangements such as direct server-client, client-client or replication across servers (Section 4.2). This is essential for applications where the server need to play a more active role, such as notifying users for changes or maintaining consistency across several servers. Some applications do poll Web servers periodically to check for updates, but if the pages rarely change polling will generate unnecessary network traffic.

5.1.2.4 Feedback delays

The CGI approach for generating dynamic information on the Web is based on a request-response model. This increases feedback delays, as the server has to be contacted after each user input. Such delays may be acceptable when for example a document is requested, but less so during simple requests that only involve a change in the state of the interface. In addition, simple computations, such as checking whether a user has filled in all the fields in a form, should be performed at the client-end to reduce unnecessary network traffic, high server load and slow user feedback.

5.1.2.5 Unreliable transmission

The HTTP protocol does not guarantee the transmission rates between servers and clients. Data transfer varies during a single transmission depending on the network and the server load. Continuous media like audio and video require a more reliable transfer mode, thus they use the alternative Real-Time Protocol (RTP).

5.1.2.6 Poor user interface

HTML is not a user interface design toolkit and it does not provide any support for common desktop features like drag and drop, multiple selection and semantic feedback.

5.1.3 Improving functionality

The above limitations restrict the scope of the Web as a development platform that is mainly suitable for asynchronous centralised applications and offer no support for synchronous notification, disconnected working and rich user interfaces. However, the Web has a significant advantage in that it is an accepted technology which is easily integrated with existing user environments and extensible through the server API. Also, users do not require additional client software to run on their machines. The following approaches have helped to remove some of the constraints of the basic architecture and make the Web more amenable as a development platform for collaborative systems.

5.1.3.1 Using CGI scripts

A simple and quick method for extending server functionality without modifying the protocols, browsers or servers is through CGI scripts. BSCW (Bentley et al., 1997b), (Bentley et al., 1997a) is an example system that provides a Web forms interface to a collaboration support system by integrating collaboration services with an extension of a standard Web server using the CGI programming interface. BSCW also provides HTTP upload and download support.

5.1.3.2 Implementing dedicated servers and clients

A special-purpose Web server can be implemented to improve performance, security and introduce new server functionality such as server-initiated notification. Unlike CGI scripting, this method is more flexible and secure for accessing existing Web applications. The BASIS WEBserver[6] is an example of a specialised server approach that enables Web access to the BASISplus document management system. In addition, a dedicated client can allow applications other than Web browsers to communicate with Web servers using HTTP. An example of a specialised client is the 'coordinator' clients in the WebFlow distributed workflow system (Grasso et al., 1997).

Servers and clients can also be customised to provide additional services. For example, in the Virtual Places system[7], the Ubique client interacts with the Virtual Places server to provide synchronous communication and allow users to be aware of the presence of other users.

5.1.3.3 Augmenting Web interface

[6] http://library.llnl.gov/basisbwdocs/bwintro.htm

[7] http://www.vplaces.net/

The basic Web functionality can be augmented by replacing the client and server components to provide richer mechanisms for the user interface, synchronous notification, update propagation and information replication. Worlds (Fitzpatrick et al., 1995) is an example of such a system. However, the end product is likely to have specific hardware and software requirements and a lack of integration with existing user environments, thus limiting its accessibility and scope of use.

Some alternative solutions exist at the server and browser level that gives developers more flexibility. For instance, the Apache server allows certain aspects of the server functionality, which cannot normally be accessed through CGI scripts, to be modified (Thau, 1996). Netscape's 'Plug-in' development kit and Microsoft's 'ActiveX' environment simplifies the process of embedding other applications in standard Web browsers. They also allow Web browsers to handle different media types directly. These extended client server programming interfaces increase the possibility for developing much richer CSCW applications that can be fully integrated with desktop environments.

The use of 'mobile code' is another promising area on the Web (Bentley et al., 1997b). Mobile code allows a client browser to download application programs or Java applets and execute them locally. Applets produce much richer user interfaces than HTML and they can support special protocols like different media types, and various collaboration services, such as event notification, simple text chat and more. In addition, applets can provide users with faster response rates by moving computation closer to the clients, subsequently reducing network traffic, server load and feedback lags. There are however security concerns that arise when code is downloaded over the Internet and executed on a user's desktop, but some progress have been made in this area.

5.1.3.4 Enhancing network protocol

The measures discussed so far mainly rely on using the standard HTTP network protocol. This protocol cannot by itself effectively meet the demands of highly interactive collaborative tasks like collaborative authoring (Whitehead and Y., 1999); therefore an enhanced network protocol is required. WEBDAV (Whitehead, 1997), (Goland et al., 1999) is an example distributed authoring protocol that supports interoperable remote collaborative authoring. It extends the HTTP network protocol to provide facilities for concurrency control – to prevent overwrite conflicts through locking, namespace operations – to copy and move Web resources and hierarchies, and property management – to create, remove and query information about Web pages. Users can collaboratively author their contents directly to an HTTP server through the WEBDAV protocol. This enhanced network protocol augments the Web functionality from a read-only mode for downloading information to a writeable collaborative medium.

This section has described the Web architecture and discussed its limitations as a platform for constructing collaborative applications. The Web does not meet some important requirements for collaborative work; for example, it does not provide equal access to the shared artefacts and it does not have any inbuilt awareness mechanisms. Also, there are not many possibilities for having different architectural arrangements on the Web to optimise

feedback delays. However, with the rapidly evolving Web technologies, some of which have been outlined above, the traditional role of the Web as a passive information repository can in fact be transformed to an active tool for cooperation and for developing CSCW applications.

5.2 Analytic focus

The rest of this chapter will now present an analysis of the various architectural options for developing collaborative applications on the Web. The choice of a particular architectural arrangement directly influences the temporal behaviour of an application. But the temporal interface behaviour is only of importance to the user when it becomes apparent to the user. So, a study of behavioural issues can enable us to determine – *why* an architectural solution is better than another.

For many years, temporal issues in interface design have been largely ignored, with the exception a few studies (Dix, 1987), (Dix, 1992a), (Dix, 1994a), (Gray et al., 1994). However, the importance of time and delays has become more widely recognised with the ever-growing use of the Internet (Johnson and Gray, 1995). The impact of delays on user interaction was illustrated in Chapter 2 (Section 2.3) by using the Web as an example

Chapter 3 examined various architectural models for single-user systems (Section 3.2). Software architecture is about dividing systems into components to perform certain functionalities – *what* the system can do. In order to work as a complete system, the components must be linked together in such a way that they can communicate effectively with each other. While all the components are running as part of the same program on the same machine, these communications are easy and acceptable response times can be achieved.

But the overview on multi-user systems in Chapter 4, showed that when such a system is distributed over a network, as is the case with many cooperative systems, components placed at different locations face higher communication costs and delays than those at the same location. Hence the choice of location – *where* the components are placed – has a significant effect on performance. The major impact of location decisions is on the pace of interaction, which subsequently affects the temporal properties of the interface such as the rate of feedback and feedthrough (Section 2.4.2).

In many distributed systems, data can be moved to improve interactive performance. Furthermore, on the Web, Java applets allow code to move and execute on user's own machines. Thus the placement decisions for the Web are not just about what is placed where, but also about *when* the data and code is at a particular location.

The following sections will now consider each of the *why, what, where, when* aspects in turn and examine all the surrounding issues: *why* – examines the behavioural issues (Section 5.3), *what* – considers the architectural components (Section 5.4), *where* – investigates the placement options (Section 5.5), and finally *when* – explores the issues surrounding code and data mobility (Section 5.6).

5.3 Why – behavioural issues

The reasons for determining *why* a particular arrangement should be chosen influence the behaviour of an application. The behavioural aspect affects the way users view the display on the screen (presentation) and depends on the architecture. The most significant behavioural implication enforced by architectural decisions is often the temporal impact. For instance, if one ignores the temporal issues then from the behavioural viewpoint, the location of the data is not important. However, for performance reasons, it is crucial that there is no perceived lag between any updates to the data and the subsequent changes being reflected on the users' display. Consequently, this may influence the selection of, for example, a centralised or a replicated architecture (Section 4.2).

The rest of this section describes the major behavioural issues that arise within Web-based collaborative work. The requirements for multi-user collaborative interfaces described in Chapter 4 (Section 4.1) will be revisited here and augmented to highlight new issues that emerge with the Web.

5.3.1 Feedback

Feedback is a common feature of direct manipulation interfaces, where objects change their behaviour when users manipulate them. Within the Web, the feedback loop involves transmission over a network. Significant network delays will therefore generate unacceptable feedback response times.

5.3.2 Feedthrough

By their very nature, cooperative work introduces delays as users having to wait for their own feedback and others feedthrough. With the Web, there are further delays and lags that are implicit in the network. The provision of rapid feedthrough therefore becomes more problematic. Current Web-based collaborative applications often provide little support for feedthrough although it is essential for maintaining fluid collaboration.

5.3.3 Awareness

One of the main difficulties in maintaining awareness on the Web is that it is not always easy to find out how changes happen especially when the communication is taking place asynchronously. Some traditional groupware systems with shared workspaces usually record who has made the updates and when they were made. But such temporal information is hard to reconstruct at a distributed level. Even synchronous interaction will pose a similar problem in the event of delays over the network. Furthermore, unpredictable timing delays on the Web, as a result of remote site failures or network bottlenecks, may in the worse case lead to a complete breakdown in the work process.

In order to enhance group work, users may require an additional form of awareness to those identified in Chapter 4 (Section 4.1.4) – an awareness of the state of the communication channel.

5.3.4 Shared objects

The coordination of cooperative work can be mediated via shared objects. Although this form of coordination is less explicit than direct communication, it does play an important role. Indeed, in many cooperative processes there may be little direct communication. Instead, coordination is mainly achieved by communicating implicitly through the artefact (Dix, 1994b).

The studies of interaction referred in the Appendix showed the importance of triggers. Like environmental cues, which were found to be crucial in reminding users of their ongoing activities, triggers could also be associated with shared objects in an electronic cooperative environment to remind users that some actions have been carried out by others and/or some further actions need to be taken.

5.3.5 Control

Due to the common focus on work, collaborative participants have to access the same data. There are potential conflicts that arise when group users are allowed concurrent access and simultaneous updates. Therefore some form of control is required to manage the shared data and the shared objects.

This will determine the nature of the cooperation dealing with issues such as who can update what, where and when; who can see the changes and whether the changes can be noticed in a reasonable amount of time. One of the most common control mechanisms is locking. Other forms of control include access rights, roles or social protocols (for more details, see Section 4.1.6).

5.4 What – architectural components

One of the main functions of cooperative architectures is the presentation and manipulation of shared information by a community of users. Chapter 3 and Chapter 4 emphasised the need for separating the application semantics from the user interface for single-user and multi-user applications respectively. With collaborative interfaces, it is necessary to identify which elements are shared between participants and which elements are different for each participant. This logical separation is also essential when deciding where elements are placed in a networked environment (this will be discussed further in Section 5.6). This section will now explore *what* architectural issues should be taken into account when developing collaborative applications for the Web.

5.4.1 Presentation

As discussed in Chapter 4 (Sections 5.15 and 5.3.2), the presentation component in collaborative systems must support alternative representations of the users display. Shared information can be presented as a single view to all the participants (WYSIWIS) or different users can receive different views. For example, a user may view some shared data in tabular form whilst another may view the same data as a graph. Similarly, group members

can also have their own private views or they can share views of the display. Some systems allow users to shift between a tightly coupled mode that supports a shared view to a loosely coupled mode, where users can view and scroll independently. In cases where the presentation or view is shared, there must be some component of the system that manages the shared information.

5.4.2 Shared data

The key element in any collaborative system is the shared application data. In the Seeheim architectural model (Section 3.2.1), the application interface model component manages the mapping between the application data and the rest of the user interface. This suggests that the visualisation of information requires both the raw data and the semantics of the data, which is usually embedded in the code in a computational setting. On the Web, this aspect is often embedded in CGI scripts, which communicate with the user interface component (Web browser) using Web pages and forms (dialogue level information). However, Java applets have opened up the possibility of including far more of the application semantics at the user interface itself.

5.4.3 Control

Section 5.3.5 highlighted the need for control mechanisms to avoid conflicts and maintain consistency. However, behavioural level control itself has to be driven by some lower level control that has to be maintained by the architecture, the most common mechanism for this being locking. Like the dialogue component in single-user applications, the control component determines the possible order of actions by different participants.

Because data is shared in collaborative applications, there is a clear distinction between the mechanisms for enabling distribution and sharing (e.g. ability to move an object) and the policies for managing those mechanisms (e.g. decisions about when and where the object should be moved to). Effective groupware systems therefore need separate low-level architectural control mechanisms to support those higher-level behavioural control policies. Architectural level control can either be centralised or peer-to-peer in nature and may be supported by a separate server or be part of the shared data infrastructure.

5.4.4 Notification

Group users usually operate simultaneously on the shared data; some users may view part of the data while others may perform changes. If the users views and the underlying data become inconsistent, feedthrough will be lost and users will cease to have a common focus on the collaborative activity.

Similar issues arise in single-user interfaces, in cases where there are multiple views of the same underlying object. Because there is ultimately a single locus of control (the user), consistency is easily handled within the dialogue control component. For example, the PAC architectural model has a hierarchy of PAC agents within the dialogue controller that manage consistency between the views (Section 3.2.4). However, it is more complex to

maintain this level of consistency in a distributed collaborative setting due to the multiple loci of control. This problem can be addressed by using a suitable notification mechanism.

A low-level notification mechanism can therefore inform the presentation component of the various changes to the data so the updates can be replicated on the users display, thus promoting feedthrough and awareness.

5.5 Where – placement decisions

In order to provide rapid semantic feedback in single-user applications, aspects of the presentation can easily leak into the application semantics as all the components are held on the same machine. But when the software is no longer running on a single machine in a distributed environment, one can no longer fudge the boundary and communications between the application and the user interface components as they are enshrined in the physical location and network connectivity. As a result, the issue of *where* the components reside is decisive in order to achieve rapid feedback.

In the Seeheim architectural model, the fast-switch is used as an optimisation feature to allow the application to communicate directly with the presentation and thus bypass the dialogue component (Section 3.2.1). In principle, all feedback could be routed through the dialogue component with more or less translation and interpretation on the way. But, in so doing, the dialogue component introduces a computational delay between the application and the presentation, thus reducing the pace of feedback. Arguably, this is not a problem for current single-user single-machine systems as they can easily perform several levels of processing and still achieve acceptable interactive response.

However, in collaborative systems, the shared data is likely to be stored remotely from the user's workstation. So, instead of a computational delay there will be a network delay. Feedback delays are bound to occur and consequently affect the rate of feedthrough. Unfortunately, one cannot simply add an extra component like the fast-switch, as it too would have to sit remote from the data or remote from the interface. Computational components can be bypassed, but not space!

One can either accept that semantic feedback will be delayed or adopt a paradigm of mediated interaction, which offers instant local feedback to show that the user's action has been recognised and subsequent semantic feedback when the effect has occurred remotely (Dix, 1995a). However, the latter solution will not be acceptable in situations where users demand direct manipulation interfaces.

5.5.1 Replication and Caching

Most solutions that aim at providing rapid feedback and increasing the availability of data involve some form of replication or caching. The objective is to bring the shared data closer to the users.

Caches are merely temporary repositories, which hold an ephemeral copy of the data at any instance in time. Each user interacts with local copies of the shared data on their workstation (figure 5.2a). Because the actual shared data is stored in a central repository, consistency can be easily maintained. Caching is widely used in the design of computer systems such as microprocessors to access recently used data. Similarly, a Web cache stores recently accessed information by users.

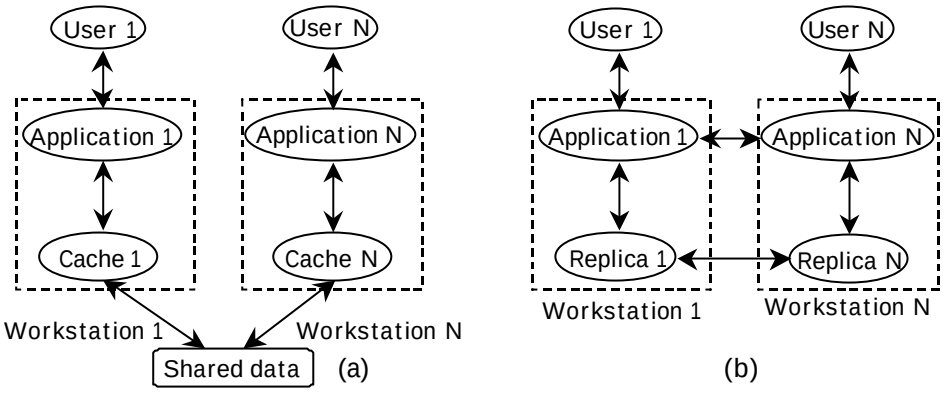


Figure 5.2 (a) Caching and (b) Replication

A Web cache is basically a dedicated computer system that monitors, retrieves and stores Web object requests. So, when users request the same objects or Web sites, the local cache sends out information to them. Cached objects eliminate the need for multiple hops on the Internet route, thus reducing the delay in the service and improving the response time. The higher the frequency of users requesting the same site, the more effective the cache is.

Replicas, on the other hand, are valid full copies of the real data that are stored locally, thus they are more persistent than caches. However, it is more difficult to maintain data and interface consistency, as replicas have to communicate between each other on a peer-to-peer basis. Replicas are synchronised by sending user input from each workstation to each replica (figure 5.2b). Multiple points of updates may lead to race condition and potential data inconsistency. For example, if a user deletes a selected object in a WYSIWIS group drawing program while another user is changing the selection to a different object, inconsistent interfaces may occur if the events arrive at each workstation in a different order.

The traditional approach to replication in distributed systems has been transparency. Race conditions are avoided by maintaining consistency across the different copies of the data through complex synchronisation algorithms. If inconsistencies still persist, a possible solution is to rollback the replica(s) and re-execute the events in temporal order (Satyanarayanan et al., 1990). However, this policy is unacceptable in collaborative interfaces because the display screens would already have been updated. Consequently, alternative solutions based on transforming updates to prevent rollback have been developed (Bentley et al., 1994).

5.5.2 Control

When rapid feedback is not the major concern, concurrency control mechanisms, such as locking or floor control can be applied to prevent race conditions altogether or at least tolerate them (Dix et al., 1993). Real-time synchronous updates may however demand special-purpose algorithms.

Any control mechanism requires meta-data, for example, to record who has the lock on which object. The meta-data itself has similar issues as the real data. It can either be maintained in a replicated fashion by using complex distributed algorithms, or more commonly maintained using a central server. When the data is stored centrally, the same server may deal with both the data and the meta-data, as is the case in traditional databases. However, a separate locking server can also be used. For example, the UNIX file system has no in-built locking mechanism; instead applications request locks on remotely stored files from a special process, the lock daemon.

In situations where off-the-shelf locking is not available or where the locking supplied is unsuitable, application developers are forced to use their own ad hoc locking mechanisms. This is usually the case with Web-based cooperative applications.

5.5.3 Notification

Delayed feedback causes problems, but delayed feedthrough is even more problematic in a distributed environment. No amount of careful placement of components can change the fact that the user making a change is a long way from other users who see the effects of that change. Although a lower pace of feedthrough is more acceptable than the feedback rate (see Section 4.1.3), what is not acceptable is the fact that changes made by a user are never reflected on other users' interfaces or only do so after a long delay. Notification mechanisms are therefore necessary to provide timely feedthrough, as discussed in Section 5.4.4.

The question of meta-data will arise here too independent of the notification strategy adopted. Information such as what objects are being managed and who wants to know about which object will be associated with this meta-data. Again, this can be stored in either a centralised or a replicated fashion.

5.5.4 Different kinds of remoteness

When remote data is accessed in a single-user system by using traditional client-server techniques, the distinction between local and remote is clear. However, in a cooperative application, the difference gets blurred because users have their own interpretation about what is local and what is remote. A user's own machine can be considered to be local while data stored or updated on another user's machine may be regarded as being remote. So, if semantic feedback relies on the data held at another user's machine, the feedback delays will be as long as if the data was held centrally, perhaps longer as central servers may provide a better response.

The situation gets even more complicated when using the Web as an infrastructure. Each user may be accessing several Web servers as well as other central servers such as databases. To a certain extent, the Web makes the physical location of the data unimportant, except insofar as the location affects the response time. However, the physical location is very important, especially when using Java applets. The security mechanisms of Java only allow the applet to access Internet services lodged on the same machine as the Web server that supplied the applet.

There are in fact four kinds of 'remote' application for the Web:

(a) another user's client

(b) the Web server for the current page

(c) a different server on the same machine as the current Web server

(d) a server on a different machine

The placement decisions for the Web therefore do not stop at local versus remote, or even client versus server. The decision about *where* server software is placed is intimately related to the techniques used to implement client software.

5.6 When – moving information and code

Early work on UIMS regards the functional component as being the semantics of the data. However, what gives data any meaning is usually embedded in the code. Data becomes information when it gets interpreted. Hence, the existence and location of the code is equally important. It is common practice to move or copy data dynamically in a networked environment to improve performance. Also, some distributed infrastructures support the migration of objects or code between machines. This section will explore the various mobility aspects of data and code individually, in preparation for their combined interaction in Section 5.8.

5.6.1 Moving data

When caching is used, the 'golden' copy of the data is stored remotely, but a copy is made locally to speed feedback. Because the data can be copied over networks in distributed collaborative applications, this implies that the place where shared data is permanently stored is not necessarily the same place as it (or a copy of it) is used. Using this simple local/remote distinction, the permanent storage place and the place of use can be classified to give the matrix in figure 5.3.

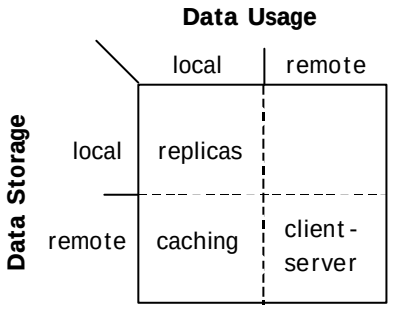


Figure 5.3 Data Usage vs. Data Storage

The matrix clearly shows the distinction between caching and replication. In caching, the 'real' data is central and stored remotely, while the local copy of the data is ephemeral and used locally. In replication, the local data is more persistent and is both stored and used locally. In the case of traditional client-server interfaces, the data is held and used remotely and only the information required to generate the interface presentation of the data is transmitted to the user's local machine.

Notice the empty location in the above matrix. In a groupware context, it is highly unlikely to have a scenario where the data is held locally and yet is used or processed remotely. But such a situation does exist for non-groupware solutions, for example in super computers.

5.6.2 Moving code

In a collaborative distributed interface, it is also important to decide where the code for the different architectural components resides and where it gets executed. The location of code execution influences the feedback rate and thus determines the system's efficiency. The location of code storage instead affects the rate at which changes to the code occur and the ease of distributing those changes, which is a form of feedthrough.

Code execution and code storage are therefore key architectural options. By using a similar matrix as that for data (figure 5.3), the code options can be classified to produce the matrix for Web-based systems in figure 5.4.

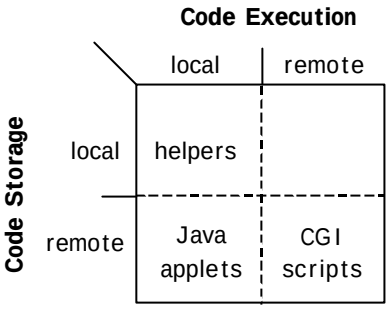


Figure 5.4 Code Usage vs. Code Storage

For instance, in CGI scripts, the code is stored remotely whereas in helpers, it is stored locally. But with both CGI scripts and helpers, the code executes in the same place, as it is stored. Java applets instead allow remotely stored code on the server to be downloaded and executed locally on the client browser at run-time. The browser handles all computation locally, thus avoiding a server-end overload unlike CGI scripts (Section 5.1.1). This form of migration can also be found in many object-based distributed systems.

As with the data matrix, figure 5.4 has an empty location. The Web does not actually cater for locally stored code to execute at the server end and it seems an unlikely option for groupware systems in general. However, in some client-server database applications, some fairly complex SQL queries can be sent to the server, which may be regarded as a form of locally stored code with SQL queries being executed remotely.

5.7 Narrowing down options for the Web

Figure 5.3 showed that shared data could be stored and used either locally or remotely. Similarly, figure 5.4 showed that code could be stored and executed locally or remotely. So, for each component of a collaborative application, we need to decide *where*, in the respective matrices, the code and data for that component reside.

At first, it looks as though there are 16 different architectural options to consider for every component, as there are 4 possibilities for both code and data. But in fact, for general distributed collaborative applications and in particular for the Web, the potential

architectural options can be narrowed down further. The matrices in figures 5.3 and 5.4 had an empty location, which seems an unlikely option for any collaborative application. Consequently, there are only 3 real possibilities for code and data and at most 3×3=9 combinations.

However, if the combination of code and data is taken into account, then the possibilities reduce further. Although data and code can be stored in different places, the code must execute where the data is used. The data and code matrix must therefore 'agree' in the location of execution and use (figure 5.5). Consequently, there is only one possibility for remote execution/use and 4 possibilities (2x2) for local execution/use. Each option will now be considered in turn.

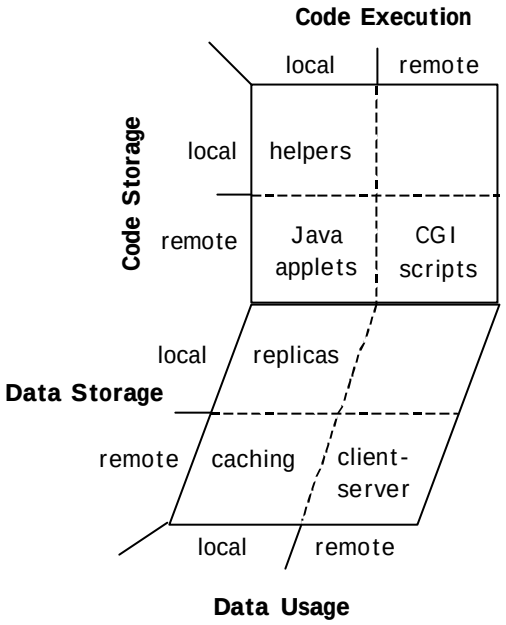


Figure 5.5 Linked matrices

5.7.1 Remote execution and use

The only possibility for remote execution and use in a collaborative application is where both data and code are stored, used and executed remotely (although each could conceivably be stored at different remote sites and only come together for execution/use).

A component of this kind can be implemented in two ways. It may be a traditional transaction based client-server application that uses CGI scripts to process transactions centrally. In fact, many Web-based repositories are of this form, like BSCW (Bentley et al., 1996) for instance. Alternatively, it may be achieved by using a specialised central server, as is the case with most chat-based Web applications (Welie and Eliëns, 1996). It should be noted that these two implementation options differ principally in the pace of cooperative interaction they enable.

5.7.2 Local execution and use

There are 4 such options for code and data from figure 5.5:

(a) code local – data local

(b) code local – data remote

(c) code remote – data local

(d) code remote – data remote

Both options (a) and (b) are of the form of a helper or stand-alone application that uses caching or replication to handle the shared data. Given the limited ability of most Web servers in allowing documents to be uploaded, it is likely that option (b) will use a non Web-based database or bespoke server. However in both cases, the Web may act as a way of locating shared resources and initiating a specialised collaborative application, without being intrinsic to the running application.

Options (c) and (d) principally involve code in the form of Java applets (although other forms of downloaded scripts are available). But the security limitations of Java does not allow applets to access files stored on users' local machine. Also, Java applets can only connect to a server that is on the same machine as the Web server they were downloaded from. Consequently, Java applets cannot operate in mode (c) with permanent locally stored data and they cannot enter into peer-to-peer communication (except by using a central switchboard server). This eliminates the option of having Java applets operating with locally held replicas.

All feedthrough must therefore be through a central server at the same site as the Web server. This effectively leaves only option (d) – Java applets with caching, as a truly Web-based option and even then, only when using a data repository that is situated at the same location as where the applet is stored.

5.8 Impact on research

The analytic framework discussed in this chapter has raised a number of behavioural, architectural, placement and mobility issues that arise with cooperative systems and the Web. The aim of this research is to develop an architectural framework that exhibits appropriate temporal properties, particularly for collaborative applications that execute on the Web. This section will highlight the main behavioural considerations of this research and examine their influence on the resulting architectural framework.

5.8.1 Behavioural considerations

A number of behavioural issues were explored in Section 5.3, but this research will focus on the provision of feedthrough. Although both feedback and feedthrough are major temporal properties, the demands for feedthrough are more important in cooperative systems for maintaining effective collaboration between group users and promoting awareness. The provision of feedthrough is more challenging in a distributed collaborative environment, as both the issues of pace of interaction between participants and network-related delays have to be taken into account. Besides, there is very little support for feedthrough on the Web.

Collaborative users often work with a large number of shared objects and it may not always be possible to maintain an appropriate rate of feedthrough for each object, even over fast networks. However, the requirements of feedthrough tend to be more flexible than feedback (Section 4.1.3). Because some objects are more significant for obtaining a sense of engagement, the concepts of quality-of-service (Rada, 1995) can be applied to give different levels of feedthrough on shared objects within a groupware architecture. For example, the sharedness of some objects like shared cursors, can be relaxed by reducing their feedthrough, but group pointers, which is a form of virtual finger used during electronic conferencing, need to have almost instantaneous feedthrough to be effective.

5.8.2 Influence on architecture

The rate of feedback and feedthrough provided to the users is driven by decisions taken at the architectural level. The need for rapid feedback points towards the use of some form of caching or replication (Section 5.5.1).

On the Web, rapid user interface feedback can be promoted by running code locally as downloaded Java applets (Section 5.6.2), but the code must execute where the data is used (Section 5.7). This opens up the possibility for the data to migrate locally and local data updates will conflict with the needs of feedthrough. Consequently, there is an important trade-off between feedback and feedthrough that needs to be addressed at the architectural level.

The standard Web protocol offers poor notification besides *server push* for promoting awareness. The *server push* technology was one of the earlier techniques designed to perform continuous update of users' screen. The server continuously runs the application program, which generates dynamic pages and sends new copies of those pages to the browser. Although this mechanism allows constant information update, it causes excessive server overhead and introduces delays. Furthermore, if several browsers attempt to access the pushed pages simultaneously, a separate copy of the dynamic page application program has to be executed for each request, consequently delaying updates further.

A more robust notification mechanism is therefore required to support feedthrough – by informing the users of any changes to the data in a timely fashion, and to enhance awareness – by keeping track of users activities. Notification services generate similar issues as

locking (Section 5.5.2). If no notification service is provided then an ad hoc mechanism is necessary; for instance, individual clients may poll one another for changes. Alternatively, a notification service may be incorporated within the data-management infrastructure; for example, Lotus NSTP (Patterson et al., 1996) offers a generic data storage and notification server. Finally, a stand-alone notification server can be used. The various design alternatives for notification services are explored in details in the next chapter.

5.9 Summary

This chapter first examined the functionalities of the Web as a development platform for collaborative applications. The Web offers a ubiquitous infrastructure and a platform independent interface that can be easily integrated with existing user environments. A brief overview of the Web architecture was presented, followed by a critique on its limitations in supporting cooperative tasks, based on its architecture, existing protocols and browser restrictions. Although the Web is unsuitable for developing systems that require highly interactive user interfaces with a high degree of synchronous interaction, the constraints of its basic architecture can be removed to meet the requirements of cooperative systems.

The focus of this chapter was on the analytic framework for constructing collaborative applications on the Web. This was based upon a systematic investigation of *why* certain behavioural issues are essential for collaborative work, *what* architectural components are necessary, *where* should the components be placed in a distributed architecture and finally, *when* should the components be moved to improve performance.

The analysis of behavioural issues identified the key architectural components of cooperative systems. The placement decisions revealed the conflicting needs of feedback and consistency on the Web. This is commonly dealt with by using either caching or replication to bring the shared data 'closer' to the user. Web applications use dynamically downloaded code of which applets are the most common. This allows both code and data to be stored in a permanent location while having an ephemeral location where they are executed or used. The mobility issues associated with data and code generated a storage/use matrix for data and a storage/execution matrix for code, which facilitated the analysis of placement options.

Although there appears to be many possible combinations of data and code placement, a close examination of their interaction within distributed environments in general and the Web in particular, limits this to only 2 'real' Web options. However, due to the security limitations of Java, applets cannot enter into peer-to-peer communication, thus eliminating the option of having Java applets operating with locally held replicas. This effectively leaves only one truly Web-based option of using Java applets with caching. This option favours rapid feedback, as the real data is located centrally, but it does conflict with the needs of feedthrough.

The behavioural and component analysis in this chapter narrows down the focus of this research on facilitating an important behavioural requirement – feedthrough, which is also a significant temporal property of collaborative work. Feedthrough is crucial for maintaining

collaboration and promoting awareness, but it is an intrinsic limitation in distributed systems in general and even more so in Web-based collaborative applications. Because the requirements for feedthrough challenge the needs for feedback on the Web, a solution to this problem can be found at the architectural level. A suitable notification mechanism is therefore required to manage the rate of feedthrough and optimise on the temporal performance. The next chapter deals with the issues surrounding the design options for notification services.

Chapter 6 Exploring the Design Space for Notification Servers

Feedthrough is an essential feature of cooperative interfaces in general, however there is often little support for it in existing Web-based collaborative applications. The need of feedthrough on the Web does conflict with that of feedback, as discussed in Chapter 5. 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).

Feedthrough is important for three main reasons – firstly, it satisfies a 'functional' purpose by allowing users see an up-to-date version of the work; secondly, it facilitates 'coordination' by preventing inconsistent updates and finally, it supports the general 'awareness' of other people at work. The first two concerns have led to some considerable work on algorithms for synchronous editing and for merging versions of asynchronously edited material. The last concern has instead always remained an informal interest in CSCW, although it has been augmented by some formal analysis of 'awareness' models (Benford et al., 1993), (Rodden, 1996). This modelling approach has arisen largely out of work on virtual collaborative environments, where the main objects of interest are the virtual locations and actions of the participants themselves, instead of documents or shared drawings.

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 key requirements: the ability to access and update shared data, and knowing when that data has been updated. The former lies behind the design of shared data repositories, either bespoke systems designed for CSCW (Bentley, 1994), (Hill et al., 1994) or off-the-shelf databases and shared object stores. The latter requires notification mechanisms.

A notification server is basically a piece of software whose task is to relay the fact that changes in data or other events have occurred. There are numerous ways in which notification services can be managed in a collaborative system (Patterson et al., 1996), (Hall et al., 1996), (Fitzpatrick et al., 1999). The aim of this chapter is to explore and clarify the design space for notification servers. The ultimate purpose of a notification server is to provide effective user-level behaviour. Issues in this chapter have been discussed in (Ramduny et al., 1998).

Section 6.1 assesses the need for notification mechanisms as a means of propagating updates. Section 6.2 gives a description of Status–Event analysis, an analytic framework developed to tackle various user interface issues (Dix, 1991), (Abowd and Dix, 1994), (Dix and Abowd, 1996a). Status–Event analysis is used here as the basis for analysing issues surrounding notification servers. The concepts of Status–Event analysis are applied in Section 6.3 to examine the ways in which an agent in a system can become aware of a

status change. This is then employed in Section 6.4 to explore the ways in which a notification server can become aware of changes in the shared data and how it in turn, makes this available to client applications. Section 6.5 presents a taxonomy of the design space for notification servers. Section 6.6 considers the use of layering between the client, notification server and user to achieve the desired pace of interaction. Finally, Section 6.7 briefly looks at some underlying notification models that have been adopted in example systems.

6.1 Need for notification mechanism

Notification mechanisms are necessary for informing users *when* the data has been updated. Even if the data is stored and accessed rapidly from a central location, it is ineffective unless the client programs know when the data 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 notification mechanisms, users may eventually see the changes that have occurred, but at a time-scale and pace that may not acceptable for the task at hand.

Each application that updates shared data can in fact be responsible for notification and consequently broadcast to all the interested parties that the change has happened. However, as with peer-to-peer methods for data replication (Section 4.2.2) this has a high overhead, both in terms of the algorithm complexity and network load. For example, each participating client program should know about all other clients in order to broadcast change information to them. Furthermore, the changes must be kept up-to-date as users join and leave the system. The overhead involved in having the application itself manage the updates is one of the core motivations into notification (or awareness) services that provide a set of standard techniques for notifying changes.

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 be either 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. The various design options for notification services will now be analysed by using Status-Event analysis.

6.2 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, event-based models fit less well when dealing with shared data in collaborative systems. The nature of shared data is that it persists – it does not just happen at a particular moment; instead 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.

Status-Event analysis was developed to deal with such phenomena (Abowd and Dix, 1994), (Dix and Abowd, 1996a). 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.

Status–Event analysis has been applied in several contexts – from the analysis of issues in fine-grained interaction (Dix et al., 1993) and auditory interfaces (Brewster, 1994), (Brewster et al., 1994), Dix, 1994 #153] to the specification of the complex behaviour of shared scrollbars in collaborative applications (Abowd and Dix, 1994), (Dix and Abowd, 1996a). Status–Event analysis has also been used in the understanding of delays in user interfaces and collaborative systems (Dix and Abowd, 1996b).

Issues surrounding status, events and agents were considered briefly in Chapter 2 (Section 2.4.1) while examining the temporal properties of interactive systems. The study of delay was centred on the idea of mediation (Section 2.4.1.2) and this will also be the key to an architectural understanding of notification mechanisms.

6.2.1 Key concepts

The two central concepts in Status–Event analysis are obviously events, which occur at particular moments (such as mouse clicks, beep, 6 o'clock) and status, which always have a value (shown by mouse position, screen, position of hands on the clock). In addition, agents (human or computational) respond to events which subsequently modify the status.

A key feature of Status–Event analysis for a particular agent is the difference between the *actual event* – some objective thing that occurs and the *perceived event* – when an agent notices that an event has occurred. 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. Likewise, with notification servers, although events occur at certain places, there may be a substantial delay before those changes are perceived. The same behaviour arises in human-human interactions, human-computer interactions and in interactions within computer or mechanical systems.

6.2.2 Mediation

The most important aspect of Status-Event analysis for analysing the role of notification servers is that of mediation. This is when some desired behaviour is achieved by interposing some additional agent or status entity. For example, in an electronic mail system, the receipt of a mail (an event) is communicated to the user by a change of the screen icon (a status) (Dix et al., 1993). The use of a status to mediate communication between agents is very common, as is the nature of shared data.

A second form of mediation occurs when a status–status relationship needs to be maintained (Dix and Abowd, 1996a). For instance, when dragging a window across the screen with a mouse, the window must keep track of the mouse position. In the real world,

status-status relationships may be a result of physical properties, for example when one end of a string is pulled the other end moves. However, in computer systems, these relationships are typically maintained by a mediating agent, which monitors the first status and thereby alters the second accordingly.

In both the case of agents communicating via status or an agent mediating a status–status relationship, it is vital to determine *how* an agent becomes aware of a status change.

6.3 Status change discovery

Consider the scenario where there is a status S and some agent A and an actual event occurs which subsequently changes the value of status S. Figure 6.1 shows the alternative interactions by which agent A can become aware of the status change.

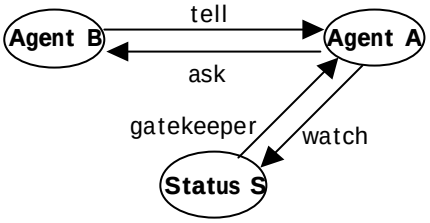


Figure 6.1 Status-agent interaction

The process that allows a change to become a perceived event for agent A does not lie in the flow of data that informs agent A of the new value of the status. Instead, it lies on the perceived event for agent A that a change has occurred. This can happen in four different ways.

6.3.1 Case 1: watch

Agent A can watch the status S. In a physical system ‘watching’ is usually a continuous activity, focussing on a specific status such as watching a pot and waiting for it to boil. But in a discrete system, 'watching' means periodically *polling* the status. Another question now arises: what event prompts the polling? This leads to three 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[8], 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.

6.3.2 Case 2: tell

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 the changes that effect them.

Here too, we may 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 in cases 1 – 4

6.3.3 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 does agent B know – leading to subcases as in case 2.

Perhaps the most notable example of this category is the logon process for a 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.

6.3.4 Case 4: gatekeeper

The status S is in some way active or is closely bound to an agent that '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

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

objects (Trevor et al., 1994). This paradigm is also applied in Suite (Dewan, 1990) and other constraint based toolkits.

The gatekeeper case can be seen to be a special instance of either case 1 or 2. If the agent is regarded as being part of the status, then it behaves as a subcase of case 1, for instance like an alarm going off. Alternatively, the gatekeeper may be seen as an agent in its own right, in which case, it can be regarded as a third subcase of case 2.

6.3.5 Source vs. Initiative

The source of an interaction can be either an agent or the status itself. Although the actual information resides in the status, the source holds the knowledge of any changes to the data. Initiative plays a key role in determining how changes of status are discovered. When the status is modified, it is obvious that an agent is responsible for initiating the communication. But when the agent is affected, then it is important to know whether the agent itself initiated the change or whether some other party chose to do so.

Cases 1 – 4 for status change discovery can be mapped onto a source versus initiative matrix (figure 6.2). Both cases 2 and 3 involve a second party agent as the source of the interaction, but the difference between them is one of initiative. In case 2, it is the second party agent B that takes the initiative to find out about the status change, while in case 3, the responsibility lies with 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.

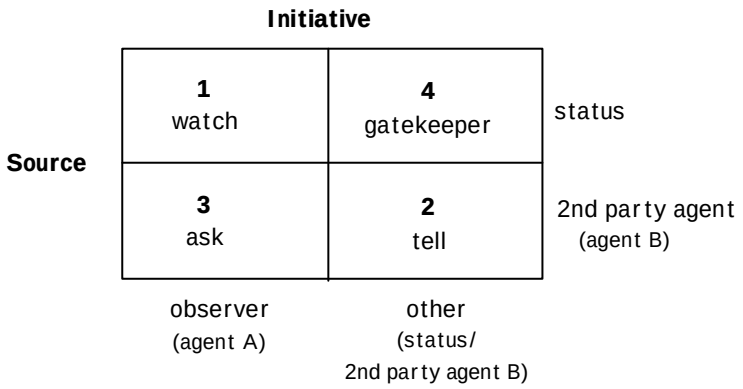


Figure 6.2 Source v/s Initiative

Given this arrangement, we may then ask ourselves what prompts one to take the initiative. For example, in case 1, this leads to the subcases, such as:

(a) it may be internal (most likely time driven), or

(b) it may be due to a third party agent (demand driven), or

(c) it may be spontaneous (either 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. A similar analysis for the cases of status change discovery is applied in the following section to notification servers in collaborative applications.

6.4 Notification Servers as Mediators

For the sake of this discussion, let us assume that there is a centralised architecture with a 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 its screen accordingly?

Although the client applications are likely to be identical on both workstations, they do 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, the focus is on the dients themselves and not on the users who will ultimately interact through them. Hence, the emphasis is on how the events about the changes to the information are propagated rather than how the information is displayed.

The options that enable the passive client to discover changes from the active client will now be explored in two scenarios:

(a) when the notification server is absent and

(b) when the notification is added.

6.4.1 Change discovery options without a Notification Server

The agents of interest are the active client and passive client, and the status is the shared data (figure 6.3).

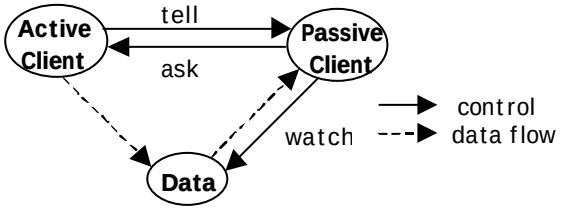


Figure 6.3 Client-data interaction without notification server

In section 6.3, four cases of discovering status change were identified. 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 change discovery without a notification server should only involve the cases where the agent is the originator of the status change, namely, the active client.

The passive client can thus discover changes in the shared data in one of the following ways:

- passive client watches or 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 (Benford et al., 1994a).

- passive client asks the active client (case 3)

This situation is less common but normally occurs in systems where there is a designated master version of an application that is responsible for managing the distribution of updates. This approach was adopted in early versions of shared screen systems.

6.4.2 Change discovery options with a Notification Server

When a notification server (NS) is introduced, it acts as an intermediary between the active client and the passive clients. This offers the benefits of managing the process and allows 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 ease of propagation offered from a central awareness service (Section 4.2.3).

In most cases, both clients communicate with the notification server in various ways. Essentially, the active client informs the notification server of the update whilst the passive client seeks to be informed of those 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 6.4 shows the potential control flows between the clients and the notification server that allow 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. The combination of control

flows that may occur will later produce the taxonomy of the design space for notification servers in Section 6.5.

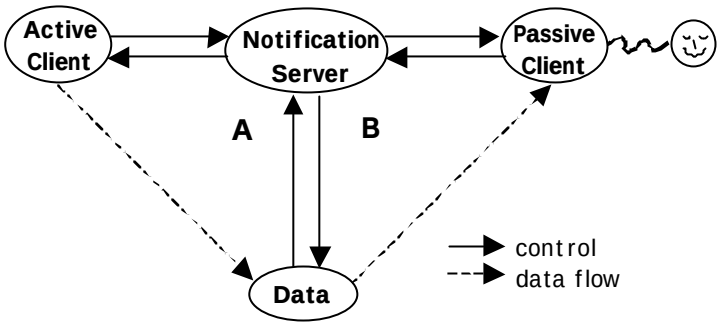


Figure 6.4 Client-data interaction with notification server

Let us first consider the interaction between the notification server and the active client (figure 6.5). By applying the options discussed in status change discovery (Section 6.3), the notification server is able to find out the stages of any change in one of the ways listed below.

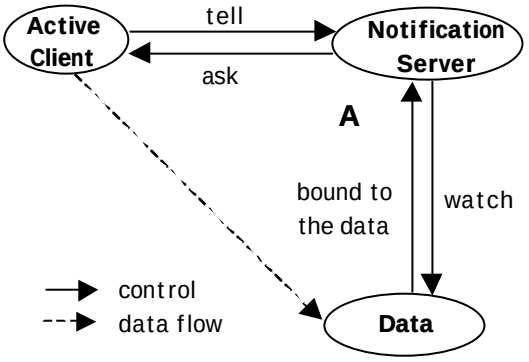


Figure 6.5 Notification server communicating with active client and data

Note, all options starts with label 'A' to differentiate from the interaction between the notification server and the passive client, which falls under label 'B', as will be seen later. Also, the same classification is observed as in cases 1 – 4 (Section 6.3) for example, '1' means watching or polling, '2' implies telling and so on.

- 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 (Lauwers and Lantz, 1990).

- notification server asks the active client (A3)

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

- notification server is bound to the data (A4)

The Rendezvous system (Hill et al., 1994) adopts such an arrangement as it separates the abstract view from the data and uses some coupling mechanism to manage updates based on an encoding of constraints (Section 4.3.4).

Once the changes in the shared data have become a perceived event for the notification server, the latter must then relay that event to the passive client (figure 6.6). 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 be either of a direct or an indirect nature.

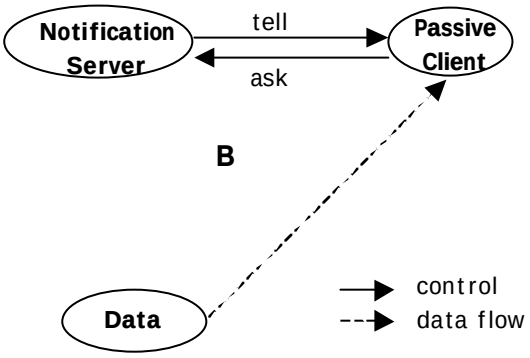


Figure 6.6 Notification server relaying change to passive client

The options for change propagation from the notification server to the passive client are:

- notification server tells the passive client (B2)

This arrangement is used by notification servers that are linked to window systems where events can be sent directly to the client. The development of the push technology on the Web also allows this form of change propagation.

- passive client asks the notification server (B3)

This is the classic arrangement used in Web-based awareness mechanisms such as WAP (Palfreyman and Rodden, 1996).

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 (options B1, B4) are ruled out.

6.5 Taxonomy of notification servers

The previous section showed that the notification server could find out about changes from the active client and the data store in 4 different ways (options A1–A4). Furthermore, the notification server can communicate the updates to the passive client in 2 ways (options B1, B2). These possibilities can be represented in a 4x2 matrix (figure 6.7).

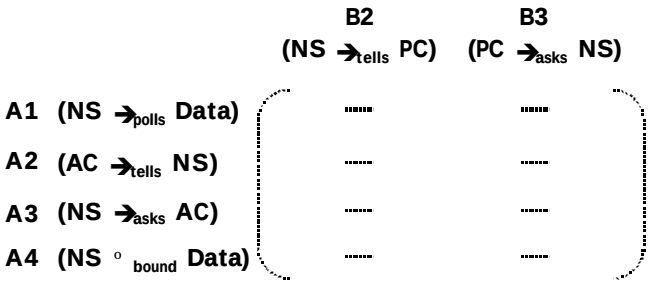


Figure 6.7 4x2 matrix for change discovery and propagation

The matrix in figure 6.7 can be populated by some example systems built using the respective protocol to generate the taxonomy of notification servers (figure 6.8).

(AC-NS) \ (NS-PC)	B2	B3
A1	desktop web crawler / agents	POP server
A2	pure notification server	certain MUDs
A3		
A4	NSTP	WAP awareness protocol

Figure 6.8 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.

6.5.1 Possible arrangements

Each of the arrangements in figure 6.8 will now be discussed in turn.

Case A1-B2: 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 the clients (B2) about the changes, perhaps via an email message.

Case A1-B3: A POP server watches for the data (A1) but it is only 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 be either 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: This can be seen as a 'pure' notification server because the notification server is entirely separate from the data store. The notification server is told (A2) about the changes from the active client and it then notifies (B2) the users' client 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.

Case A2-B3: Certain Web MUDs would fall in this category. 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 the notification server (A2) 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 knock.

Case A4-B2: NSTP (Patterson et al., 1996) supports this arrangement. The notification server is closely coupled to the shared data repository (A4). In the event of any updates, the notification server tells the clients (B2) about the changes in the shared state information.

Case A4-B3: The WAP awareness stateless protocol (Palfreyman and Rodden, 1996) 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 the awareness server (B3).

Case A3-B2 and Case A3-B3: These arrangements represent the empty row A3 in figure 6.8, as they are both ineffective. Case A3–B3 is particularly inefficient because it implies that the passive client would constantly have to ask for changes from the notification server and the latter would then send a request to the active client. However, case A3–B2 where the notification server can be partially stateless is more likely to occur. The notification 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 notification server needs to avoid contention on the demands.

6.5.2 Location of notification server

The discussion so far has assumed the presence of a single notification server and data repository within a centralised architecture. 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 6.9.

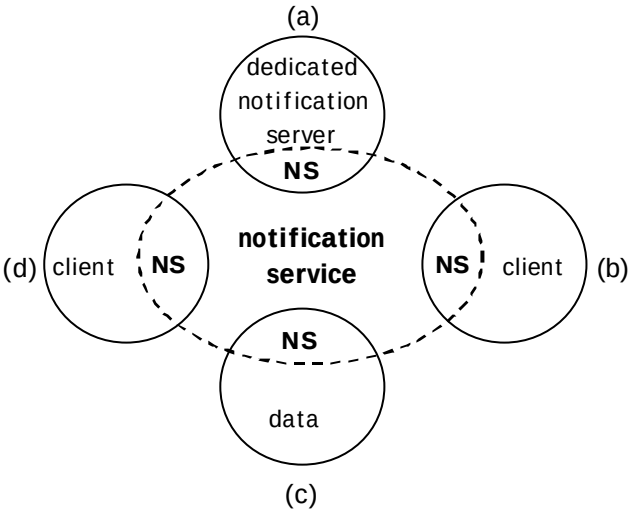


Figure 6.9 Location of notification server

Four major location options can be identified:

(a) the notification server is *closely bound* to the data repository

The notification server resides in the same physical address space as the data store or the data is at least part of the server. An example is a database supporting triggered actions. The notification server does not have to explicitly ask for the changes, the data store can inform the notification server about them.

(b) 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 or sit somewhere else on the network.

(c) a distributed *peer-to-peer* notification service

Often a conceptual notification server is realised as a software abstraction within the clients using peer-to-peer communication. An extreme example is AETHER (Sandor et al., 1997), which percolates awareness information from node to node in a network, thus effectively providing an emergent distributed notification service uniformly throughout the network.

(d) 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.

This section has presented a taxonomy of notification server types and considered some example systems that satisfy the different arrangements of change discovery and change propagation. No system had implemented a 'pure' notification server arrangement when this analysis was carried out (Ramduny et al., 1998). Such an arrangement allows the notification and the data to be separate from each other. This is particularly important on the Web where the protocols that access data are fixed, thus forcing notification to be added at a separate level.

6.6 Notifying users

The motivation of this research lies in the timeliness of information by providing an 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 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 the true push technology (based for example on a Java applet) is compared with a browser refreshing Web pages based on expiry time, then from the user point of view, they may appear to be little different. However, at a lower level, the applet is informed by the 'pushed server' while the browser polls after the expiry date.

6.6.1 Layering

These complex interactions can once again be explained through Status−Event analysis. Status−Event analysis shows 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 uses 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. For example, Lotus Notes does not actively notify users when the database has changed by default. Instead, it puts a mark against the changed notes in a list view. Users only become aware of the changes when they explicitly look at the relevant list view. At the user level, this corresponds to asking (case 3) but at the lower level, the Notes server informs the Notes client when changes occur (case 2) so it can update its local structures.

Alternatively, many email clients periodically poll the server (case 1) 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.

Different layers can therefore be used between client–notification server and client–user to achieve the desired pace of interaction.

6.7 Notification models

A number of systems have been designed to act as notification or awareness servers, some of which have already been considered in the taxonomy of notification servers (Section 6.5). The underlying notification models employed by these systems are usually based on either an event-based approach or a status-oriented approach.

6.7.1 Event-based

NSTP (Patterson et al., 1996) supports an event-based or channel-based approach for notification. Corona (Hall et al., 1996) adopts a similar event-based approach, however it uses a 'publish-subscribe' service to maintain notification. The published notifications are multicast to distributor nodes and they in turn multicast them to other distributors that send them to local subscribers. The Elvin (Fitzpatrick et al., 1999) notification service instead acts as a distributor of events and works on a producer-consumer model. Producers detect events and push them to the notification service and the latter then distributes them to the interested consumers.

6.7.2 Status-oriented

Systems that are expressed in terms of events are not as effective as status-oriented models. Status-oriented models fall very close to the spirit of Status–Event analysis (Dix, 1998).

AETHER (Sandor et al., 1997) is based on a status-oriented approach which percolates awareness information from node to node in a network, thus providing an effective notification service uniformly throughout the network. Formal awareness models (Benford et al., 1993), (Benford and Fahlén, 1993), (Benford et al., 1994b), (Rodden, 1996) are more status-oriented. For example, instead of "when person A enters the room, person B is informed", these models are phrased as follows: "when the nimbus (region of influence) of person A intersects that of person B, they should be aware of one another".

Status-status mappings are also found in single-user interfaces (e.g. for dragging) and multi-user interfaces (e.g. for keeping users views consistent). Hence, toolkits and user interface development systems usually have some form of event notification mechanism to handle user interface events, such as mouse clicks. The Smalltalk MVC model (Section 3.2.3) uses a mechanism whereby objects can register themselves as dependants of another object, so the latter can then inform its dependants about changes to its state (Lewis, 1995).

A similar technique is applied in X-Motif callbacks (OSF, 1995), Java JDK 1.1 source-listener event model (Flannagan, 1997) and active values (Section 4.4.3). However, in callback-style toolkits, the relationships between status phenomena are often coded in terms of event callbacks. A more effective mechanism is to use toolkits that are based on constraints (Section 3.5.3), as they are founded on status-status relationships, which in some ways makes them closer to Status–Event analysis.

The analysis applied in this chapter to investigate the design space for notification servers has been theoretically augmented and formalised in another study (Dix, 1998), which decomposes the process of event propagation and unpacks the relationships between agent–agent, status–agent and status–status. The chain of interactions (causality chain) that lead to an agent or a status to be influenced by a certain event is mapped out to reveal behaviours of event discovery. An interesting observation from that study is the apparent reversal of initiative and causality. Causality is related to the event flow, which manages the flow of control, whereas initiative determines how changes to the status are discovered.

6.8 Summary

The aim of this chapter was to explore and clarify the design space for notification servers to enable a better understanding of the issues involved. A general model of status change discovery taken from Status–Event analysis was applied to notification server architectures.

Status–Event analysis is an analytic framework that includes both aspects of events and status. Status is very useful for describing relationships that have a persistent value through time, like shared data, where event-based models fit less well. Agents usually communicate via the mediating status or they mediate a status-status relationship. This mediation position is central for analysing the role of notification servers.

Four main cases for status change discovery were identified namely, an agent watches the status; an agent is told by a second-party agent; an agent asks the second-party agent and finally, the gatekeeper scenario, where the status is bound to the agent and knows ‘instantly’ when the status is changed. Some of these cases can be divided further into subcases. A similar analysis was then employed to explore the ways in which a notification server (mediator) can become aware of the changes in the shared data (status) and how it in turn makes it available to the clients (agents).

The notification server (NS) acts as an intermediary between the client that performs the change – the active client (AC), and the client that observes the change – the passive client (PC). The notification server is only responsible for mediating control between the clients and not for passing on data. The analysis highlighted the similarities between the communication from AC–NS and from NS–PC. It also emphasised the important distinction between the knowledge of *what* has changed in the shared data and the knowledge *that* it has changed. Furthermore, it generated a taxonomy based on issues of source and initiative within the three-way AC–NS–PC communication.

A conceptually single notification server is not necessarily confined to a specific location within a CSCW system. The various location options for the notification server were considered. The notification can either be packaged within the data or it may sit remotely from the data. Indeed, the notification server can be placed in various physical locations and may even be distributed over several components. In the latter case, it is 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. The different categories of notification that may be seen at different levels were examined. This layering mechanism allows a client application to use non-optimal low-level notification services and yet achieve acceptable user-level behaviour.

Finally, some notification models used in notification or ‘awareness’ servers and toolkits were reviewed. Event-based models tend to be less effective than status-oriented models, which fit closer with Status–Event analysis.

The taxonomy of the design space for notification servers discussed in this chapter has provided a framework and a vocabulary to compare and discuss different notification mechanisms with an aim to inform and improve design. Also, the use of Status–Event analysis as a foundation to this study ensures that the framework for notification architecture does indeed cover the design space.

The provision of feedthrough in a Web-based collaborative application requires a notification server, preferably one that acts as a 'pure' notification server at the architectural level. The 'pure' notification server arrangement allows a separation of concern between notification and data, and it will be used as a design driver for the experimental notification server described in Chapter 8. Prior to this, the next chapter will discuss how the notification server can actually be used to provide effective user-level behaviour.

Chapter 7 Impedance Matching: Coping with Limited Resources

In the real world, the feedthrough between participants is usually mediated by the physical properties of artefacts and space. However, 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. Chapter 6 explored the various design options for notification servers. An important issue in providing effective user-level behaviour lies in the frequency at which the notification server should send updates or feedthrough information to the users. This also depends heavily on the desired pace of interaction. The issues surrounding pace of interaction were dealt with in Chapter 2.

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 and Bellotti, 1992). Whereas goal-directed activity usually requires detailed and timely feedthrough, awareness is typically longer term and more 'fuzzy'. For implementation, the difference between goal-directed feedthrough and awareness are largely about quality of service (QoS) (Rada, 1995). Both goal-directed feedthrough and awareness require some form of underlying notification mechanism. However, the differences in QoS suggest that the notification server should be able to modify the rate and quality of notification to match the required feedthrough at the user interface.

This chapter investigates how collaborative users can be provided with timely updates by controlling the frequency of notification through *impedance matching*. Impedance matching is usually employed in engineering terms to describe the procedure in circuit design for matching unequal source and load impedance to optimise the power that the source delivers to the load. Impedance denotes how much a device resists the flow of an AC signal whereas resistance shows how much a device resists the flow of a DC signal.

The term *impedance matching* is employed in the context of this research more as a metaphor to describe the notion of matching the required and supplied pace of feedthrough to maximise user-level behaviour at the interface. Perhaps the method of adjusting the pace of feedthrough could be simply called ‘matching’. However, previous publications related to this work (Ramduny et al., 1998), (Ramduny, 1999), (Ramduny and Dix, 2002) have already referred to the term impedance matching, hence it has been maintained in the thesis.

Section 7.1 examines the need for impedance matching to control the rate of change propagation between collaborative users. Section 7.2 describes how notification servers as mediators are ideal for supporting impedance matching by controlling the pace of feedthrough. Feedthrough demands can be reduced by subsequently reducing the pace and the volume of updates. The issues surrounding pace and volume impedance matching are discussed in Section 7.3, together with some related implementation issues. Section 7.4 explores the potential triggers for pace impedance by analysing their effects on event

propagation through the use of time-space diagrams. Section 7.5 assesses some scenarios where impedance matching can be introduced to provide a controlled pace of feedthrough and awareness to collaborative users. Finally, Section 7.6 analyses some outstanding issues that arise from impedance matching.

7.1 Need for impedance matching

Let us consider the effect of change propagation within a collaborative environment. In Chapter 6 (Section 6.4), a distinction was made between the client who performs the changes – *Active Client* (AC) and those who view the changes – *Passive Client* (PC). Note, the role that the AC and PC assumes is not permanent, it depends on which client is performing the action at any given point in time.

Figure 7.1 shows a scenario where an active client is propagating updates to a passive client and it in turn, passes the updates to the user. Collaborative work usually involves communicating over a network. Thus, the AC–PC interaction is influenced by the available bandwidth, whereas the user–PC interaction is influenced by the response time in seeing the changes. Obviously, the shorter the response time, the more effective the user-level behaviour is.

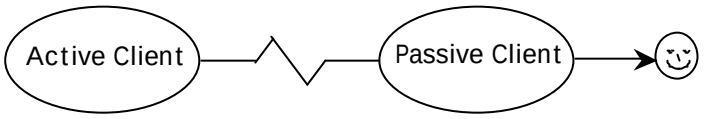


Figure 7.1 Update propagation

Collaborative users often interact with a large number of shared objects. This generates a high volume of updates that need to be broadcast to all the users. The response time with which users see those updates will increase unless the updates are sent rapidly. Usually, there are not enough network and computational resources available to sustain such a high rate of feedthrough. The updates could still be broadcast rapidly, but some degree of throttling is required at the user-end.

Even if the network was infinitely fast and there was an infinite amount of memory, a maximum rate of feedthrough for all the objects will generate further network congestion. The extra computational load would undoubtedly mean delays for all the objects, including the ones that are most salient and important. Furthermore, from a cognitive point of view, users should not be overloaded with too much information, as it is annoying and it usually results in a poor user interface (Section 2.2.1). Users may thus find it too distracting to cope with a very fast rate of feedthrough.

The rate of feedthrough should therefore be reduced to such an extent that the updates are broadcast at a fast enough rate and yet be acceptable to the users. Consequently, the passive client need not forward the updates to the users at the same rate that it receives them from the active client. The passive client could still accept changes from the active

client at a fast rate but it can forward the updates to the users less often to make the pace of feedthrough more acceptable.

Furthermore, the pace of delivering feedthrough is crucial and depends on the type of the task. A study (Pausch, 1991) found that rapid feedback of low fidelity wireframe models was far better than slower photorealistic rendering. Delivering feedthrough at the wrong pace can therefore be problematic. For instance in figure 7.1, if the rate of updates generated by the AC is too high and the rate at which the PC informs the user is too slow, users may act without having an up-to-date knowledge of one another's actions. Similarly, if the rate of updates generated by the AC is too low and the rate at which the PC informs the user is too high, users may be easily distracted by irrelevant changes.

Clearly, there is a need for some form of matching between the active client and the passive clients in order to obtain the right pace of feedthrough. This matching of the required and supplied pace of update events is called *impedance matching* (Ramduny et al., 1998), (Ramduny and Dix, 2002). A mismatch in the required and supplied pace of feedthrough will inevitably affect user-level behaviour. Impedance matching is therefore essential for delivering feedthrough that is both effective for the user and efficient for the system. The next section will justify the choice of placing impedance matching within the notification server.

7.2 Where to control pace of feedthrough

The following discussion assumes that each user is interacting through a single client device and for any update or user action, the active client is the client of the user who initiated the action and the passive clients are the clients of the rest of the users who receive feedthrough.

7.2.1 Interaction without notification server

In the absence of a notification server, the active client is responsible for propagating the changes to the shared objects to the passive clients. This can either happen through a broadcast mode (figure 7.2a) or through a peer-to-peer interaction between the clients (figure 7.2b).

In the broadcast mode of interaction (figure 7.2a), there is a central point of contact between the active client and the passive clients and a single virtual channel is used for communicating updates. All passive clients will therefore receive the same notification events. This implies that events have to be delivered at the rate of the fastest client, and any per-user impedance matching has to take place at the passive client.

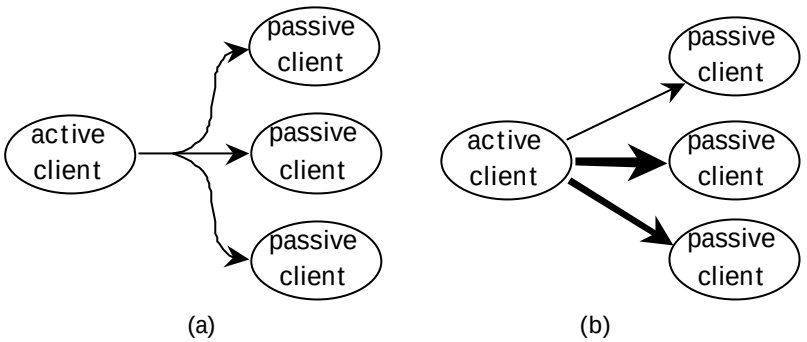


Figure 7.2 (a) broadcast and (b) peer-to-peer interaction

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 7.2b), the active client maintains a separate channel with each passive client. Consequently, the active client can itself filter the event stream on a per-client basis. The active client will however need to know about each individual passive client when replicating any changes. This form of interaction enables each passive client to receive different rates of feedthrough (represented by the different line thickness in figure 7.2b), but at the expense of some fairly complex filtering mechanism at every active client.

7.2.2 Interaction with notification server

The presence of the notification server allows both broadcast and peer-to-peer mode of interaction (figure 7.3). 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 (peer-to-peer).

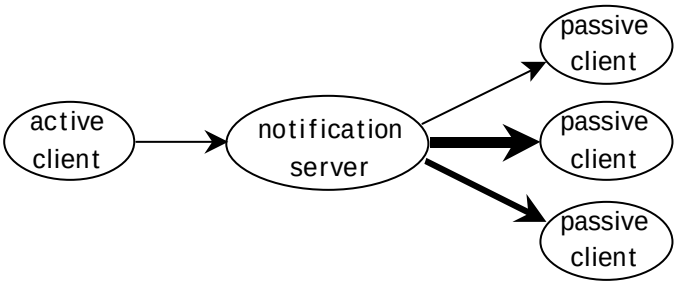


Figure 7.3 Using notification server as mediator

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 connect to 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.

7.3 Impedance Matching Policies

Having established that the notification server is ideally placed to support impedance matching, this section will explore the different ways in which impedance matching can be achieved and also consider some related implementation issues.

Impedance matching embodies both the volume of updates and the rate at which updates are notified to the users . The feedthrough demands can therefore be reduced by:

- sending updates less often (pace impedance)
- sending less updates (volume impedance)

Pace impedance deals with the frequency or rate of notification while volume impedance influences the amount of updates 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 network and computational resources. But this calls for a certain amount of filtering to be carried out.

7.3.1 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 in lowering network overheads. For instance, message headers need not be transmitted each time messages are broadcast.

7.3.2 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 low level of detail and yet still be acceptable to the users. Users could thus receive a shorter response time and the application could cope with a busy network. However, this should not jeopardise the quality of information broadcast.

An example of volume impedance is the use of flags for 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. So, even if a delay were introduced before the flags are sent out, this would not disrupt the user-level behaviour.

7.3.3 Impedance matching vs. 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.

For instance, when QoS is applied 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 available data.

7.3.4 Implementation Issues

During group work, users interact with several participants through a large number of shared objects over different timescales. Not everyone would necessarily be interested in the changes to all the interface objects at the same time. Users are more likely to have a higher interest in changes to certain objects than others. For example, certain interface objects may be regarded as *focus objects* and they require almost instantaneous feedthrough to be effective. Other less important objects may instead be considered as *peripheral objects* and the rate of change notification can be reduced accordingly.

With impedance matching, the server has to delay feedthrough to the clients. As a result, some form of event queues must be held before the updates are sent across to the clients. This lays an extra storage load on the server and it also implies that when updates are

eventually sent they have the accumulated size of all the delayed messages. Ideally, the server should be able to compress the event queues, for instance the event queue:

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

could be reduced to:

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

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

In an event-based model, when a user manipulates an object, the client generates an event, which then gets sent to the server. The server only knows about the types of events associated with the object and not the related pace of interaction. In order to provide the right pace of the feedthrough, the server should know about the pace of interaction associated with a certain object.

Users' client can therefore register a level of interest for an object with the server. For example, a client may register a high-pace interest for the *focus objects* but only a low-pace interest for the *peripheral objects*. In this manner, the server can deliver feedthrough at a rate that matches the users' pace of interaction.

7.4 Exploring pace policies

This section 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). A simple client-server mode of interaction is assumed, where user agents send messages to each other through a central server. Messages sent across the network are usually transmitted as lower level events.

Figure 7.4 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 the transmission of 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 7.4 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.

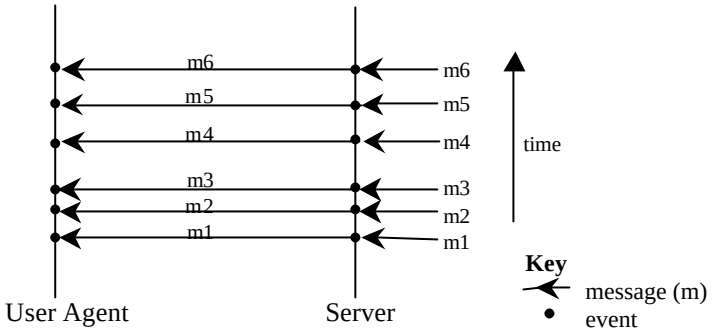


Figure 7.4 Time-space diagram without impedance matching

As pace impedance is about sending information less often (Section 7.3.1), one could ask the following 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

7.4.1 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 7.5 for example, the first message stream consists of messages m1, m2 and m3 but only m4 is sent out in the second message stream.

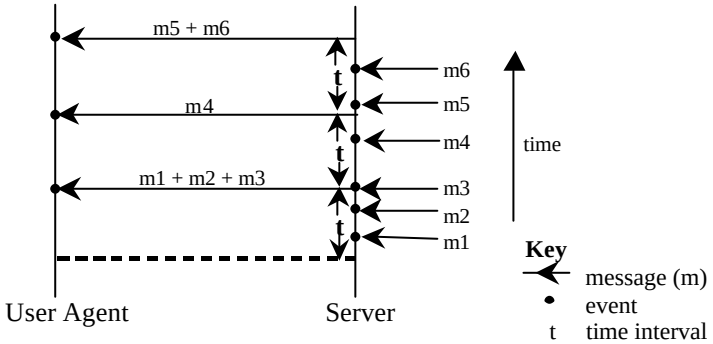


Figure 7.5 Time-space diagram with fixed time interval

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.

7.4.2 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 certain time delay. In figure 7.6 for instance, when the server receives the first message, it starts the timer and the messages are buffered until a certain time delay (δ) has passed, 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.

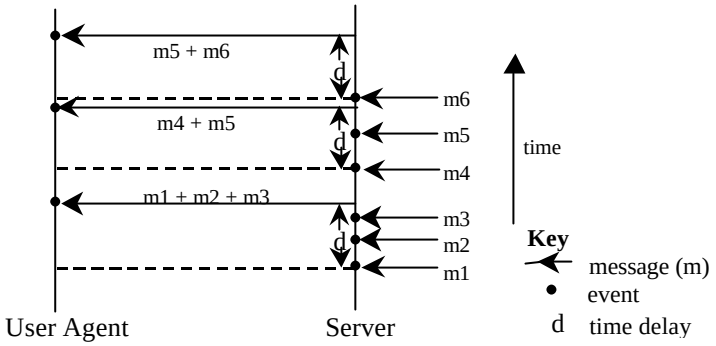


Figure 7.6 Time-space diagram with time delay

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.

7.4.3 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 7.7 shows the user agent receiving an event after the server has received a maximum of three messages.

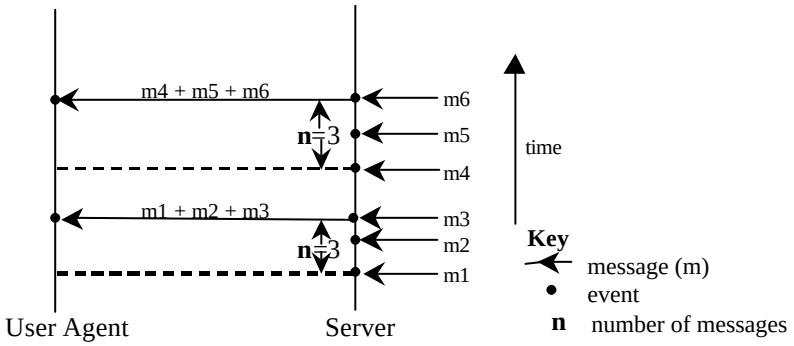


Figure 7.7 Time-space diagram with volume of messages

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.

7.4.4 Message size

With this option, the server forward messages to the clients once a maximum size is reached as it is not always effective to send several gigabytes of messages. Figure 7.7 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.

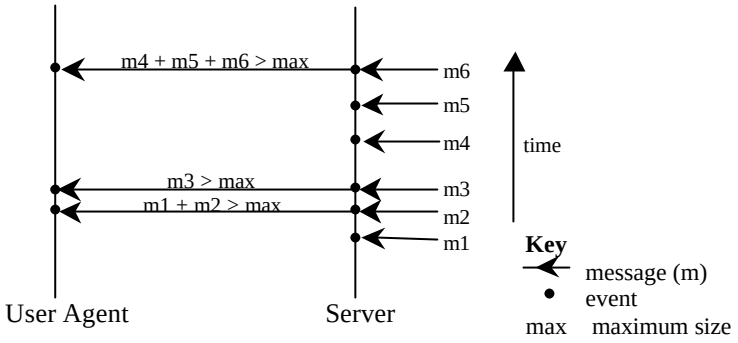


Figure 7.8 Time-space diagram with message size

7.5 Scenarios for impedance matching

This section will now look at some example collaborative scenarios and assess whether impedance matching could improve the provision of feedthrough. Three example systems are considered namely, a bulletin board system, a multi-user chat system and an avatar-based chat system. These applications have been chosen because they share a common factor – they all enable users to communicate with each other. However, each system supports communication over different temporal dimension and through a distinct interface.

7.5.1 Bulletin board system

A bulletin board system consists of a number of discussion forums which users can join and post messages to. Figure 7.9 shows the layout of an example university bulletin board system. Users can register to a number of discussion forums and add their contributions. They can also start up new topics of conversations and search for specific messages and respond to them.

Bulletin board systems often operate in an asynchronous mode. The rate at which users are notified of new contributions depends on the system. Some systems do not provide any form of explicit notification while others act as moderated email lists and send digests to users once every month. However, there are a small number of systems that notify users of the status of the latest posts on a daily basis, usually by email, which are either sent explicitly by the forum moderators or generated automatically. In the latter case, although the volume of information sent to the users is not significant, the email at least informs users that there have been some changes to the system. This is, in some ways, a kind of implicit impedance matching.

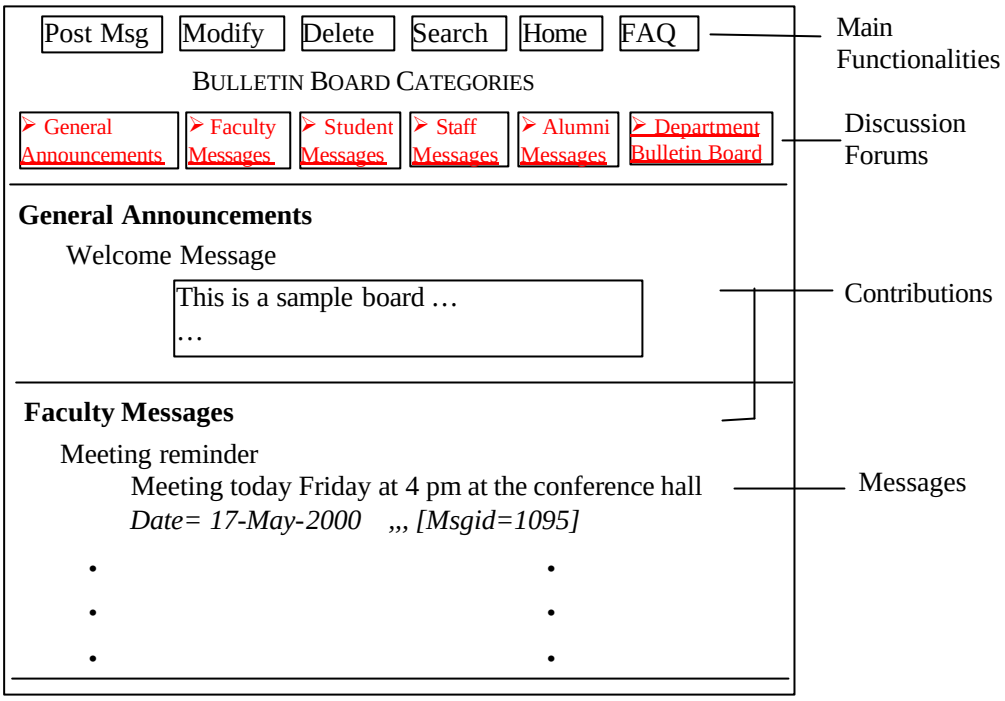


Figure 7.9 Example bulletin board system layout

Because interaction in bulletin board systems occurs over a fairly long-term (hours or days) users do not always expect a rapid response from other members. The rate of notification is too low, hence the demands on network load and bandwidth will not be significant enough to influence the rate of feedthrough. Therefore there is no need to implement impedance matching in this case as it will not necessarily improve users' performance to a great extent.

7.5.2 Multi-user chat system

A multi-user chat system also allows several participants to engage in discussions but unlike bulletin board systems, most of the communication here takes place in real-time. Users convene at virtual channels with a topic of conversation and hold public or private chat sessions. Example multi-user chat systems include Babble (Erickson et al., 1999) (figure 7.10) and Xchat (Zelezny and Langley, 1999) (figure 7.11).

Babble allows users to engage in both synchronous and asynchronous communication by maintaining persistent conversations. It uses some form of social protocol to display awareness information. Xchat is a graphical Internet Relay Chat (IRC) client that runs on Unix like systems. Both Babble and Xchat offer mainly a textual mode of interaction.

User interaction in multi-user chat systems occurs at a much faster rate than bulletin board systems in general, as information is mainly exchanged synchronously though various channels. The rate of update notification to the participants is higher and the task of managing the data exchanges and user controls becomes more complex.

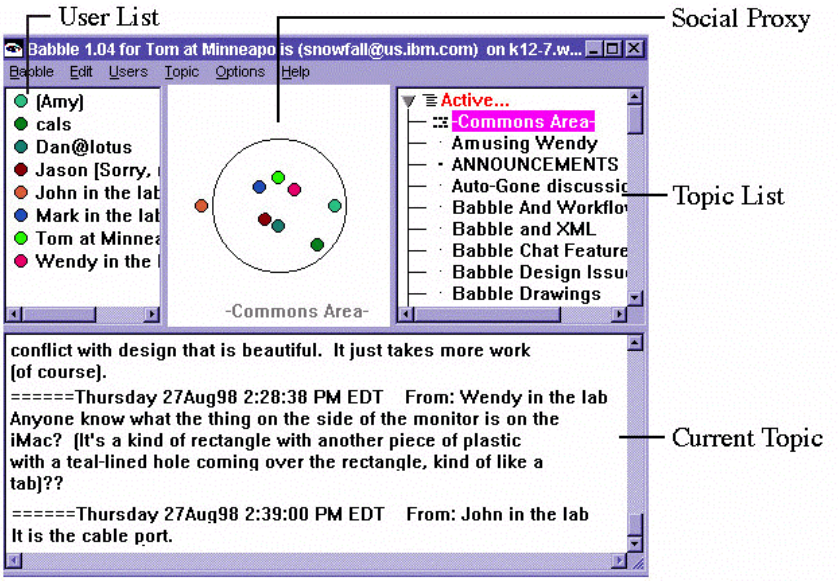


Figure 7.10 Example Babble screenshot

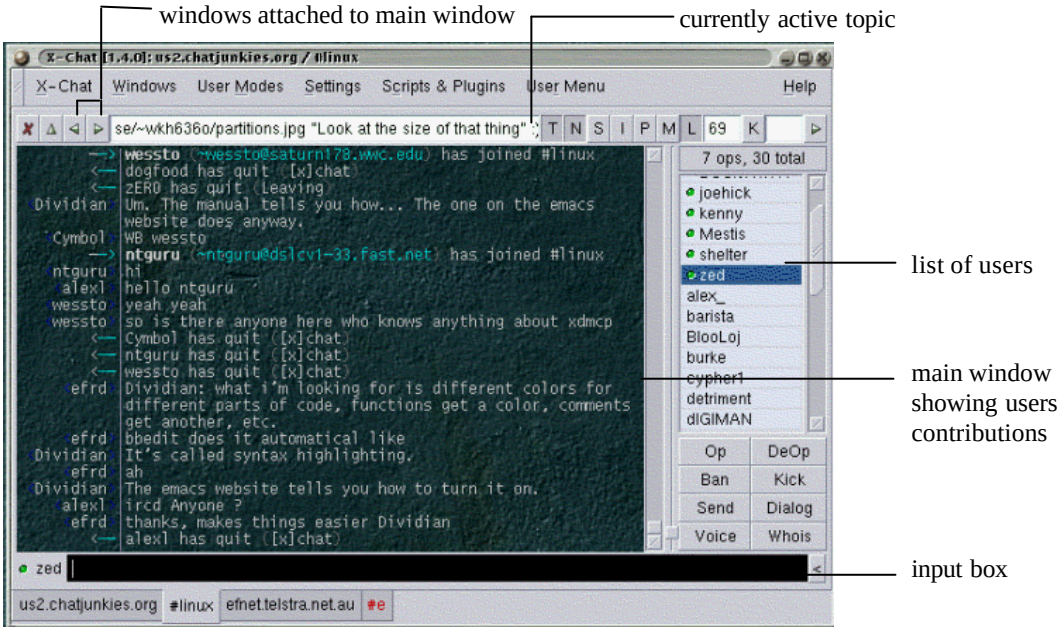


Figure 7.11 Example Xchat screenshot

When users maintain several channels of conversation simultaneously, it is more difficult to keep track of all the conversations together. Usually, each thread of conversation is displayed on separate windows. In Xchat for instance, the main window has several windows linked to it, each representing a separate thread of conversation, which is only brought to the front when activated by the user. However, users may also be interested to join in other conversations in the background windows at the same time.

The next section shows how impedance matching can be applied to notify users of the contributions in the different threads of conversations while their focus is on a particular chat session.

7.5.2.1 Applying impedance matching

Users in multi-user chat sessions are often involved in several discussions, however they tend to focus on one particular discussion at a time, typically represented by the conversation in the top-level window (figure 7.10). With impedance matching, a user's client can register a high-pace interest with the updates on the top-level window but only a low-pace interest with the changes in the secondary background windows.

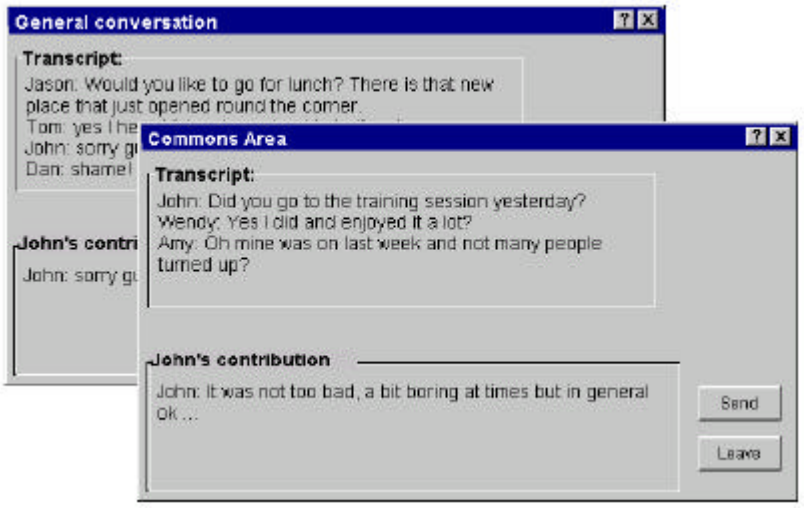


Figure 7.12 Example chat session with impedance matching

Figure 7.12 shows John, Wendy and Amy chatting in the 'commons area'. John is also involved in another chat session 'general conversation' in the background window. So with impedance matching, John will receive an instant feedthrough of any text entered in the 'commons area', but he will see the updates to the 'general conversation' less often. John may only be informed that the contribution has changed through some form of background feedback but this may not necessarily happen straight away.

7.5.3 Avatar-based chat system

Several chat systems have been developed to support distance collaboration in virtual 3D environments with the creation of virtual worlds and virtual communities (Greenhalgh and Benford, 1995). Web-based real-time chat systems have also become popular. For example the Active Worlds Browser[9] provides a 3D-type interface where users adopt an avatar, and unlike traditional chat rooms, a user can point-and-click to walk closer to other users.

[9] http://www.activeworlds.com/

Furthermore, systems such as MUDs (Multi-User Dungeons) or MOOs (MUD Object-Oriented) also require enhanced awareness mechanisms to make users feel their presence in virtual space. Issues such as the proximity of the users and their closeness to the artefacts play a significant role in maintaining awareness. The provision of a high pace of feedthrough is even more problematic in this complex environment under limited resources.

Some existing avatar-based chat systems use the notion of "rooms" to provide a spatial context where multiple users can play simultaneously. The notification of users' interaction and dialogue are usually via text. When a player is inside a particular room, she can hear every dialogue in that room alongside descriptions of other occupants' actions. After leaving the room, the player is no longer aware of the activities in the room she have just visited; instead she is given descriptions of her current location. However, some players may still be interested in the ongoing activities in the rooms they previously visited as they may wish to join in at a later stage.

The next section examines how impedance matching can be applied in such a system to provide users with a controlled pace of awareness of the activities in the different rooms, while their focus moves from one room to another.

7.5.3.1 Applying impedance matching

Typically, a user will focus on one particular room at a time; therefore a high rate of feedthrough must be provided for all the activities in that room. The rate of feedthrough for the activities in the secondary rooms need not be the same, but some additional form of awareness would be desirable. Figure 7.13 shows a screenshot of an example avatar-based chat room[10].

In the example avatar-based chat room, users 'newuser', 'Samantha' and 'harmsworth' conversing in the 'lobby' room (figure 7.13). Users can also join and leave chat rooms at any time. Consider the case when 'newuser' decides to join two other chat rooms while still conversing in the 'lobby'.

[10] http://www.weirdoz.org/visualchat/

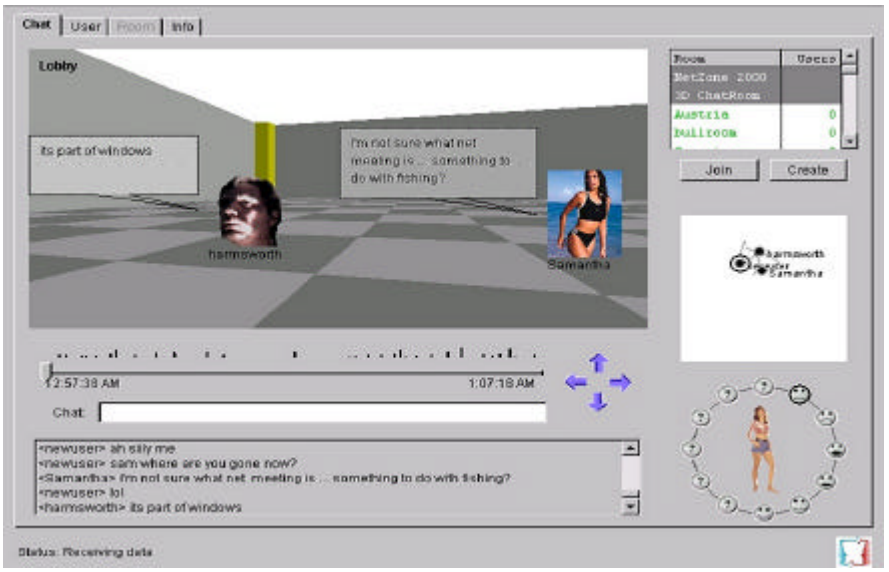


Figure 7.13 Example avatar-based chat room

With impedance matching, 'newuser' may be presented with the following interface (figure 7.14). The window on the left represents the conversation that 'newuser' is having in the 'lobby', where her main focus lies. The two reduced sized windows on the right instead represent the other chat rooms that 'newuser' is joined to at the same time, but in which she only has a secondary interest.

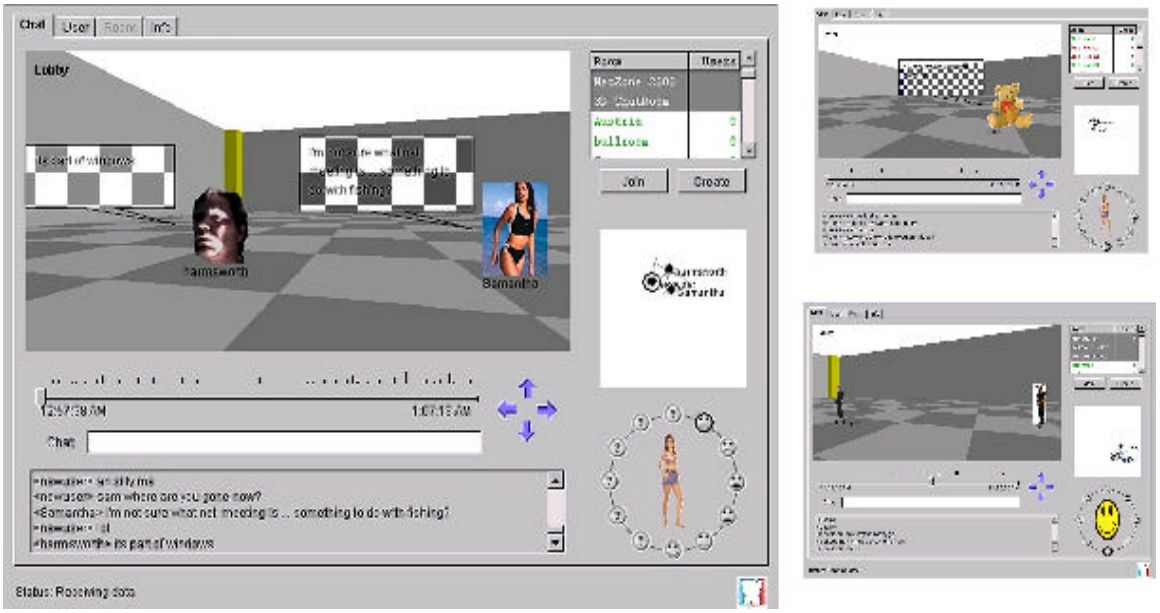


Figure 7.14 Example avatar-based chat room with impedance matching

A high-pace interest will be registered to the contributions in the 'lobby' but only a low-pace interest will be associated with the contributions in the other rooms. 'newuser' will therefore be notified of the changes to the 'lobby' almost as they occur and they will be displayed on the main window on the left. At the same time, 'newuser' will be made aware of the contributions to the other rooms on the windows on the right, but the rate of

notification will be at a much lower. In this manner, 'new user' can be notified of changes to the chat rooms depending on her interest rate.

In general, the rate of notification in the changes to the rooms users have registered a low-pace interest in, will be triggered by some sort of event for pace impedance, as discussed in Section 7.4. For instance, if the trigger is based on the temporal factor, updates may be sent out at regular time intervals or after a certain time delay. On the other hand, if the trigger is based on the volume, then updates may be sent out after a certain number of contributions have been received or when the size of the contributions has reached a maximum value.

In addition, some form of passive notification can be used to promote awareness, for example highlighting window frames, changing colour of text, raising a flag or even using some distinct sounds. The role of passive notification is mainly to convey awareness information for low-pace interest objects or *peripheral objects* (Section 7.3.4). The use of flags to mark new or changed materials is also an example of volume impedance and they do give temporal reduction for free (Section 7.3.2).

7.6 Further issues

The examples discussed above showed how impedance matching could improve the temporal behaviour of an application. However, the implementation of impedance matching generates some outstanding issues and these are discussed below.

7.6.1 Impact of rich media

The scenarios discussed in Section 7.5, mainly allowed information to be exchanged in a textual mode. However, some chat systems like ICQ[11] also enable users to exchange communication verbally via voice-over-IP through the use of Internet phone such as BuddyPhone[12]. 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 make the provision of feedthrough even more problematic.

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

[11] http://web.icq.com/

[12] http://www.buddyphone.com/

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 and Bly, 1992) makes use of frame-grabbing software for each media space and then distributes low-resolution digital images. Similarly, in NYNEX Portholes (Lee et al., 1997), although the WebCam operated at a slow rate, the images were 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.

7.6.2 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 7.15 illustrates the peer-to-peer ordering of events between two users, John and Mary chatting on two conferences, VRML and FILE MAKER.

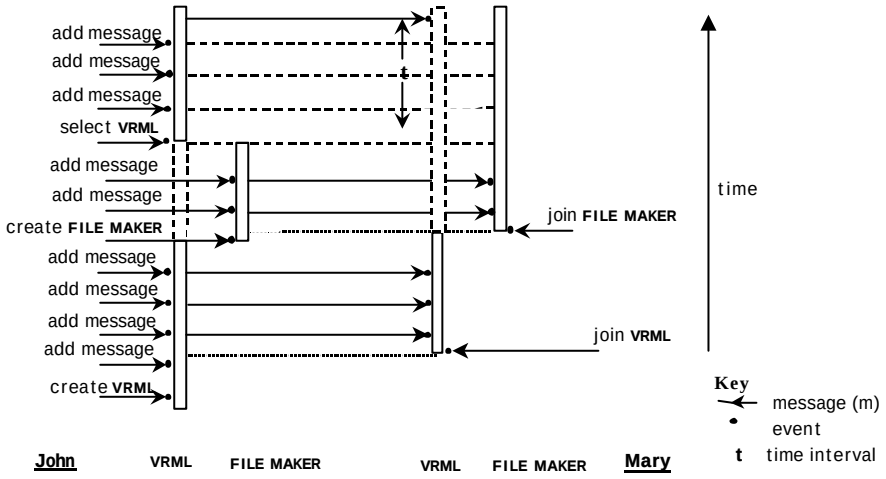


Figure 7.15 Timing diagram with point-to-point ordering of events

Starting with the lowest event, the timing diagram shows that John first creates the VRML conference and adds a message. 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, as illustrated by the conversation extract between John and Mary in figure 7.16.

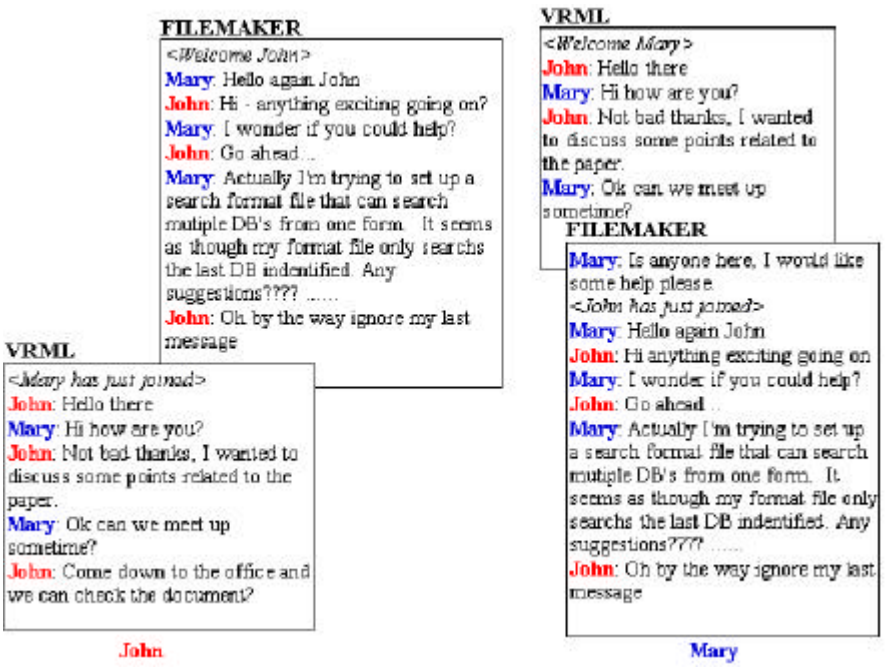


Figure 7.16 Example conferencing system transcript

Both John and Mary are chatting on two conferences, VRML and FILE MAKER. John's focus is on the VRML conference as the window is in the foreground whereas Mary's focus is on the FILE MAKER conference. Mary's last contribution that was addressed to John on the VRML conference was "ok can we meet up sometime". Mary's focus then switched to the FILE MAKER conference. As the VRML conference is no longer Mary's focus, John's reply is not forwarded to her immediately.

A short while later, John decides to join the FILE MAKER conference, where he catches Mary again and offers to help out with "Go ahead". John remembers their last conversation on the VRML conference, and seeing that it is not a good time for them to meet after all, he tells Mary "oh by the way ignore my last message". Mary is unaware of the context of this last message from John and she obviously interprets it as John being unable to help her with her FILE MAKER problem!

A message flags up on Mary's screen shortly after to inform her of a new contribution in the VRML conference. Mary now receives John's reply "Come down to the office and we can check the document?". So, Mary makes her way downstairs to John's office completely oblivious of the fact that John's last message on the FILE MAKER conference was actually meant to tell her not to come down to his office at this very minute!

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, event ordering 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 – the user can see the effects on different objects whereas the computer regards them as being distinct.

7.6.3 Priority of notification

The ordering of events may also be affected when events have some priority associated with them. The notion of priority may be useful during impedance matching so high priority events are serviced before low priority ones. The notification server could in fact use the priority as a means of flushing the queued events.

Consider a scenario where two messages are queued up at the notification server and another message with higher priority joins the queue. The notification server has two options: it can either flush the two outstanding messages from the queue immediately and send out the higher priority message straight after or it can leave the low priority events in the queue and deal with the high-priority ones first. The former option implies that low priority messages will be sent out at the same time as the high priority message, thus increasing the network load. The latter option instead requires the notification server to be aware of the priority constraints related to the ordering of the messages and this is problematic to deal with especially if there are some dependencies between the messages. The notification server will therefore need to have some semantic knowledge about the encapsulation of various kinds of messages and events.

The essence of impedance matching neither lies in changing the application semantics nor in increasing the load. Impedance matching is more concerned with reducing the computational and network load. However, by pushing more semantic knowledge towards the notification server, the latter will sidestep its fundamental role of routing events between clients during impedance matching and subsequently increase in complexity.

Another related issue is instead of having priority associated with a single event, priorities could be assigned to event types. The handling of events types in such a situation can be problematic. For example, if an event type is of a higher priority but has a slow pace then does this mean that it should overtake a high pace event type. Scheduling algorithms are traditionally employed to deal with multiple threads by prioritising them. However, scheduling will generate similar problems as event ordering particularly if high priority tasks

depend on low priority ones. Alternatively, such information could be pushed towards the clients so they can inform the notification server directly of the desired rates of handling the priority of notification. The clients will thus need to have a lot more knowledge about the ordering of the events and consequently increase in complexity.

7.6.4 Generating notification of non-events

Often some processes do not need to be aware of the occurrence of events but instead they do care when the event does not happen. For example, the alarm process in a heart beat monitor only goes off when a beat is missing! In the current framework, the support for the notification of non-events is only possible if all events go through the application. However, the addition of a specialised service such as a Watchdog Manager within the architecture (see figure 7.17) can facilitate this functionality.

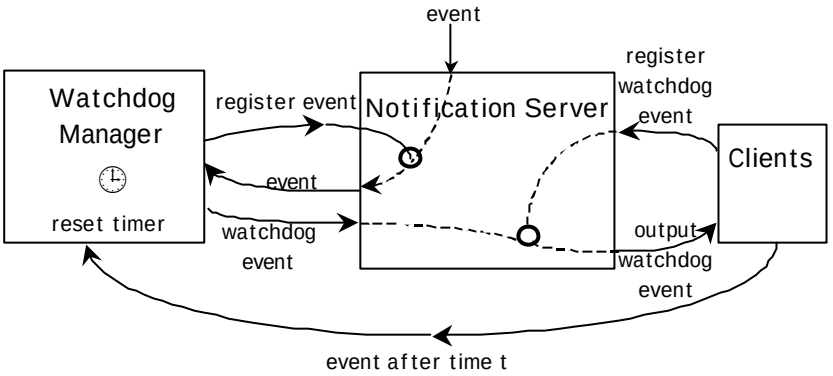


Figure 7.17 Monitoring the occurrence of non-events

When an event occurs the Notification Server informs the Watchdog Manager of that particular event. The Watchdog Manager then registers for the event with the Notification Server. The client also registers for watchdog events with the Notification Server. The Watchdog Manager keeps an internal timer and checks for the occurrence of the registered events. If an event does not occur after a certain time, the timer is reset and the Watchdog Manager generates a watchdog event, which is sent out to the clients.

The Watchdog Manager can thus monitor the generation of non-events and inform the clients directly about them. Furthermore, the Watchdog Manager can reside on the same machine as the Notification Server or it could be in-built within the Notification Server, hence the above architecture would be efficient.

7.6.5 Optimising the timing of notification delivery

Consider the following scenario. Client A is interested in event ① and wants to know about it after every 10 seconds. Client B is also interested in event ① but it wants to know about it after every 1 second. In order to optimise the timing of notification delivery, the notification server can broadcast event ① to both clients simultaneously, thus reducing the network load. Client A may either ignore event ① or it may place it in a queue and batch process it later. The same thing could happen if say event ① is in a queue at the notification server and the latter is about to send out event ② to client A. The notification server could piggyback both events at the same time as they have the same recipient.

However, like priority issues, the optimisation of events may affect the ordering of the events. For example, let us assume that client A is interested in both events ① and ② but it does not want to know about event ① that often, whereas client B has a high urgency for event ①. When the notification server broadcasts event ① to both clients, there is a danger that client A may receive event ② before event ①. So, if the order of the events matters for client A, in other words event ② depends on event ①, this will give rise to the problem of race condition. This situation is problematic to deal with unless the clients take it upon themselves to manage the order of the events and this requires the clients to have a lot more semantic knowledge.

7.6.6 Impedance matching in other areas

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

The collaborative model of awareness based on the spatial interaction of objects (Benford and Fahlén, 1993) lies on the 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 fairly similar to that of the *focus objects* (Section 7.3.4). 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 been augmented with third party objects (Benford et al., 1997), which use aggregation to achieve a form of 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 used in the HIBROWSE interface (Ellis et al., 1994) to provide users with an overview of the contents of the database while searching and browsing large data sets.

The implementation on the HIVE CVE system (Greenhalgh et al., 2000) uses full fidelity information but it seems likely that scaleable implementation will require pace management.

7.7 Summary

The discussion in this chapter has centred on the analytic framework for impedance matching – the matching of the required and supplied of update events. The notification server, through its central mediating position, was found to be ideally placed to support impedance matching, by adjusting the frequency of notification to meet the users pace of interaction. Users can thus see the changes in the objects they are highly interested in almost instantly, while still being informed about changes to the peripheral objects, albeit at a lower pace. Impedance matching therefore enhances both goal-directed feedthrough and awareness, thus exploiting the limited availability of computer resources and network bandwidth.

In order to enable the notification server to provide effective impedance matching that satisfies each client's requirements, the clients should inform the notification server of their required pace interest on particular objects via some form of protocol. The communication between the clients and the notification server does not require the latter to have any knowledge of the application semantics; hence the notification server can still remain as a separate entity.

The issues surrounding pace impedance and volume impedance were then examined. Pace impedance can be achieved by using the notification server as an intermediary to match the required pace of updates of the passive client (user client who views the changes) with the supplied rate of the active clients (users' client who perform changes). Volume impedance can be largely met by having different forms of application-specific, low-granularity update events. However, the issues surrounding volume impedance have not been dealt with in much detail in this chapter. Pace impedance policies were analysed further by investigating the different triggers for regulating pace and showing their effects on the flow of events through the use of time-space diagrams.

Some implementation issues related to impedance matching were also considered. A few scenarios were then explored to assess the feasibility of impedance matching within a collaborative environment. The example systems facilitated communication over different timescales, thus producing different rates of feedthrough. Impedance matching was found to improve the temporal behaviour especially in situations where a large number of updates were rapidly generated. Finally, some outstanding issues related to impedance matching were discussed.

Impedance matching controls the pace of feedthrough to the clients by delaying the updates events. This may affect the order in which the events are propagated to the clients. The incorrect ordering of events will not have a big impact if there is no causality or dependency between them. But in a chat system for instance, users can be easily confused if the messages exchanged reach them in the wrong order. Furthermore, the way in which people interact socially with one another can also influence the order of the messages. Some

systems avoid getting things in the wrong order by manipulating the semantics and interconnections between the events, at the expense of some complex algorithms.

Additional issues such as the priority of notification, the notification of non-events and the optimisation of notification delivery were considered and their impact on the impedance matching framework, in particular the way in which they may affect ordering of events was discussed.

In order to investigate the actual behaviour of a notification server as an impedance matcher, an experimental notification server called Getting-to-Know (GtK) has been constructed, which will be described in the next chapter. GtK demonstrates most of the design principles discussed in this chapter, but the provision of impedance matching is limited to pace impedance matching based on the 'fixed time interval' and 'volume of messages' triggers.

Chapter 8 Getting-to-Know: An experimental Notification Server

Notification servers operate at a low-level within the computer system but their ultimate purpose is to provide effective user-level behaviour. In Chapter 6, the framework for the design options of notification servers emphasised the need for a separable notification server. Chapter 7 presented an analytic framework for impedance matching to provide users with a controlled pace of feedthrough, thus promoting temporal interface behaviour in collaborative applications. The notification server was found to be ideally placed to perform impedance matching between end-user clients.

The framework for impedance matching can be applied to augment other notification mechanisms or build new notification servers over different low-level messaging infrastructures. Getting-to-Know (GtK) is largely an example to show that the principles for notification server design and impedance matching can be achieved in a practical implementation. This chapter describes the issues surrounding the development of the GtK purpose-built separable notification server, which is based on a distributed object infrastructure. Some of these implementation issues are discussed in (Ramduny and Dix, 2002).

Impedance matching can be achieved through pace impedance and volume impedance. Volume impedance is essential for controlling the quality/fidelity of notified information. However, the issues surrounding volume impedance require further investigation, hence it has not been implemented within GtK. Similarly, GtK is not concerned with event ordering issues, as a solution to this problem lies either at the underlying system level or at the programmer level in understanding the semantics of the infrastructure. GtK only supports pace impedance, based on two triggers: fixed time interval and volume of messages.

Section 8.1 describes the basic distributed layered architecture that supports GtK. Section 8.2 examines the messaging and event layer and analyses the protocol employed for passing messages between different communication objects within the GtK infrastructure. The main functions of the GtK notification server are discussed in Section 8.3. Section 8.4 shows how GtK has been augmented to provide pace impedance matching. Finally, Section 8.5 considers the exchange of messages and events between the different components of an example real-time Web conferencing application, which has been constructed to explore the practicality of GtK further.

8.1 Basic architecture

The GtK notification server is built over several layers of custom and standard infrastructure (figure 8.1). At the base lies the standard low-level Internet TCP/IP protocol accessed via Java networking classes.

The Java `socket` class is used to implement a reliable stream network connection between one or more clients and a multi-threaded server. The server uses the `ServerSocket` class to accept connections from clients on a particular port. When a client connects to the port, the `ServerSocket` allocates the client a new `socket` object attached to a new port to enable it to communicate with the server. The server then carries on listening on the `ServerSocket` for additional client connections.

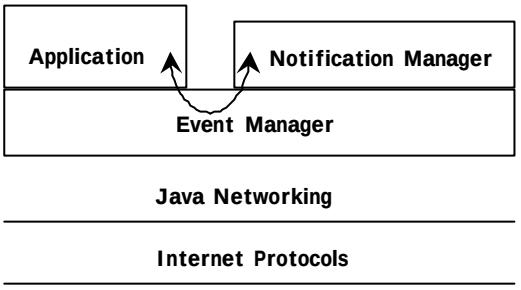


Figure 8.1 GtK infrastructure

On top of the *Internet Protocols* and *Java networking* layers, there is a custom event management layer, the *Event Manager*, which supports directed message delivery between agents on different physical machines.

Finally, the *Notification Manager* uses the *Event Manager* to allow the GtK notification server to receive notifications about changes from active clients or information servers and to pass on the notifications about those events to the passive clients.

8.2 Messaging and event layer

A messaging system essentially allows separate, uncoupled applications to reliably communicate asynchronously. The messaging system architecture generally replaces the traditional client/server model with a peer-to-peer relationship between individual components. Messaging systems offer several advantages. They encourage loose coupling between components, thus enabling dynamic and flexible systems to be built, whereby some components can be modified without affecting the rest of the system. They also provide high scalability, easy integration into heterogeneous networks and reliability due to lack of a single point of failure.

With the proliferation of distributed applications, a number of messaging systems have recently emerged to deal with the problems of synchronisation, reliability, scalability and security. There are three types of messaging systems that are commonly used:

- Publish/Subscribe – this supports an event-driven model where producers "publish" events, while consumers "subscribe" to events of interest and consume the events. Producers associate messages with a specific topic and the messaging system routes messages to consumers based on the topics consumers have registered their interests in. This model therefore supports multiple senders and multiple receivers.
- Point-to-point – this model is used when one process sends a message directly to another process. Usually, messages are routed to an individual consumer, which maintains a queue of "incoming" messages. Messaging applications send messages to a specified queue and clients retrieve messages from the queue. Although there may be multiple senders of messages, there is only a single receiver for the messages.
- Request-Reply – this model is used when an application sends a message and expects to receive a reply in return. This is the standard synchronous object-messaging format and is often defined as a subset of one of the other two models.

Very often, messaging systems support both point-to-point and publish/subscribe messaging models. An example is the Java Message Service (JMS)[13], which is part of J2EE (Java 2 Enterprise Edition). Although JMS defines Queues (for point-to-point) and Topics (for publish/subscribe) as targets for messages, it does not require the provider to implement both.

SOAP[14] (Simple Object Access Protocol) describes the format for XML-based messages exchanged on the Web. It is a lightweight protocol for exchanging information in a decentralised distributed environment. SOAP RPC (Remote Procedure Call) uses XML for marshalling requests and replies. SOAP messaging is based on a point-to-point model with request-reply. Messages are exchanged either in the form of an inquiry, which is initiated by the client, or an update, whereby the server sends information to all registered clients in a 'push' format. The exchange can either be in a synchronous or an asynchronous mode.

8.2.1 Messaging protocol

The event management layer within the GtK architecture implements a distributed asynchronous messaging protocol, thus giving GtK a uniform, generic location-independent event model. 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 has to be integrated with user interface code using threads. Consequently, interfaces end up using two models: event-based windowing code

[13] http://java.sun.com/products/jms/faq.html

[14] http://java.sun.com/webservices/docs/1.0/api/

and RMI networking code. An asynchronous event model for distributed agents is preferable as it is closer to the way modern UI code works and it also allows a uniform model between user events and remote events. Thirdly, RMI depends on Java serialisation, which does not tend to be 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.

The Event Manager does the marshalling of events or messages between the Notification Manager and the Application (figure 8.1). 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 by applying a post office type metaphor.

8.2.2 Message format

All messages and events are of the simple form:

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

where `sender reference` is of type:

```
sender_id : object_id
```

and recipient reference is of type:

```
recipient_id : host_id
```

Each message has an associated type:

```
event_type
```

and the actual message is a tuple:

```
data
```

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

8.2.2.1 Message class

The `Message()` class provides utilities to structure the events exchanged between the communication objects.

```
public class Message {

// Public Constructors

   public Message (int fromObj, int fromHost, int toObj, int
                   toHost, String eventType,String data);

   public Message (String mess);

// Public Instance Methods

   public String format(); // returns formatted message
   public void parse(String mess); // unpacks message
}
```

`format()` allows a message to be structured in the appropriate form whereas `parse()` unpacks the message to extract the relevant data.

8.2.2.2 Event handler

Objects that handle events do so through instances of the methods of the `EventHandler()` interface.

```
public interface EventHandler {

// Public Instance Methods

   public abstract void youAre(int id);
   public abstract void newEvent(Message event);
}
```

When an object first connects to the Event Manager, it registers itself as an event handler. The connection handler uses the `youAre()` method to tell the object its identifier (`id`). `newEvent()` allows the object to handle a `Message` event. A new object is assigned a new `id` through the `addObject()` method. The event handler object and its `id` are then stored in a table.

```
// Public Instance Method in CommonConnectionHandler class

public static int addObject(EventHandler obj) {

   int newId = tmpId+1;          // newId value = old value + 1
   if (newId not found in event handler table myObjects) {

      // add obj EventHandler and newId in table myObjects

      myObjects.put(newId, obj);
      obj.youAre(newId);}        // assign newId to EventHandler
   tmpId = newId;             // assign tmpId to newId value
```

```
    return newId;              // return newId value
}
```

8.2.3 Message exchange

The following fragments of code show how messages are exchanged between client and server objects. A client object registers itself to the server and launches the appropriate event handler through the `newEvent()` method call.

```
// thread in ServerConnectionHandler and ClientConnectionHandler
   class

do { // keep reading from connection until user exits

   str = readLine(); // read in a line

   // calls Message class to parse the string and get recipient id
   Message M = new Message (str.trim());

   // check to see if recipient id is in table myObjects
   EventHandler theObj = (EventHandler)myObjects.get(M.toObj);
   if null error
   else
      theObj.newEvent(M); // launch event handler
}
```

For instance, if the client object launches the `TranscriptPanel()` event handler, the call in the `setTranscript()` method shown below sends the `eventType` of the message to the server object. The `eventType` is a specified string through which the client object informs the server object of the type of event that is being exchanged. The same form of exchange occurs between the server object and the client object.

So assuming that the `eventType` is "`new client`", the call in `setTranscript()` changes to:

```
//call from public instance method setTranscript() in Client
   TranscriptPanel()class
   ...
   mySharedData.ch.sendTo(myId, toObj, toHostId, "new client",
                          mySharedData.userName);     --------①
   ...
```

Different kinds of events can be exchanged between client and server objects. For example, a "`new client`" event from the client tells the server that a new client has just joined, a "`new line`" event instead tells the server that the user client has sent some text and a "`client left`" event states that the client has left.

Both client and server objects use the `sendTo()` method call referred in ① to send formatted messages to each other. `sendTo()`first parses the input stream to extract all the relevant data before sending the message across.

```
//Public Instance Method in ServerConnectionHandler and
   ClientConnectionHandler class

public static void sendTo(int fromObj, int toObj, int toHost,
                          String eventType, String data) {
   Message M1 = new Message(fromObj, fromHost, toObj, toHost,
                            eventType, data);
   String theMess = M1.format();
   output.println(theMess);    // send formatted message
}
```

When the server receives a formatted message from the client, its `TranscriptObject()` event handler interprets the `eventType` through its `newEvent()` method call.

For example, if the server receives a `"new client"` event, the server object translates it to a `"greeting"` event, which is sent back to the client object through a `sendTo()` method call. The client object in turn, converts the `"greeting"` event, through its own `newEvent()` method call in the `TranscriptPanel()` event handler to display a more meaningful message to the user screen such as "Hello user".

```
// Public instance method newEvent() in Client class
    TranscriptPanel()

public void newEvent(Message event) {
   ...
   Object etype = event.type;
   if ( etype.equals("greeting") ){

     // display greeting text on screen
     transcript.appendText( "Hello "+ event.data + "\n");
     } else if ... { ... }
     else // event type not recognised by receiver
     { transcript.appendText(etype + " not recognised by " + myId
     + "\n");}
}
```

This section has described how the Event Manager has been implemented to support the exchange of messages and events. The next section will now consider the functionalities of the Notification Manager.

8.3 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.

A new client object registers itself with the Notification Manager in the same way as it did previously with the Event Manager by launching the appropriate event handler. However, unlike ① (Section 8.2.3), the recipient's reference `toObj` is now replaced by `NOT_MGR`, a well known identifier for the Notification Manager. Similarly, the recipient's host reference `toHostId` is now the Notification Manager's reference `notHostId`.

```
//call from public instance method setTranscript() in Client
   TranscriptPanel()class

   ...
   mySharedData.ch.sendTo( myId, NOT_MGR, notHostId, "add
                           interest", d.format() ); ---------②
   ...
```

Also, instead of passing a data string across as in ① a formatted data packet `d.format()` is transmitted in ②. This contains specific information related to a particular client, such as its identifier, event type and remote client event type all bundled together. Furthermore, the new client object now sends out an `"add interest"` event to the Notification Manager.

8.3.1 Main functions

The Notification Manager handles the three main functions of the GtK notification server:

- add interest – tells the notification server that a specific network object wants to know about specific events for a second network object

```
public static synchronized void addInterest (int objid, String
                                eventType, int clientid, int
                                remObjid, String remeventType);
```

- remove interest – tells the notification server to cancel some or all of the interests for a given object

```
public static synchronized int removeInterest (int objid, String
                                eventType, int clientid, int
                                remobjid);
```

- tell all – asks the notification server to broadcast an event to all interested objects

```
public static synchronized void tellAll(int objid, String
                                eventType, String data);
```

The parameters used by `addInterest()`, `removeInterest()` and `tellAll()`methods are described below.

`int objid`: reference of the object which wants to register an interest

`int remobjid`: remote network object reference

`int clientid`: client reference where the request is coming from

`String eventType`: event type of the object

`String remeventType`: event type of the remote network object

`String data`: actual message that gets sent between the network objects

The above 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 (Flannagan, 1997), except that these Java events are limited to a single Java process.

8.3.2 Managing interests

GtK maintains an interest table that keeps a list of interested clients for specific objects. Each object in the interest table has one or more recipients (figure 8.2).

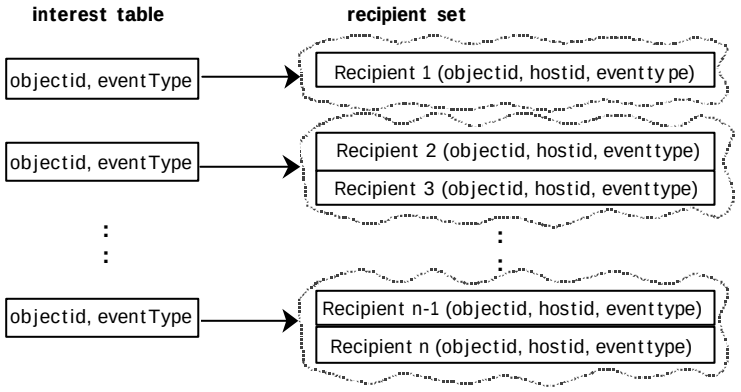


Figure 8.2 Interest table

The interest table is updated through the ‘add interest’ and ‘remove interest’ functions. The following pseudocode explains how this happens.

8.3.2.1 Add interest

```
public static synchronized void addInterest (int objid, String
                                eventType, int clientid, int
                                remObjid, String remeventType) {

  recipient set for Object = interestTable.get(objid, eventType);
  if null{ // first time object referred to
    put (objid, eventType) in interestTable; }
  create new Recipient(remobjid, clientid, remeventType);
  add Recipient to recipient set;
}
```

8.3.2.2 Remove interest

```
public static synchronized int removeInterest (int objid, String
                                eventType, int clientid, int
                                remobjid) {

  recipient set for Object = interestTable.get(objid, eventType);
  if null { error }  // first time object referred to
  for each object in the recipient set {
    get Recipient (toObj, toHost, etype);
    if ( Recipient.toHost == clientid && Recipient.toObj ==
           remobjid){
      remove Recipient from recipient set;}
  }
  if recipient set is empty {
    remove (eventType, objid) from interestTable;
  }
  return 0;
}
```

8.3.3 Broadcasting events

Events are broadcast through the ‘tell all’ function. When an object asks GtK to `tellAll()`, GtK first matches the event type and objects with the interest table and then passes on the event with optional type translation to all interested clients.

8.3.3.1 Tell All

```
public static synchronized void tellAll(int objid, String
                                        eventType, String data)
                                        {

   recipient set for Object = interestTable.get(objid, eventType);
   if not null {
      for each object in the recipient set {
         get Recipient (toObj, toHost, etype);
         // send data to all clients
         ServerConnectionHandler.sendTo(objid,Recipient.toObj,
                     Recipient.toHost, Recipient.eType, data);}
   }
}
```

Type translation is a method that is used to represent the same event to two different remote objects by giving it some meaningful name. For example, the `eventType` in `tellAll()`gets translated to a different event type at the Event Manager level, based on the recipient's event type `Recipient.eType`. The next section gives an example that shows type translation in progress.

8.3.4 Illustrating type translation

Consider an online conferencing system where users interact through an applet interface. After registering to the conferencing system, users can participate in one or many conferences. Each conference is managed by a 'transcript object' at the server-end and each client applet has a conference object attached to it. The Notification Manager mediates the interaction between the client and server objects. Each object is identified by a `HostId` and an `ObjId`. Figure 8.3 shows the flow of events between the different client and server objects.

When a user joins a certain conference, the client-end conference object sends an event to the Notification Manager to tell it to 'add an interest' for that user and that particular conference in the 'interest table'.

```
addInterest (99, "new client", 7, 115, "new line")
```

Consider the case when the user is about to leave the CSCW conference, she joined in earlier. In response to the user's input "bye", the applet object sends the following message to the server-end 'transcript' object:

```
115:7:43:0:new line:bye
```

The transcript object updates its internal state and asks the Notification Manager to inform all the clients who have an interest in the CSCW conference through:

```
tellAll (43, “new line”, “bye”)
```

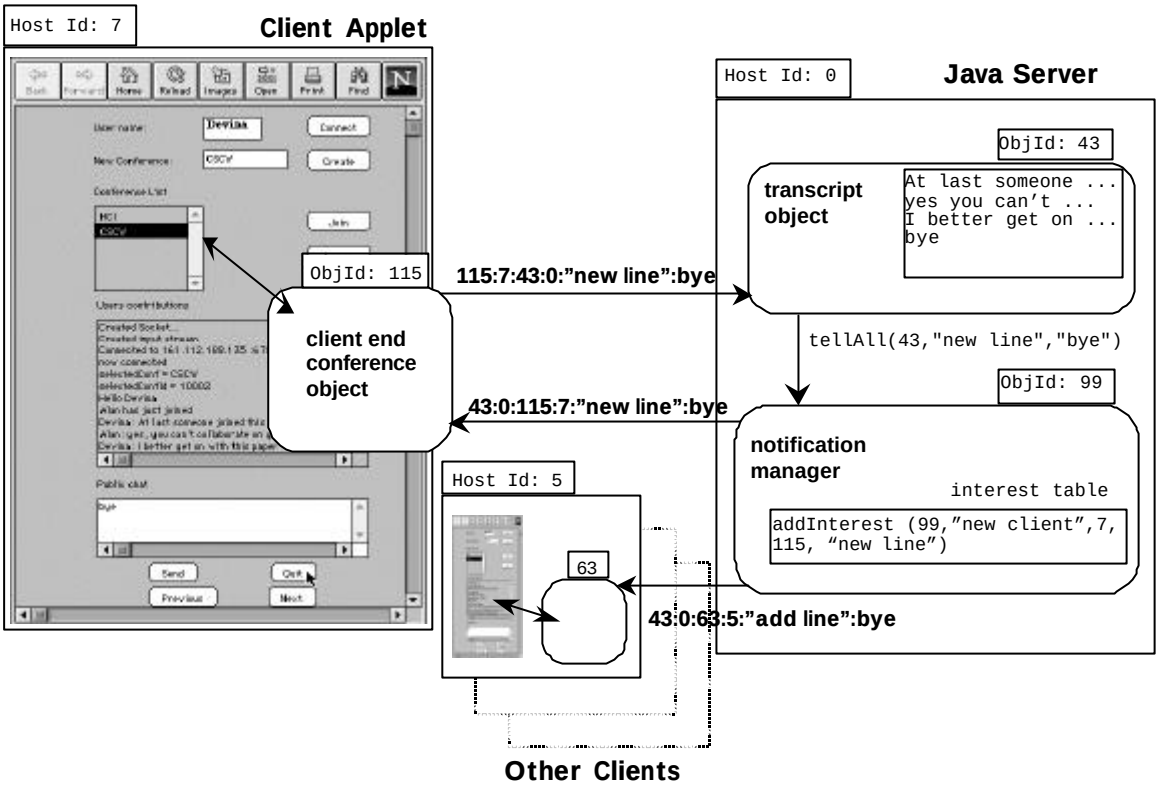


Figure 8.3 Flow of events between client and server objects

The Notification Manager broadcasts the message to all the client objects, but with a different `eventType` in each case. For instance, the Notification Manager may send this event to the client object, with `Objid`, 115:

```
43:0:115:7:new line:bye
```

but the following event to the client object, with `Objid`, 63:

```
43:0:63:5:add line:bye
```

The `eventType` has therefore been translated from `“new line”` in the client with `Objid` = 115 to `“add line”` in the client with `Objid` = 63. The client objects use this information to update the content of the windows on each user's machine.

Event translation simplifies the process of differentiating between different clients. The clients essentially receive the same event, but in a slightly different format. If all the interested clients received the same `eventType`, it would be more difficult to track the origin of the message. Event translation therefore helps to increase comprehension.

8.4 Augmenting GtK for Impedance Matching

This section will now describe how the GtK notification server has been augmented to provide pace impedance. Although impedance matching emerges from an abstract notion, it has been implemented in GtK to investigate its actual behaviour on a practical level.

As discussed in Chapter 7 (Section 7.3), 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 updates.

8.4.1 Pace parameters

Two pace parameters have therefore been defined:

```
int 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 passed on to the client. This corresponds to the 'volume of messages' trigger (Section 7.4.3).

```
long time;      // duration
```

`time` enables a client object to specify the maximum delay on events before they are sent out and this represents the 'fixed time interval' trigger (Section 7.4.1).

`queueLength` and `time` are combined into a single data structure:

```
Frequency = (queueLength: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.

8.4.1.1 Frequency class

```
public class Frequency {

// Public Constructors
   public Frequency(int queueLength, long time);

// Public Instance Methods
   public long getQueueLength(); // returns queue length
   public boolean timeNow(int currentQueueLength, long
            currentDelay); // check if is time to flush queue
```

```
    public long nextTime(long oldestMessageTime);
                            // calculate next time to flush queue }
```

8.4.2 Managing interests with frequency

The need to support pace impedance 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 8.4).

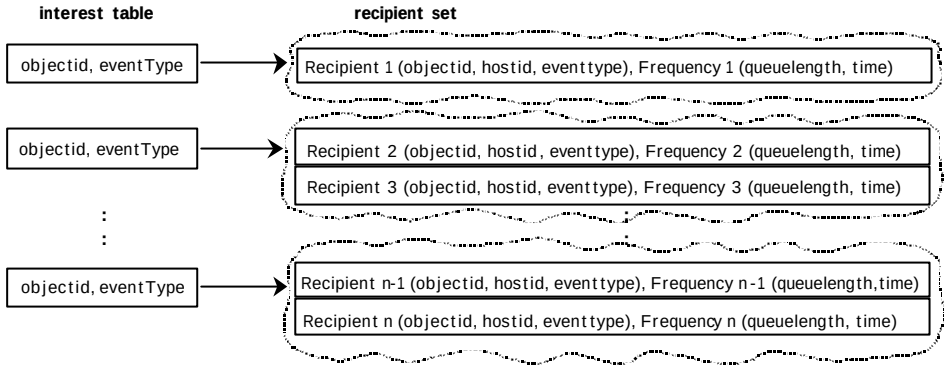


Figure 8.4 Effect of pace impedance on interest table

A `NotifRecord()` class is created to maintain the link between each recipient and its frequency. `NotifRecord()` provides methods which allow recipients to manage their event queues.

```
public class NotifRecord {
// Public Constructor

   public NotifRecord(Recipient recipient, Frequency howOften);
                  // creates a record of recipient and frequency
   // Public Instance Methods

   public void changeFrequency(Frequency howOften);
                              // change frequency of notification
   public int queueLength();                 // returns queue length
   public long time();              // return current system time
   public boolean timeToFlush();    // check if time to flush queue
   public long nextFlushTime(); // return next time to flush queue
   public void addQueue(String message); // add message to queue
   public String removeQueue();       // remove element from queue
}
```

When a network object now adds an interest for a second network object, it also includes the frequency with which it wants to be notified of the changes to the second object. The

'add interest' function (Section 8.3.1) therefore accepts the pace parameters, `queueLength` and `time` as part of its call:

```
public static synchronized void addInterest (int objid, String
                                  eventType, int clientid, int
                                  remObjid, String remeventType,
                                  int queueLength, long time)
```

The following pseudocode shows how the interest table is updated when the frequency of notification is taken into account.

8.4.2.1 Add interest

`addInterest()`starts with the same instructions as before (Section 8.3.2.1). However, after creating a recipient, the method creates a corresponding frequency and they are linked together to generate a `NotifRecord`. The `NotifRecord` is then added to the recipient set and linked with the relevant record in the interest table.

```
public static synchronized void addInterest (int objid, String
                  eventType, int clientid, int remObjid, String
                  remeventType, int queueLength, long time) {

   recipient set for Object = interestTable.get(objid, eventType);
   if null{ // first time object refered to
     put (objid, eventType) in interestTable ; }
   create new Recipient(remobjid, clientid, remeventType);
   create new Frequency(queueLength,time);
   create new NotifRecord (Recipient,Frequency);
   add NotifRecord to recipient set;
}
```

8.4.2.2 Remove interest

`removeInterest()` removes some or all the recipients for a certain object in the interest table by matching the `NotifRecord` in the recipient set with the client reference where the request came from.

```
public static synchronized int removeInterest (int objid, String
                        eventType, int clientid, int remobjid) {

   recipient set for Object = interestTable.get(objid, eventType);
   if null { error } // first time object referred to

   for each object in the recipient set {

     get NotifRecord (Recipient (toObj, toHost, etype),Frequency);
     if ( NotifRecord.Recipient.toHost == clientid &&
          NotifRecord.Recipient.toObj == remobjid) {
```

```
        remove NotifRecord from recipient set; }}
    if (recipient set is empty {
      remove (eventType, objid)from interestTable; }
    return 0;}
```

8.4.3 Event queue management

For each recipient, a queue of outstanding events is maintained. New events are timestamped and added to the rear of the queue. Events are flushed and delivered to recipients at appropriate times depending on the pace parameters.

Events are flushed from the queue when either:

(a) current queue length > value set for `queueLength`.

(b) current delay > value set for `time`.

where current delay = (current system time – time first event was queued).

Case (a) is triggered when an event is added to the queue.

Case (b) requires an alarm process to be set in the notification server. Conceptually, there is one alarm for each non-empty queue, which is set to flush the queue at:

`nextTime` = (timestamp of the first event in the queue + `time`)

In the actual implementation, only one alarm process is used and set to the closest of the relevant alarm deadlines. This is reset when events are added to an empty queue or after the queue is flushed due to the previous alarm.

8.4.3.1 Tell all

With impedance matching, the 'tell all' function has to provide different rates of feedthrough. Instead of broadcasting the notification events to the interested clients straight away as before (Section 8.3.1.1), `tellAll()` now adds the event for each interested object to its recipient's queue and records the time the event was added. It then calls `checkFlush()`, which checks if it is time to flush the queue (when either case (a) or case (b) above is satisfied).

```
public static synchronized void tellAll(int objid, String
                                        eventType, String data) {

  recipient set for Object = interestTable.get(objid, eventType);
  if not null {
    for each object in the recipient set {
      get NotifRecord();
      NotifRecord.addQueue(data); // add event to the queue

      // check if it is time to flush the queue
      checkFlush(objid, eventType,NotifRecord);}
  }
```

```
}
```

If the time to flush the queue has been reached, `checkFlush()` sends the events to the interested clients otherwise, the next flush time is calculated and the alarm is set to ring at that particular time.

```
public static synchronized void checkFlush(int objid, String
                                          eventType, NotifRecord
                                          notifRecord){

  if (notifRecord.timeToFlush()) { // if time to flush queue is
                                      reached
    flushQueue(objid,eventType,notifRecord);}   // send output
    else { // this happens when events are added to an empty
             queue
      // calculate the next time to flush the queue
      long nextTime = notifRecord.nextFlushTime();
      alarm.set(nextTime); }        // reset alarm to nextTime
}
```

`flushQueue()`removes an event from a queue whose flush time has been reached and sends it out to the interested clients.

```
public static synchronized void flushQueue(int objid, String
                                          eventType,NotifRecord
                                          notifRecord) {

  String data = notifRecord.removeQueue(); // remove event from
                                             queue

  // send output to all interested clients
  ServerConnectionHandler.sendTo(objid,notifRecord.recipient.toObj,
                                 notifRecord.recipient.toHost,
                                 eType,data );
 }
```

8.4.3.2 Alarm process

The alarm process calls a callback that continuously polls to check if an event is ready to be flushed.

```
public static class MyAlarmCallback implements AlarmCallback {

  public void ring() {
    flushonTime();}
}
```

Only one alarm process is actually used for each non-empty queue. However, each non-empty queue maintains a separate deadline for its flush time. `flushonTime()` checks each recipient's flush time deadline. If the flush time has been reached the queue is flushed otherwise, the alarm is set to the closest of the relevant alarm deadlines.

```
public static synchronized void flushOnTime(){
  set nextTime to Long.MAX_VALUE; //set nextTime to a large value
  for each object in the interestTable(objid, eventType) {
    for each object in the recipient set {
      nrNextTime = get notifRecord.nextFlushTime();
      if nrNextTime <= current time { // if flush time is reached
        flushQueue(objid, eventType, notifRecord);}//flush queue
      // otherwise calculate next time to flush
        else if (nrNextTime <= nextTime) {
          set nextTime to nrNextTime;}
    }
  }
  alarm.set (nextTime); // set alarm to ring at nextTime }
```

The alarm is reset when events are added to an empty queue or after a non-empty queue is flushed due to a previous alarm.

8.4.4 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. GtK allows a client to change the frequency with which it wants to be notified of any updates.

The client object sends a "`change frequency`" event together with the new values for the pace parameters `queueLength` and `time`. This will also alter the frequency of each recipient for that client object.

8.4.4.1 Change frequency

`changeFrequency()`updates the frequency record for the client object with the new values of `queueLength` and `time`. It also alters the frequency of each recipient for that object and subsequently checks to see if it is time to flush the queue.

```
public static synchronized int changeFrequency(int objid, String
                          eventType, int clientid, int remobjid,
                          int bufferLength, int time) {

  recipient set for Object = interestTable.get(objid, eventType);

  if null {          // first time object referred to
    return 1; }   // return error

  for each object in the recipient set {

    get NotifRecord();

    if ( NotifRecord.Recipient.toHost == clientid &&
                  NotifRecord.Recipient.toObj == remobjid) {
      // build new frequency object

      create new Frequency(queueLength,time);
      // change old frequency

      notifRecord.changeFrequency(Frequency);

      // check if it is time to flush queue

      checkFlush(objid,eventType,notifRecord);
    }
  }
  return 0;
}
```

8.5 Example real-time online conferencing application

The previous sections have shown how the Event Manager and the Notification Manager components have been implemented on the GtK framework. The last component on the distributed layered infrastructure (starting from the base in figure 8.1) is the Application. The Application does not execute as a stand-alone component. It interacts closely with the Notification Manager through the Event Manager. This interaction is essential for any purpose-built application to function properly. Also, the Notification Manager needs the relevant information from the Application in order to broadcast information to the users at the right pace and right granularity.

An example real-time online conferencing application has been constructed on the GtK infrastructure (figure 8.5). The application allows users to create conferences on various topics and launch sessions on one or more conferences with several participants simultaneously. The system supports live discussions but also enables late joiners to catch up on any ongoing sessions.

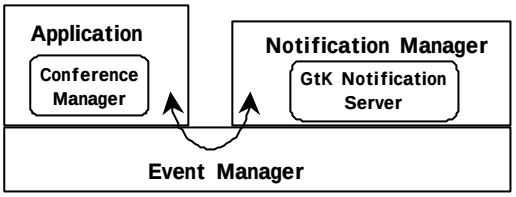


Figure 8.5 Conferencing exemplar on GtK infrastructure

The Conference Manager interacts with the client objects and the Notification Manager by exchanging events and messages through the Event Manager. Throughout the discussion presented in this chapter, the terms message and event seem to convey the same meaning. However, there is a subtle distinction between them.

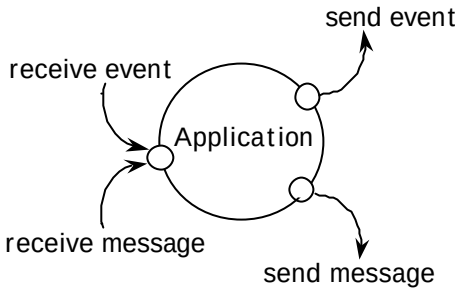


Figure 8.6 Event vs. message

A message is usually directed towards a specific object whereas an event announces the presence of a certain object. An application can be regarded as being in the centre receiving and sending messages and events (figure 8.6). Both events and messages have the same format and they are received through the same mechanism, but they differ in the way they get sent out. Figure 8.7 shows the flow of messages and events between the different components in the example conferencing system.

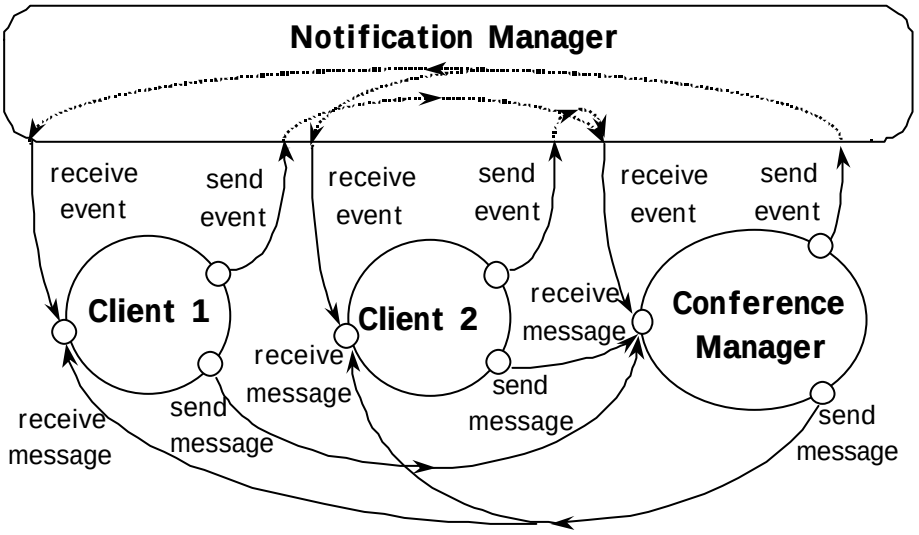


Figure 8.7 Event and message exchange in conferencing exemplar

Messages are events that are sent directly from one object to another and the sender object does not receive a feedback from the receiver object. For example, the client objects send and receive messages directly to and from the Conference Manager.

Events instead only announce the presence of an object and they are mediated through the Notification Manager. For example, client objects register with the GtK notification server by sending an event to the Notification Manager. The latter then responds to the user clients by sending some feedback, such as a welcome note. Similarly, the Conference Manager can send events to the client objects via the Notification Manager, for instance through the 'tell all' function. The Notification Manager responds by sending broadcast events to the user clients.

8.6 Summary

This chapter described how the GtK experimental notification server has been implemented over several layers of custom and standard infrastructure. The functionalities of each layer and their implementation details were examined. A distributed asynchronous protocol is used for exchanging messages and events within the GtK infrastructure. The Event Manager allows point-to-point asynchronous messaging between different communication objects in the same or in different address spaces.

The Notification Manager manages the main functions of the GtK notification server. The distributed object layered infrastructure enables the Notification Manager to be aware of every other object. Client objects can either add or remove interests for different objects with GtK. GtK maintains a list of interested clients for specific objects and their recipients. When changes occur to certain objects, GtK broadcasts notification events to the respective recipients.

The way in which GtK has been augmented to provide pace impedance matching was also analysed. Two pace parameters are introduced to control the frequency of updates – the queue length and maximum delay time. The 'add interest' and 'remove interest' functions were subsequently modified to include each recipient's frequency of notification. Furthermore, a FIFO queue of outstanding events is maintained for each recipient. The events are flushed and broadcast to the recipients at the relevant time depending on the pace parameters. An alarm process sets and resets the maximum delay time parameter.

Users often interact with different interface objects at a non-uniform pace. A client object can change the frequency which with it wants to be notified by sending a 'change frequency' event to GtK together with the updated pace parameters. GtK also modifies the frequency of each recipient for that client object accordingly.

An example real-time Web conferencing application has been constructed on the GtK framework. The exchange of messages and events between the different components of the application were finally considered. The example application is similar to many Web-based chat systems. However, its main purpose is to demonstrate the practicality of the GtK

separable notification server as a pace impedance matcher and this will be dealt with in the next chapter.

Chapter 9 Demonstration through an Exemplar

The GtK infrastructure described in Chapter 8 provides a framework for building applications that support pace impedance matching by using a separable notification server. Chapter 6 showed how a 'pure' notification server separates the concerns of data from notification and Chapter 7 emphasized that such a notification server could in fact be used for performing impedance matching. As argued in Chapter 2, the interface affords an effective user-level behaviour when users receive feedback and feedthrough information at a rate that matches their pace of interaction. The analysis in Chapter 5 also reinforced the need for timely feedthrough of information to effectively support users engaged in distributed collaborative work.

This chapter describes how an example real-time Web conferencing system has used the GtK framework to provide collaborative users with an interface that matches the rate of feedthrough they receive with their pace of interaction. This demonstration acts as a technical evaluation of the GtK framework. Although the evaluation is unlike traditional forms of evaluation, it does provide a critique of the framework. This chapter deals with aspects related to the construction of the real-time Web conferencing exemplar on the GtK framework. It applies some of the issues discussed in Chapter 7 and uses the implementation details described in Chapter 8. The next chapter complements this assessment by evaluating the framework from an architectural viewpoint.

Section 9.1 gives a rationale that justifies why a conventional evaluation method has not been applied to the GtK framework. Section 9.2 describes the behaviour that users receive at run-time when they interact with the different functionalities of the example Web conferencing application. Section 9.3 then shows how the example application has been implemented on the GtK framework to support its interface behaviour. Finally, Section 9.4 examines the way in which the example application uses the GtK notification server to provide collaborative participants with a pace of feedthrough that matches their interest levels.

9.1 Evaluation criteria

There are usually two motivations for carrying out an evaluation. The first aims at demonstrating the advantages of a particular concept, idea or artefact and proving that the artefact is an improvement over what was previously available. The second motivation is to bring out the disadvantages of an artefact, with a view of identifying those aspects that require further work.

The example real-time Web conferencing application that has been developed is similar to many Web-based chat applications, although its novelty lies in providing users with a controlled pace of feedthrough. However, the main purpose for implementing the example

application was to demonstrate the practicality of the GtK notification server as an impedance matcher within the GtK framework.

It is often problematic to evaluate a toolkit or a framework embodied in code. If the aim of the evaluation is to show that the toolkit produces good applications, then multiple applications should be built, not once but several times. In addition, third party users should be involved in the process of building those applications. But none of these techniques have been employed here. Indeed, it would have been impractical to cover such an in-depth evaluation within the scope of this research.

The focus of this research lies on gaining a deep understanding of how applications can be built to provide a desirable temporal behaviour within a distributed collaborative setting and hence facilitate user cooperation. In order to meet this goal, the emphasis throughout this work has been on the architectural aspects of application building through a number of analytical studies.

The findings of those studies have enabled the development of the GtK framework that enables impedance matching. The conferencing application described through the rest of this chapter merely acts as an exemplar that demonstrates the feasibility of pace impedance matching in improving the temporal behaviour that user receives at the interface level.

9.2 Interface behaviour

This section describes the behaviour of the example real-time Web conferencing system from the users' perspective. It provides a system walkthrough that analyses the visible parts of the user interface and shows how users interact with the different functionalities offered by the interface.

The conferencing application allows users to create conferences on several topics and launch discussion sessions with different participants at the same time. The discussion sessions are mainly held in real-time but late joiners can also catch up with any ongoing session. The application offers functionalities that are common to most Web-based chat systems. However, the novel feature of the conferencing application is its ability to enable users to interact with multiple conferences simultaneously while adjusting the pace of feedthrough to match the users' rate of interests.

9.2.1 Connect to application

Like many Web-based chat applications, the nature of the interaction pushes towards the use of applets for maintaining conversations. Users connect to the conferencing application that runs on a server through an applet interface from any common Web browser.

A user launches a conference client by typing the correct URL on a Web browser. Following this action, a typical applet is downloaded on the user's screen (figure 9.1).

Figure 9.1 Typical client applet

The `Message` area displays feedback information in response to users' actions. For instance, when a user first connects to the server, a message notifies her if the connection is successful or not.

9.2.2 Register with application

Let us assume that Jane has just launched a client applet on her screen. Jane registers with the conferencing application by first typing in her name in the `User name` field and then clicking on the `Connect` button (figure 9.2).

Figure 9.2 User registration

If Jane is successful in registering with the server, she can proceed to create and join any conference. Note, if other users are already logged on to the system and created some

conferences prior to Jane's registration, a list of the existing conferences would be displayed under the `Conference List` at this stage (see figure 9.3).

9.2.3 Create new conference

If Jane wants to create a new conference called `JAVA`, she simply has to type in the conference name in the `New Conference` field and click on the `Create` button (figure 9.3). The `Conference List` is subsequently updated and sent out to all the participants who are logged on to the system.

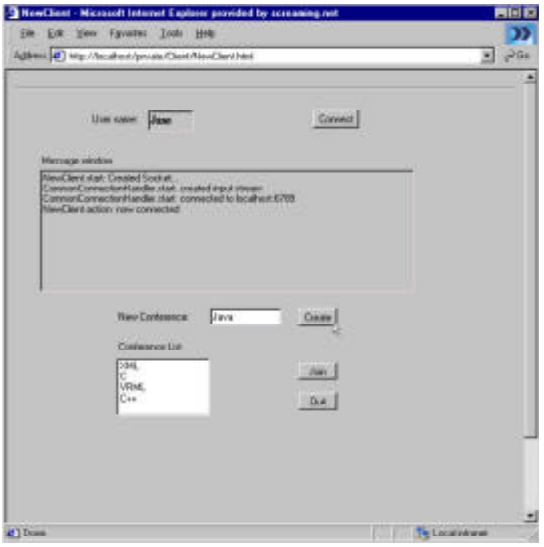


Figure 9.3 Create new conference

9.2.4 Join conference

Jane joins a conference by first selecting its name from the `Conference List` and then clicking on the `Join` button (figure 9.4).

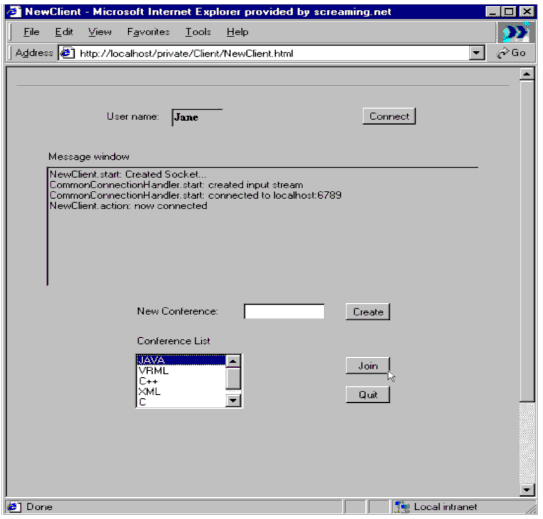


Figure 9.4 Join conference

A new window labelled with the conference name pops up on Jane's screen followed by a welcome message `"Hello Jane"`, which is displayed on the `'Users contribution'` area (figure 9.5). The lower `'public chat'` text area accepts users input while the upper `'Users contribution'` message area displays all the participants' contributions to that particular conference.

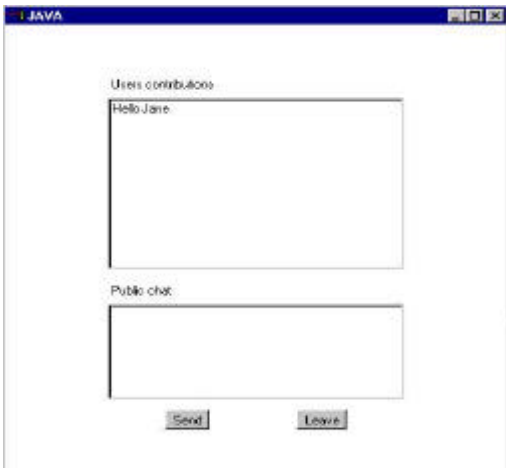


Figure 9.5 Pop-up conference window

Soon after Jane is active on the `JAVA` conference, Tom decides to join in. Jane subsequently receives the message `"Tom has just joined"` which informs her of Tom's presence (figure 9.6).

9.2.5 Add contribution

Users can add their own contributions to a conference by typing in some text and clicking on the `Send` button. The contributions are then broadcast to all the participants logged on to the same conference.

Figure 9.6 Add contribution

9.2.6 Interact with multiple conferences

Users can participate in more than one conference simultaneously. For each conference, the user needs to select the conference name from the `Conference List` and then click on the `Join` button (figure 9.4). A separate conference window pops up on the user's screen each time. Users may therefore receive several overlapping windows, each representing a particular conference contribution.

Figure 9.7 shows two overlapping conference windows for Tom, one representing the `XML` conference and the other showing the `JAVA` conference. Although it appears that Tom is active on both conferences, his focus in fact lies on a specific conference at any instance in time.

In any windowing system, the top-most window indicates the user's focus as it has keyboard control. From this we can deduce that, Tom's focus is on the `XML` conference, given its window position. Hence, any new contributions to the `XML` conference are likely to be of more interest to Tom. The `JAVA` conference window is instead in the background, which implies that Tom only has a passive interest in the changes to that conference. A similar reasoning was applied when investigating potential scenarios for impedance matching in Chapter 7 (Section 7.5.2).

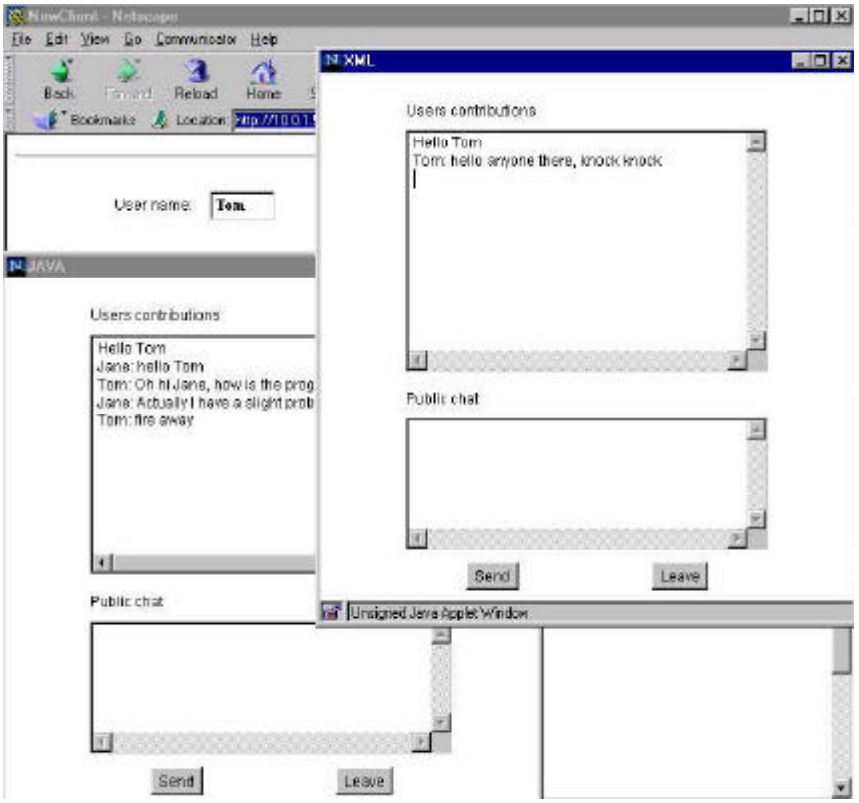


Figure 9.7 Overlapping conference windows

Tom therefore receives changes to the `XML` conference immediately after a new contribution is added. This high rate of feedthrough may in some cases be restricted by the network

latency. Tom does not necessarily receive new contributions to the JAVA conference as soon as they are sent out.

However, if Tom decides to shift his focus back to the JAVA conference or wants to catch up on the thread of conversation going on there he simply has to click on the title bar of the JAVA conference window at any time, to bring it into focus. This action immediately increases the pace of feedthrough that Tom was getting for the JAVA conference. Consequently, any outstanding contributions are displayed straightaway on Tom's JAVA conference window. Furthermore, Tom will receive new contributions to the JAVA conference almost as soon as they are posted but any new contributions to the XML conference will be communicated to him at a much lower pace. The implementation issues surrounding pace frequency are described in Section 9.4.

9.2.7 Leave conference

A user can leave a conference at any time by clicking on the Leave button on the conference pop-up window (figure 9.8). After Jane leaves the JAVA conference, the conference window closes on her screen. Shortly after, Tom and any other partcipants on the JAVA conference receive the message "Jane has just left" on their screen, which informs them of Jane's departure (figure 9.9).

Figure 9.8 Leave conference

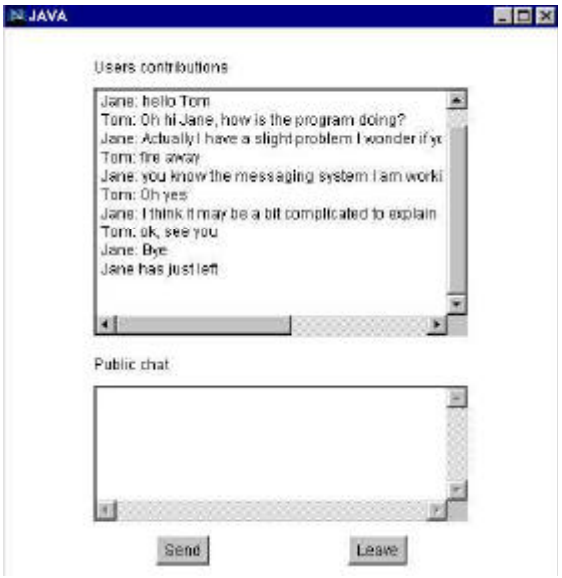


Figure 9.9 Notification of departure

9.2.8 Quit application

Users can quit the conferencing application by simply clicking on the "Quit" button on the main window of the client applet (figure 9.1). This action closes down the applet window. Moreover, users do not have to explicitly leave the conferences they had joined before quitting the application. For example, if Jane quits the application before leaving the JAVA conference, Tom and any other partcipants on the JAVA conference would still be informed that she has left.

This section has demonstrated the behaviour of the example real-time Web conferencing application at run-time. It has also examined the effect of users' simultaneous participation on multiple conferences and shown how the pace of feedthrough is adjusted to match their focus. The next section will now describe how the underlying GtK framework supports the application's interface behaviour and assists in implementing its various functionalities.

9.3 Application implementation

The application is deployed as a number of client objects, which run as applets on a user's Web browser (Section 9.2) and a Conference Manager that executes on a server and uses GtK for notification purposes. The Conference Manager is implemented at the Application level of the GtK framework and sits on a server (possibly remote) alongside the Notification Manager (Section 8.5). The Conference Manager manages the conferences created by the users.

The following discussion will use some of the method calls already described in Chapter 8 and will also build on them to show the interaction between the Conference Manager and the Notification Manager through the Event Manager. The interaction is illustrated as a series of events and messages exchanged between the different components. The distinction between events and messages was established in Chapter 8 (Section 8.5).

9.3.1 Connect to Conference Manager

The server runs on a pre-defined port and starts up the Conference Manager and the Notification Manager by registering them as event handler objects with the Event Manager (Section 8.2.2.2). A unique identifier is assigned to the Conference Manager and the Notification Manager respectively.

```
// call from main() in NewServer.java

   // add Notification Manager event handler with id = 99
   ServerConnectionHandler.addObject(99, notifier)
   // add Conference Manager event handler with id = 999
   ServerConnectionHandler.addObject(999, conference)
```

The server then carries on listening for connections from client objects.

9.3.2 Register with Conference Manager

When a user connects to the application (Section 9.2.2), the client object registers itself with the Notification Manager through an `"add interest"` event type (Section 8.3) and asks the Notification Manager to be told about the Conference Manager by sending its identifier `CONF_MGR = 999` as part of the data packet, `d`. This is represented as event ❶ in figure 9.10.

```
// call from public instance method action() in Client.java
```

```
theSharedData.ch.sendTo(confId, NOT_MGR, notHostId, "add
                        interest",d.format())
```

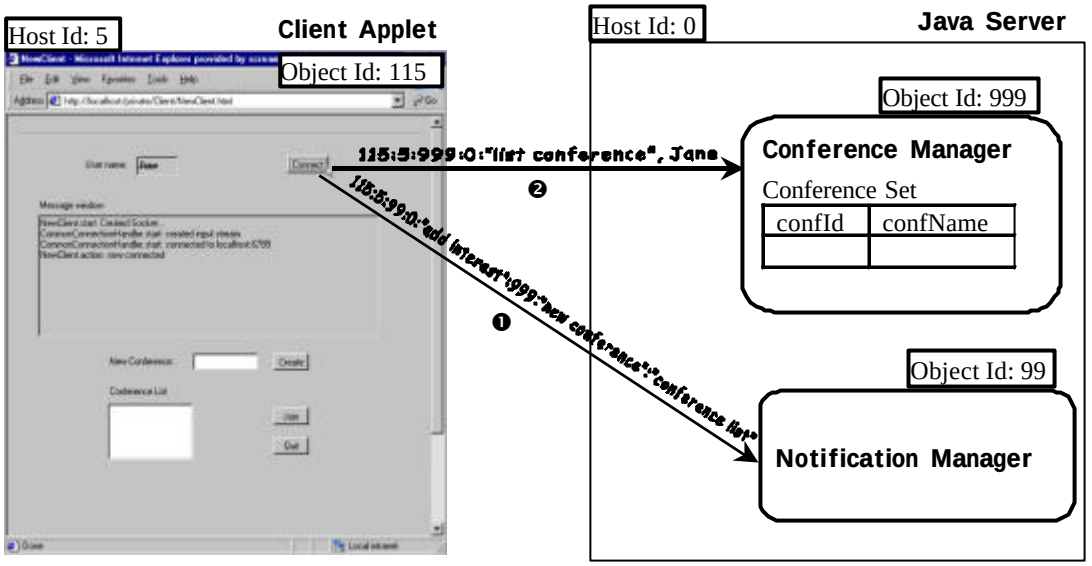


Figure 9.10 Client object registers with Conference Manager

The client object also asks the Conference Manager to list any conferences, if some have already been created, by sending out message ❷ with a "`list conference`" event type.

```
// call from public instance method action() in Client.java
   theSharedData.ch.sendTo(confId, CONF_MGR, confHostId, "list
                           conference", theSharedData.userName)
```

On receiving the "`list conference`" event type, the Conference Manager first checks to see if the conference set is not empty. If that is the case, the Conference Manager then sends out a "`conference list`" event type and a data packet `d`, which contains the conference names and their identifiers, to the client object (figure 9.11).

```
// call from public instance method listConference() in Server
   ConferenceManager class
public static synchronized void listConference(int remoteObjid,
                                               int clientid) {
   // build new conference list and send list to clients
   create new BuildConferenceList(conferenceSet);
   DataPacket d = buildList.getList();// get list into data packet
   ServerConnectionHandler.sendTo(CONF_MGR,remoteObjid,clientid,
                                  "conference list", d.format());}
```

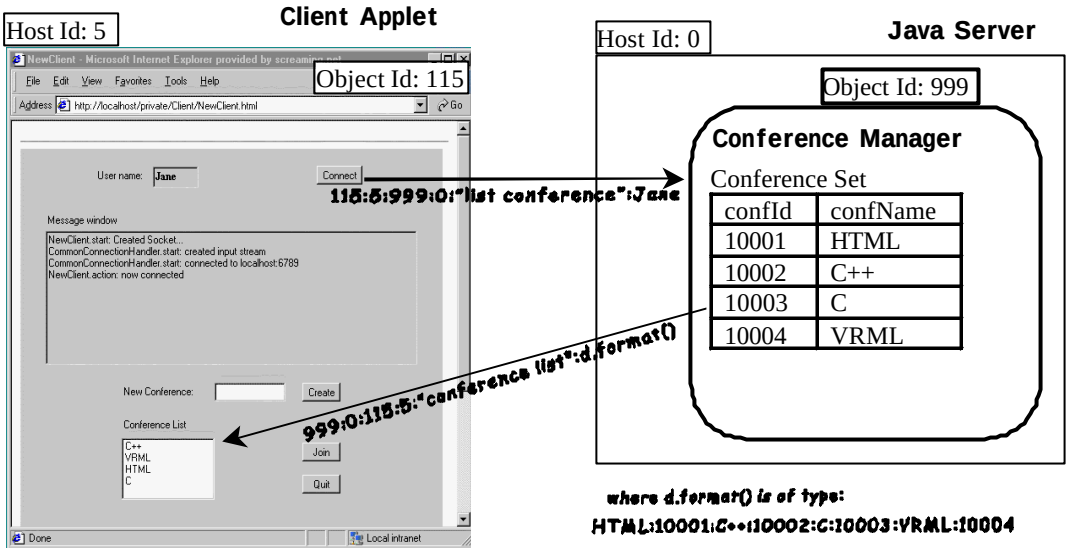


Figure 9.11 Conference Manager sends conference list to client object

9.3.3 Create new conference

When a user creates a new conference (Section 9.2.3), the client object sends message ❶ (figure 9.12) to the Conference Manager with a `"new conference"` event type and the conference name.

```
// call from public instance method action() in Client
   ConferencePanel class

      mySharedData.ch.sendTo(confId, CONF_MGR, confHostId,
                             "new conference",confName)
```

If the conference name does not already exist in the conference set, the Conference Manager proceeds to create the new conference.

`createConference()` registers the conference name as an event handler, which automatically generates a new conference identifier (Section 8.2.2.2). The conference name and its identifier are added to the conference set and thereafter rebuilt into a data packet. The Conference Manager then tells the Notification Manager to broadcast the updated conference list to all the users by sending message ❷ with a `"conference list"` event type through the `tellAll()` method call (Section 8.3.3).

```
// call from public instance method createConference() in Server
   ConferenceManager class

public static synchronized void createConference(String
                          confName,int remoteObjid,int clientid) {

   create new TranscriptObject(confName); // create new event
                                          handler for conference

   // return automatically generated conference identifier

   confId = ServerConnectionHandler.addObject(TranscriptObject);

   add conference TranscriptObject to conferenceSet;

   create new BuildConferenceList(conferenceSet); // build new
                                                  conference list

   DataPacket d = buildList.getList(); // get list into data
                                       packet

   // broadcast updated conference list to all the clients

   NotificationManager.tellAll (CONF_MGR, "conference list",
                                d.format());

}
```

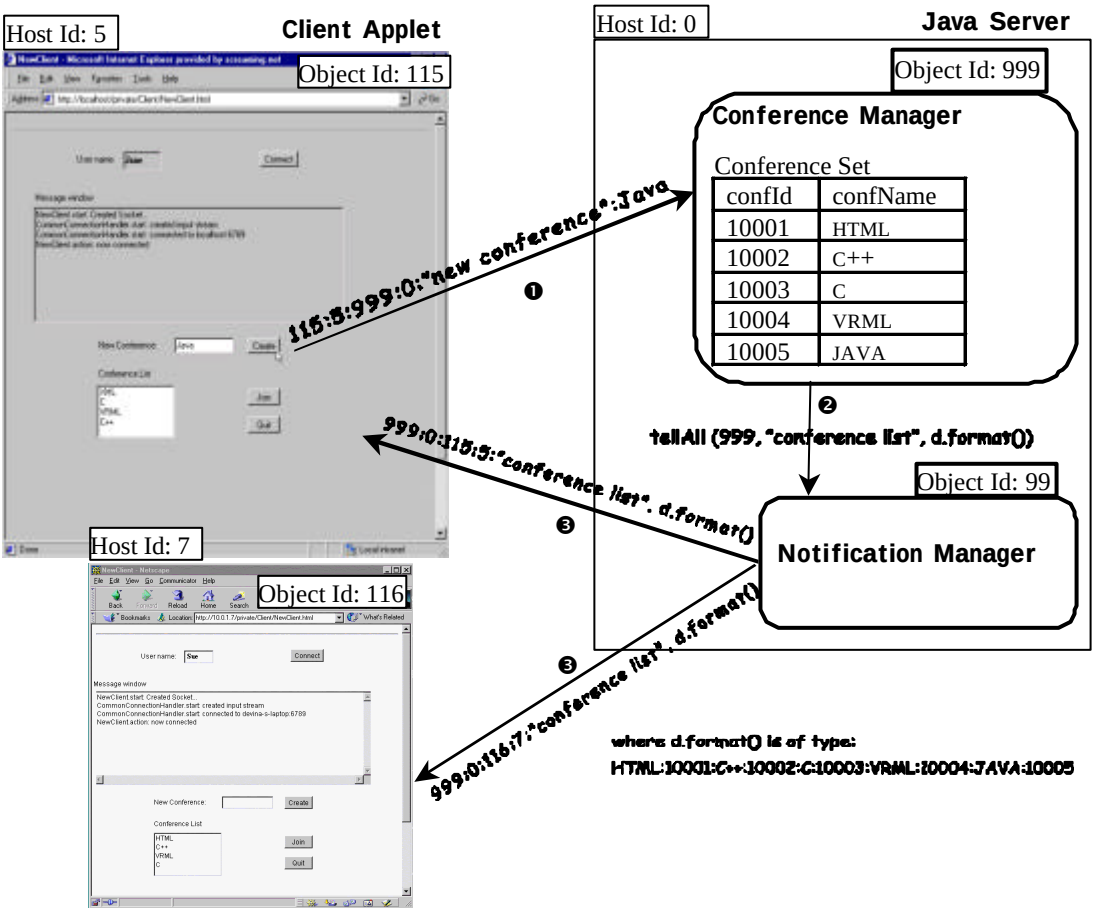


Figure 9.12 Create new conference and broadcast updated list

On receiving that request, GtK uses the `tellAll()` method (Section 8.3.3.1) to broadcast the updated conference list to all the registered clients with a `"conference list"` event type (events ❸ in figure 9.12). As each client maintains its own copy of the conference table that stores the conference names and their identifiers, the `"conference list"` event type triggers them to update their local copy of the conference table.

Each client applet then refreshes the conference list locally to reflect the change. By maintaining a local copy of the conference table, client objects do not need to query the notification server if they require any conference details, thus reducing network exchange.

9.3.4 Join conference

Consider the example of Jane joining the `JAVA` conference (Section 9.2.4). Jane's client object launches a new frame on the screen with the conference name as its title. The new client conference object has its own identifier allocated to it, but it still runs on the same host as the client object.

```
// call from public instance method action() in Client
   ConferencePanel class

   // find selected conference name from the conference list
   String selectedConf = mySharedData.confList.getSelectedItem();
   if (selectedConf != null) {// find conference id from local
                              conference table
     selectedConfId = myconferenceTable.getConfId(selectedConf,
                                                  mySharedData);
     mySharedData.theTranscriptList.createNewFrame(selectedConfId)
                            ; // create new conference window
   }
```

The client object sends message ❶ (figure 9.13), which includes the conference object identifier and `"new client"` event type, to the Transcript object on the Conference Manager. Each conference is associated with a Transcript object, which is responsible for monitoring the interaction between different clients registered to a particular conference.

The client object also sends event ❷ to the Notification Manager with an `"add interest"` event type, the client conference object identifier and a whole list of other interested event types, as part of the data packet, to make GtK aware of the Jane's interest in the `JAVA` conference.

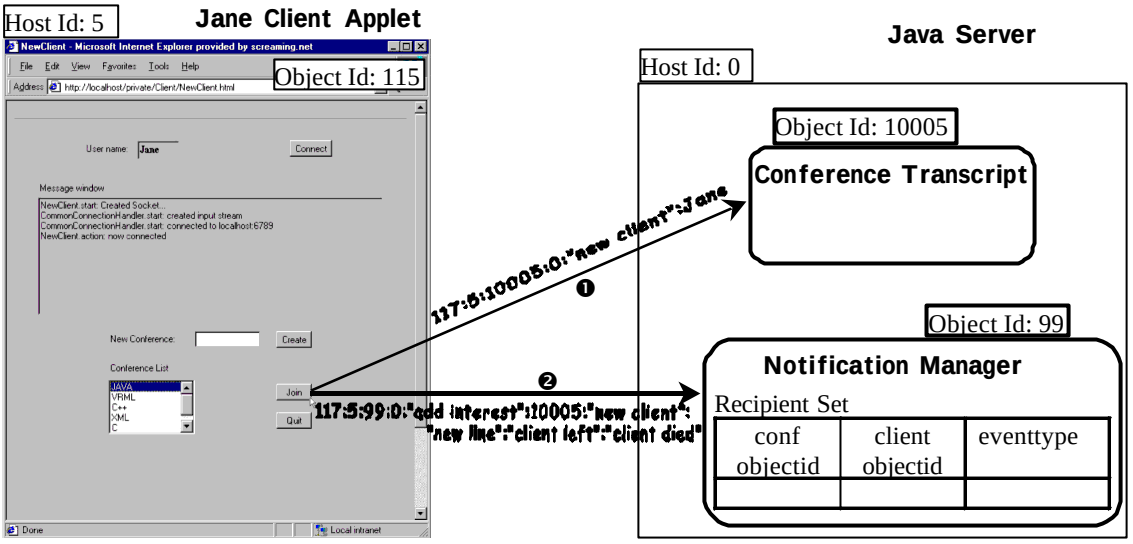


Figure 9.13 Join conference

```
// call from public instance method setTranscript() in Client
   TranscriptPanel class

   // send "new client" event and client name to Transcript object
      on Conference Manager

   mySharedData.ch.sendTo(myId, toObj, toHostId, "new
                          client",mySharedData.userName);

   // add interest with Notification Manager

   mySharedData.ch.sendTo( myId, NOT_MGR, notHostId, "add
                          interest",d.format() );
```

The "add interest" event type triggers GtK to add Jane's interest in the JAVA conference through the addInterest() method (Section 8.3.2.1). This creates a new recipient for the JAVA conference with Jane's conference client object identifier and its related event types. The recipient is then added to the recipient set (figure 9.14).

The Transcript object then tells the Notification Manager to notify Jane's presence to any other clients logged on the JAVA conference by sending event ❸. At the same time, the Transcript object acknowledges Jane's presence by sending message ❹ with a "greeting" event type to her conference client. The latter interprets the event type locally to display a welcome message, 'Hello Jane' on her JAVA conference window.

GtK uses the tellAll()method (Section 8.3.3.1) to find the list of interested clients (recipients) for the JAVA conference by matching the event type and object identifier in the recipient set. Figure 9.14 shows the case when only Jane is active on the JAVA conference, therefore GtK does not need to notify her presence to any other users.

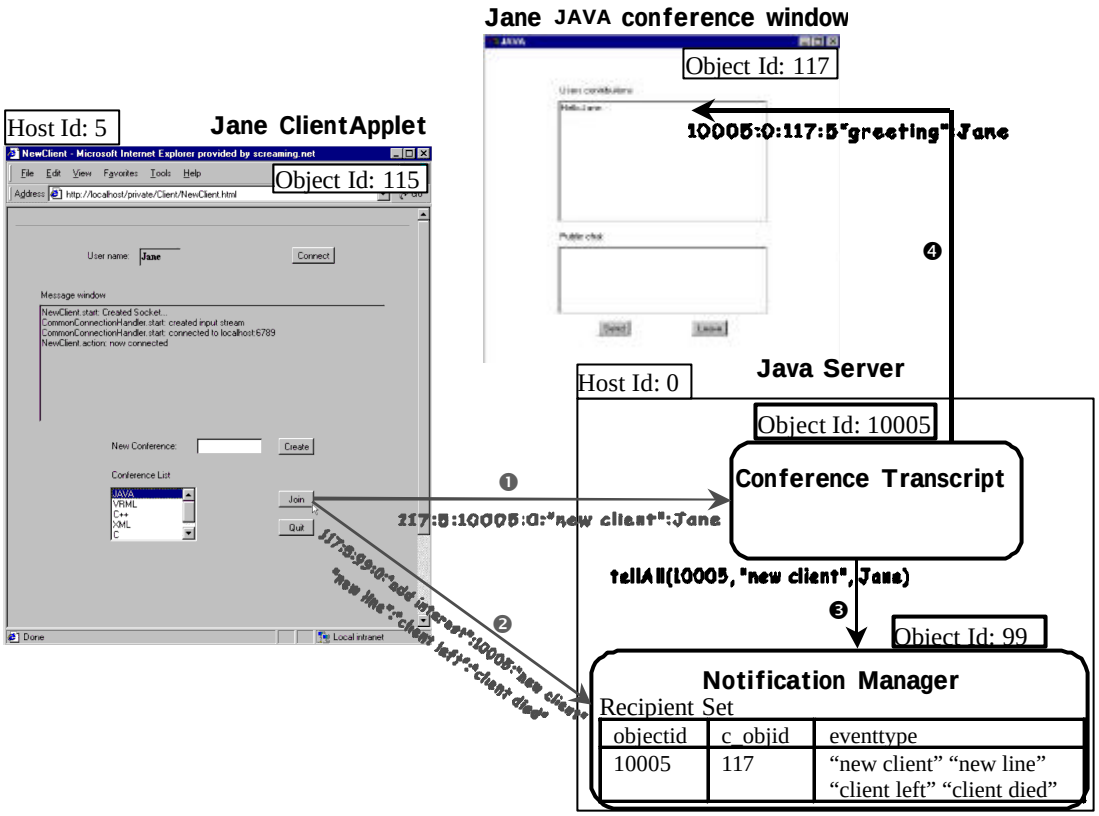


Figure 9.14 Send greeting message

However, if Sue decides to join the JAVA conference shortly after, the sequence of event and message flows changes to those shown in figure 9.15.

Sue's client registers with the Transcript object by sending message ❶ and with the Notification Manager though event ❷. The Notification Manager updates the recipient set with Sue's conference client's details. The Transcript object then tells the Notification Manager to notify Sue's presence to any other clients on the JAVA conference with event ❸, while it responds to Sue's conference client object with message ❹.

GtK responds to the Transcript object request for broadcast by searching through the recipient set to find the list of interested clients, which in this case is Jane. GtK sends event ❺ to Jane's conference client object with a "new client" event type and Sue's client name. Jane's conference client object translates the "new client" event type locally by displaying the message 'Sue had just joined' on her JAVA conference window.

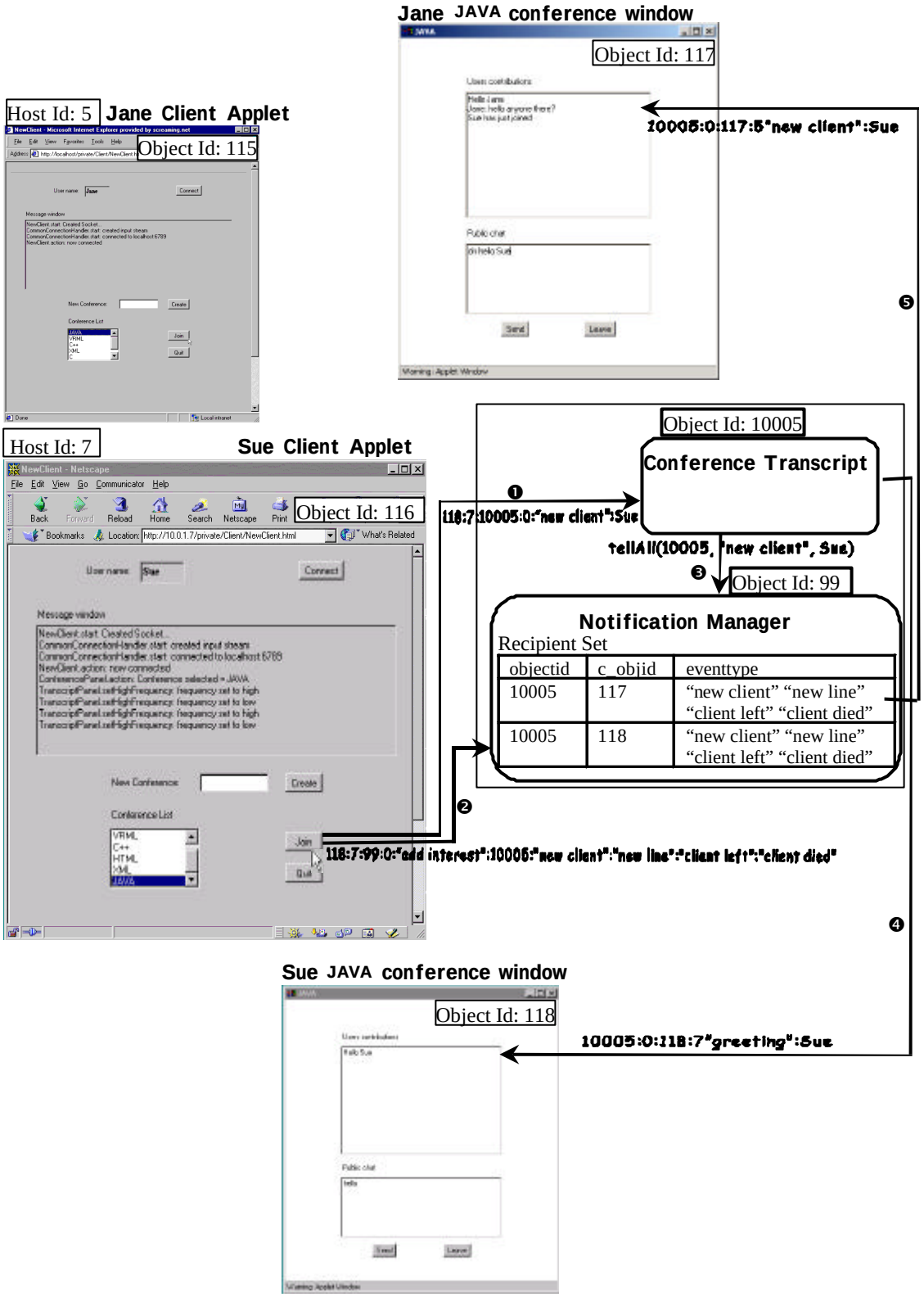


Figure 9.15 Another user joins conference

9.3.5 Add contribution

When Jane adds some contribution (Section 9.2.5) to the JAVA conference, her conference client object sends message ❶ (figure 9.16) to the Transcript object on the Conference Manager with a "new line" event type and the text as the data packet.

```
// call from public instance method action() in Client
   TranscriptPanel class

// send "new client" event and input line to Transcript object
mySharedData.ch.sendTo(myId, toObj, toHostId, "new line",line);
```

The Transcript object then tells the Notification Manager to broadcast Jane's input through event ❷. If Jane is the only user who is active on the JAVA conference, GtK responds by sending event ❸ back to her conference client object with the input text and a "new line" event type. As a result, Jane's conference client object displays the text on her JAVA conference window.

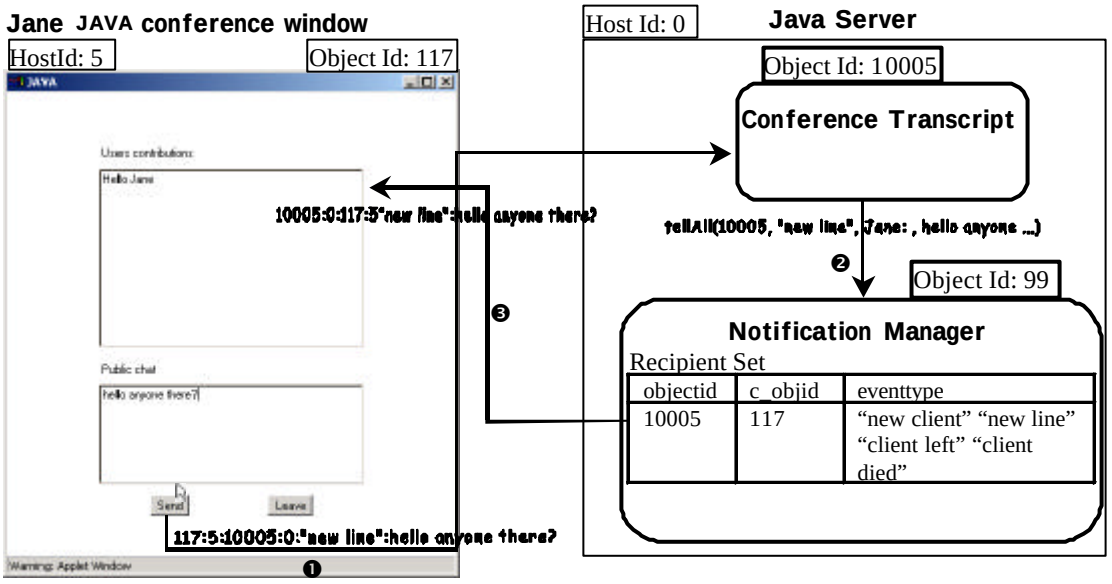


Figure 9.16 User adds contribution

But if Sue is also active on the JAVA conference, GtK will broadcast Jane's input to Sue's JAVA conference window simultaneously. Figure 9.17 shows the flow of messages and events between the client objects, the Conference Manager and the Notification Manager when both Sue and Jane are conversing on the JAVA conference.

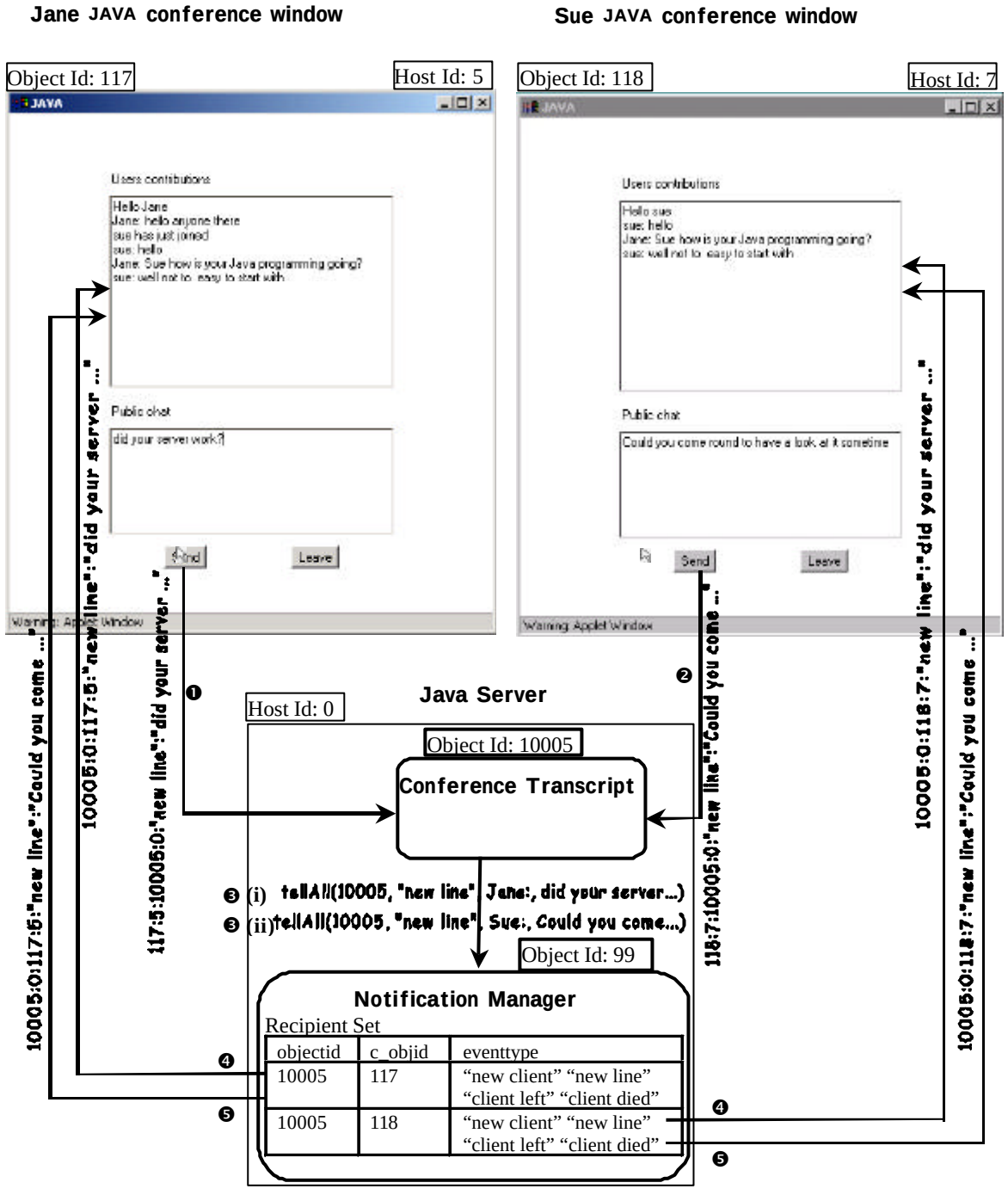


Figure 9.17 Contributions from multiple users

Because messages are exchanged between more than one client object, the messages may overlap, thus giving rise to the possibility for race condition. Chapter 7 considered an example where race conditions can occur as a result of the incorrect ordering of messages (Section 7.6.3). However, in the conferencing exemplar, the data is managed centrally and this at least ensures that the Notification Manager broadcasts the users input in the same order as it received them.

9.3.6 Leave conference

When Jane leaves the JAVA conference (Section 9.2.7), her conference client object sends message ❶ (figure 9.18) with a "client left" event type to the Transcript object on the Conference Manager. In addition, the conference client object sends event ❷ to the Notification Manager with a"remove interest" event type plus a list of the client's registered event types enclosed in a data packet to inform the Notification Manager that the client no longer has an interest in that conference.

```
// call from public instance method leave() in Client
   TranscriptPanel class

   // send "client left" event to Transcript object

   mySharedData.ch.sendTo(myId, toObj, toHostId, "client
                          left","leave");

   // tell Notification Manager to remove interest

   mySharedData.ch.sendTo(myId, NOT_MGR, notHostId, "remove
                          interest",d.format() );
```

GtK subsequently removes Jane's interest in the JAVA conference from its recipient set through the removeInterest() method (Section 8.3.2.2).

Furthermore, on receiving the "client left" event type, the Transcript object on the Conference Manager removes Jane's conference client object and tells the Notification Manager to inform all clients on the JAVA conference that Jane has just left through event ❸.

```
// call from public instance method personLeft() in Server
   TranscriptObject class

   clientNames.remove (hostid); // remove hostid from the
                                   ClientNames table

   // tell NotificationManager to inform all interested clients of
      the client's departure

   NotificationManager.tellAll(myId, "client left", client);
```

GtK uses the tellAll() method (Section 8.3.3.1) first, to find the list of clients that need to be informed and second, to broadcast event ❹ with a "client left" event type to the relevant conference client objects.

When Sue's conference client object receives the "client left" event type, it interprets it locally and displays the message 'Jane has just left' on Sue's JAVA conference window.

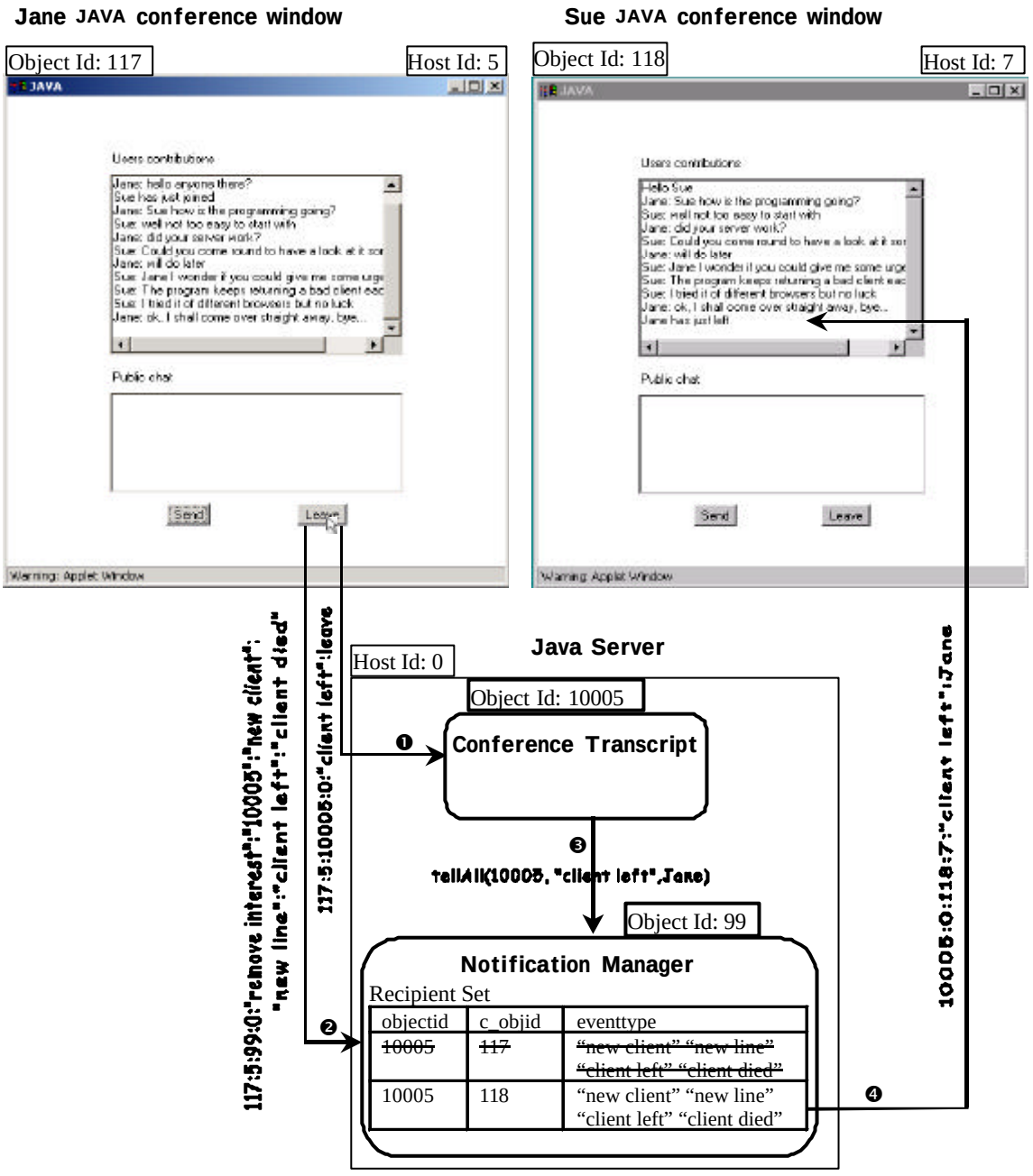


Figure 9.18 Leave conference

9.3.7 Quit application

When a users quits the conferencing application (Section 9.2.8), the client object sends a message to the server Connection Handler with a `"quit"` event type, which tells the server to stop listening on the user's client socket. At the same time, the client object sends an event to the Notification Manager with a `"remove interest"` event type. This triggers GtK to remove any interest that the client object may still have on any other objects in the recipient set through the `removeInterest()` method.

```
// call from public instance method action() in Client
   TranscriptPanel class

   // tell Server Connection Handler that client has left the
      application, line = "quit"
   mySharedData.ch.sendTo(confId, CONF_MGR, confHostId, "leave
                          conference",line );
   // tell Notification Manager to remove user interest
   mySharedData.ch.sendTo(confId, NOT_MGR, notHostId, "remove
                          interest",d.format() );
```

This section has described how the main functionalities of the example conferencing system have been implemented at the application level through the GtK framework. The following section will now discuss how users who are conversing on multiple conferences can receive a rate of feedthrough that matches their focus or rate of interests.

9.4 Pace controlled feedthrough

GtK uses pace impedance matching to manage the rate of feedthrough that users receive when they interact with several conferences simultaneously. This section will examine how pace impedance matching is actually implemented in the example conferencing application by applying the method calls in Chapter 8 (Section 8.4). The interface behaviour was shown in Section 9.2.6.

9.4.1 Set frequency levels

The provision of feedthrough information is based on the user's focus on a particular conference at any instance in time. The example real-time Web conferencing application uses two frequency levels:

(a) high-level frequency

A high pace of feedthrough is used for the top-most conference window and the client object requests instant feedthrough (limited by network latency), corresponding to the default pace parameters for `queueLength` and `time` (Section 8.41) namely, `queueLength = 0` and `time = -1`.

(b) low-level frequency

When a conference window is moved to the background this is detected by the client object which sets a lower pace of feedthrough by using non-zero `queueLength` and `time` pace parameters.

In order to associate the right frequency with each conference window, the users' focus need to be tracked.

9.4.2 Track users focus

Java generates a `GOT_FOCUS` event if a particular window is in focus and a `LOST_FOCUS` event if the window is in the background. The client object therefore monitors the Java event generated by each conference window. As the users' focus changes from one conference to another, the frequency level changes accordingly.

If a conference window triggers a `GOT_FOCUS` event, the client object uses the `setHighFrequency()` method call to set a high-level frequency to the conference. Instead, if a conference window triggers a `LOST_FOCUS` event, the client object first double checks to see if that window is still inactive after a short time delay, by setting up an alarm that wakes up after the delay has elapsed. This safeguards against a user accidentally clicking on a window. If the conference window is still in the background, the client object uses `setLowFrequency()`method call to set a low-level frequency to the conference.

When a background window is brought to the front, the client object resets the conference to a high-level frequency with default pace parameters.

9.4.3 Register pace interest

In order to enable GtK to provide a controlled pace of feedthrough, client objects have to register different frequencies of pace interest with the Notification Manager through a `"change frequency"` event type. The frequency level is transmitted as part of the data packet, `d`.

```
// call from public instance method doSetFrequency() in Client
   TranscriptPanel()class

// change frequency with new values for queuelength:time
   mySharedData.ch.sendTo(myId, NOT_MGR, notHostId, "change
                          frequency",d.format() )
```

The `"change frequency"` event type triggers GtK to change the client's object pace of feedthrough to the relevant frequency level through the `changeFrequency()` method (Section 8.4.4.1). The frequency of each recipient (or interested client) for that client object is also altered and GtK subsequently checks if it is time to flush the queue.

Contributions to the top-most conference window are flushed out immediately from the recipients' queue each time a new contribution is added and sent straight away to the respective client objects. On the other hand contributions to the background conference window remain in the recipients' queue until it is the time to flush the queue (Section 8.4.3). However, when a background window is brought to the front, any contributions waiting in the recipients' queue become 'overdue' and they are instantly sent out to the relevant client objects, thus leading to the 'catch up' behaviour.

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 and feedthrough 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 instance, when a new object is made visible or a window is popped to the front. This is precisely why GtK separates registering interests, typically once per object, from setting frequency which may happen repeatedly.

9.4.4 Illustrating pace impedance matching

Let us assume that there are three users Jane, Sue and Tom who have logged on to the example conferencing application, as shown in figure 9.19 (a). The Recipient set for each conference is managed by the Notification Manager and it includes the following fields:

- `source id`: this is of the form `ObjectId:HostId` (Section 8.22) and it a unique identifier for the transcript object. Each conference has a related transcript object on the Conference Manager. For example, the `source id` for the `CSCW` conference is `10005:0`.

- `target id:` this is also of the form `ObjectId:HostId` but it identifies each client object. The `ObjectId` on its own is not unique (for instance, both Jane and Sue have the same `ObjectId` for their `CSCW` conference objects) but when it is used in combination with the `HostId`, then it becomes a unique identifier.

- `eventtype`: this represents the type of event associated with a message or event.

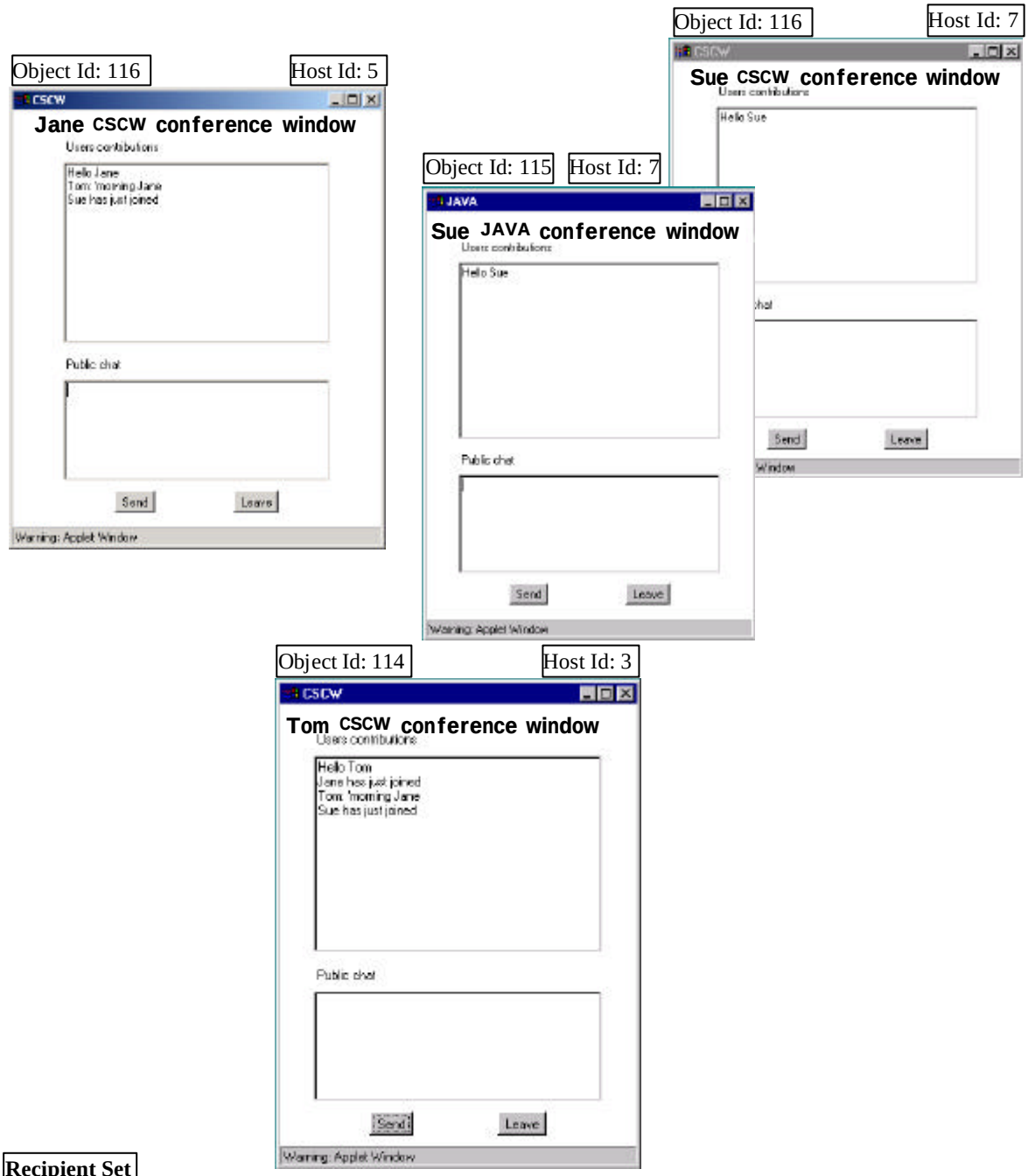
- `frequency:` this is the frequency level that a client object registers with the Notification Manager.

- `queue:` this shows the different contributions placed in the recipient's queue.

In figure 19 (a) both Jane and Tom have joined the `CSCW` conference while Sue has joined two conferences, namely the `JAVA` conference where her main focus lies (window in the foreground) and the `CSCW` conference where she only has a passive or peripheral interest (window in the background).

Jane and Tom clients' register a high pace interest in the changes to the `CSCW` conference by sending a `"change frequency"` event type to the Notification Manager with default frequency parameters.

Sue's main focus lies on the `JAVA` conference, so her client registers a high pace interest in the changes to the `JAVA` conference. Sue's client also registers a low pace interest in the changes to the `CSCW` conference by sending a `"change frequency"` event type to the Notification Manager with a pre-defined frequency of (`3,30000`).



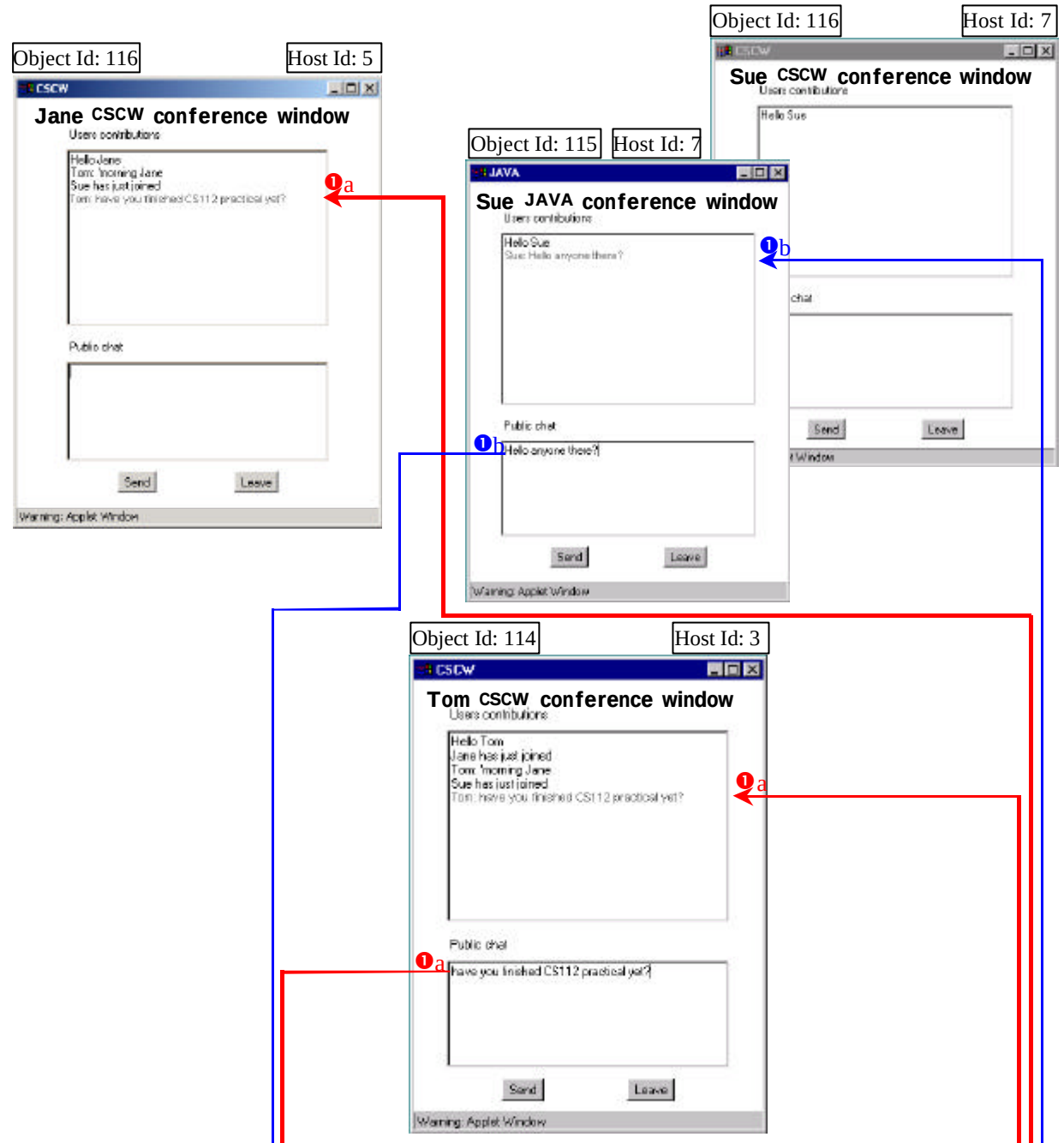
Recipient Set

Transcript object	source id	target id	eventtype	frequency	queue
CSCW conference	10005:0	116:5		(0, -1)	
		114:3		(0, -1)	
		116:7		(3, 30000)	

Transcript object	source id	target id	eventtype	frequency	queue
JAVA conference	10006:0	115:7		(0, -1)	

Figure 9.19 (a) Example scenario

The Notification Manager will thus update Sue's `CSCW` conference client when either a maximum of `3` messages are reached or the length of time the messages have been in the queue is over `30000 milliseconds`.



Transcript object	source id	target id	eventtype	frequency	queue
CSCW conference	10005:0	116:5	"newline"	(0, -1)	"have you finished CS112 practical yet?"
		114:3	"newline"	(0, -1)	"have you finished CS112 practical yet?"
		116:7	"newline"	(3, 30000)	"have you finished CS112 practical yet?"

Transcript object	source id	target id	eventtype	frequency	queue
JAVA conference	10006:0	115:7	"newline"	(0, -1)	"Hello anyone there?"

Figure 9.20 (b) Adding contributions

Figure 9.19 (b) shows that Tom has added contribution ❶a to the CSCW conference while Sue has added contribution ❶b to the JAVA conference. When the Notification Manager receives input ❶a and ❶b, they are first added to their relevant recipient's queue. GtK

then verifies whether the time to flush each recipient's queue has been reached by checking if either one of the pace parameters, `queueLength` or `time` is met.

This condition is satisfied, in the case of the `CSCW` conference by Tom and Jane, and in the case of the `JAVA` conference by Sue. Consequently, GtK flushes Tom and Jane `CSCW` conference clients queue straight away and sends out input ❶a to them. The same happens with Sue's `JAVA` conference client queue and she receives input ❶b immediately. However input ❶a remains in Sue's `CSCW` conference client queue and GtK sets an alarm to wake up at the next flush time (Section 8.4.3.1).

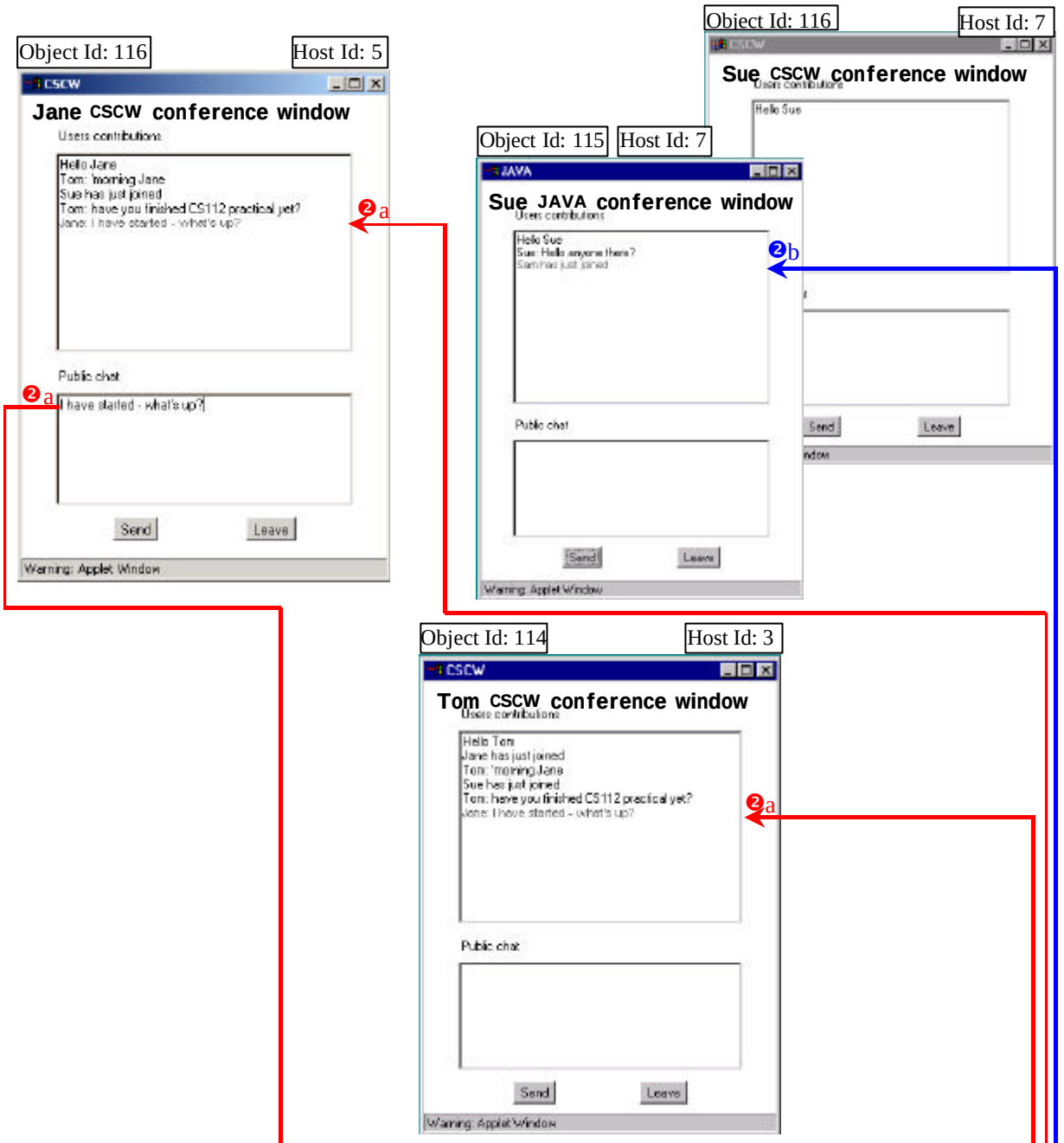
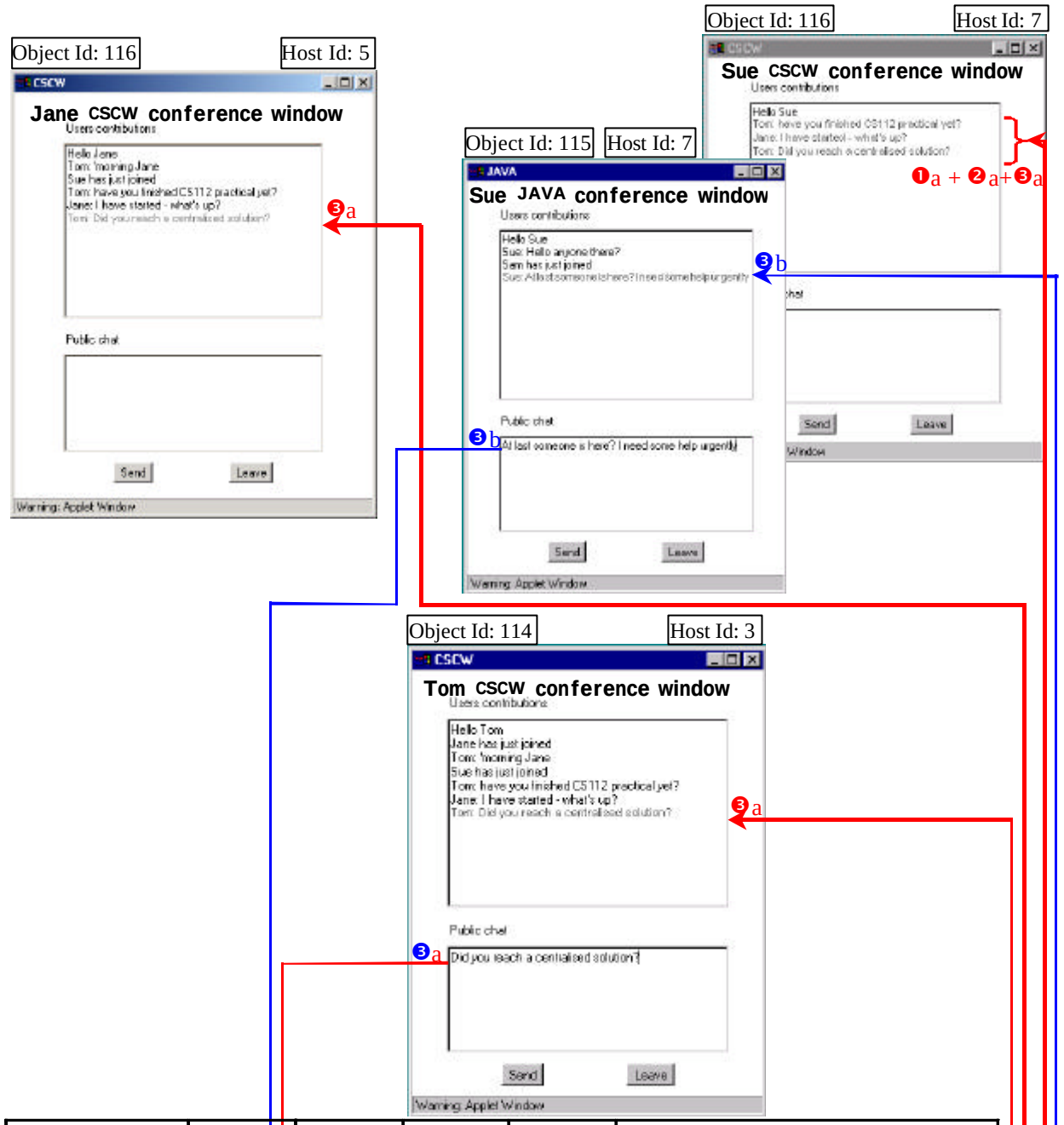


Figure 9.21 (c) Managing contributions

Figure 9.19 (c) shows Jane replying to Tom on the CSCW conference with contribution ❷a. As before, GtK broadcasts input ❷a to Tom and Jane CSCW conference clients straight away. But input ❷a is added to Sue's CSCW conference client queue as the time to flush the queue has not been reached.

Another user Sam who has now joined the JAVA conference (represented by the blue arrow). Note that, Sam's conference window has been omitted from the figure in order to keep the layout simple. The JAVA conference Transcript object on the Conference Manager sends a message with a "newclient" event type to Sam's JAVA conference client, to which the latter responds by displaying a greeting message on Sam's JAVA conference window (Section 9.3.4).

At the same time, GtK finds all the recipients for the JAVA conference to inform them of Sam's presence. Sue therefore receives event ❷b to her conference client with a "newclient" event type. Sue's JAVA conference client responds by displaying the message '*Sam had just joined*' on her JAVA conference window (Section 9.3.4).



Transcript object	source id	target id	eventtype	frequency	queue
CSCW conference	10005:0	116:5	"newline"	(0, -1)	"Did you reach a centralised solution?"
		114:3	"newline"	(0, -1)	"Did you reach a centralised solution?"
		116:7	"newline"	(3, 30000)	"have you finished CS112 practical yet?" "I have started - what's up?" "Did you reach a centralised solution?"

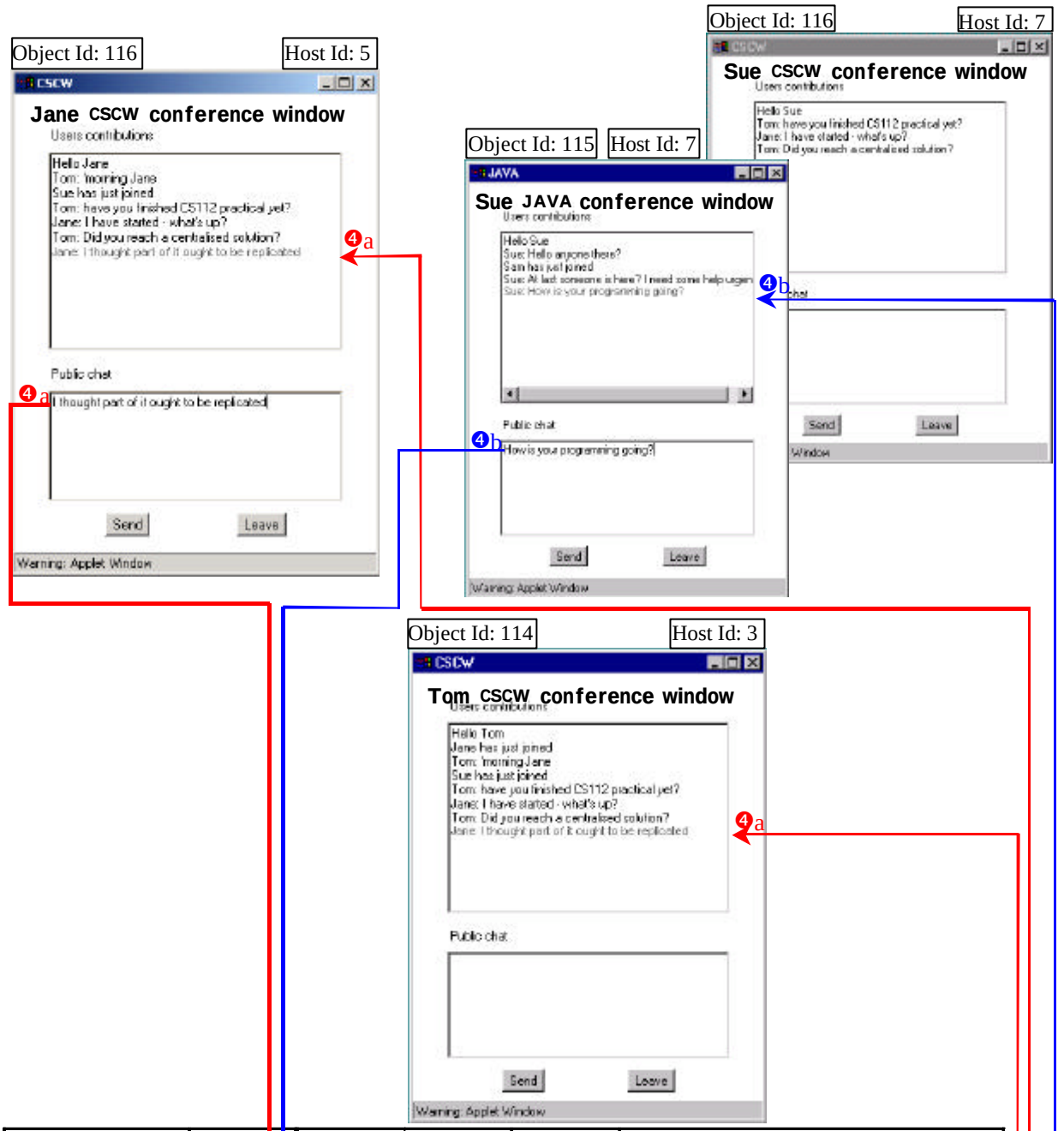
Transcript object	source id	target id	eventtype	frequency	queue
JAVA conference	10006:0	115:7	"newline"	(0, -1)	"At last someone is here? I need some help urgently"
		118:6	"newline"	(0, -1)	"At last someone is here? I need some help urgently"

Figure 9.22 (d) Queue flush time reached

Figure 9.19 (d) shows Tom's response to Jane on the CSCW conference with contribution ❸a while Sue sends contribution ❸b to the JAVA conference. Again, GtK instantly

forwards input ❸a to Tom and Jane `CSCW` conference clients and input ❸b to Sam and Sue `JAVA` conference clients.

GtK adds input ❸a to Sue `CSCW` conference client, but now the 'check for the flush time' returns true as the `queueLength` pace parameter of the frequency equals its maximum length of `3` messages. Consequently, GtK flushes the items from Sue's `CSCW` conference queue and Sue receives all three contributions ❶a,❷a and ❸a in a single event on her `CSCW` conference window. Note that, if the `time` pace parameter had exceeded, it would trigger a similar reaction.



Transcript object	source id	target id	eventtype	frequency	queue
CSCW conference	10005:0	116:5	"newline"	(0, -1)	"I thought part of it ought to be replicated"
		114:3	"newline"	(0, -1)	"I thought part of it ought to be replicated"
		116:7	"newline"	(3, 30000)	"I thought part of it ought to be replicated"

Transcript object	source id	target id	eventtype	frequency	queue
JAVA conference	10006:0	115:7	"newline"	(0, -1)	"How is your programming going?"
		118:6	"newline"	(0, -1)	"How is your programming going?"

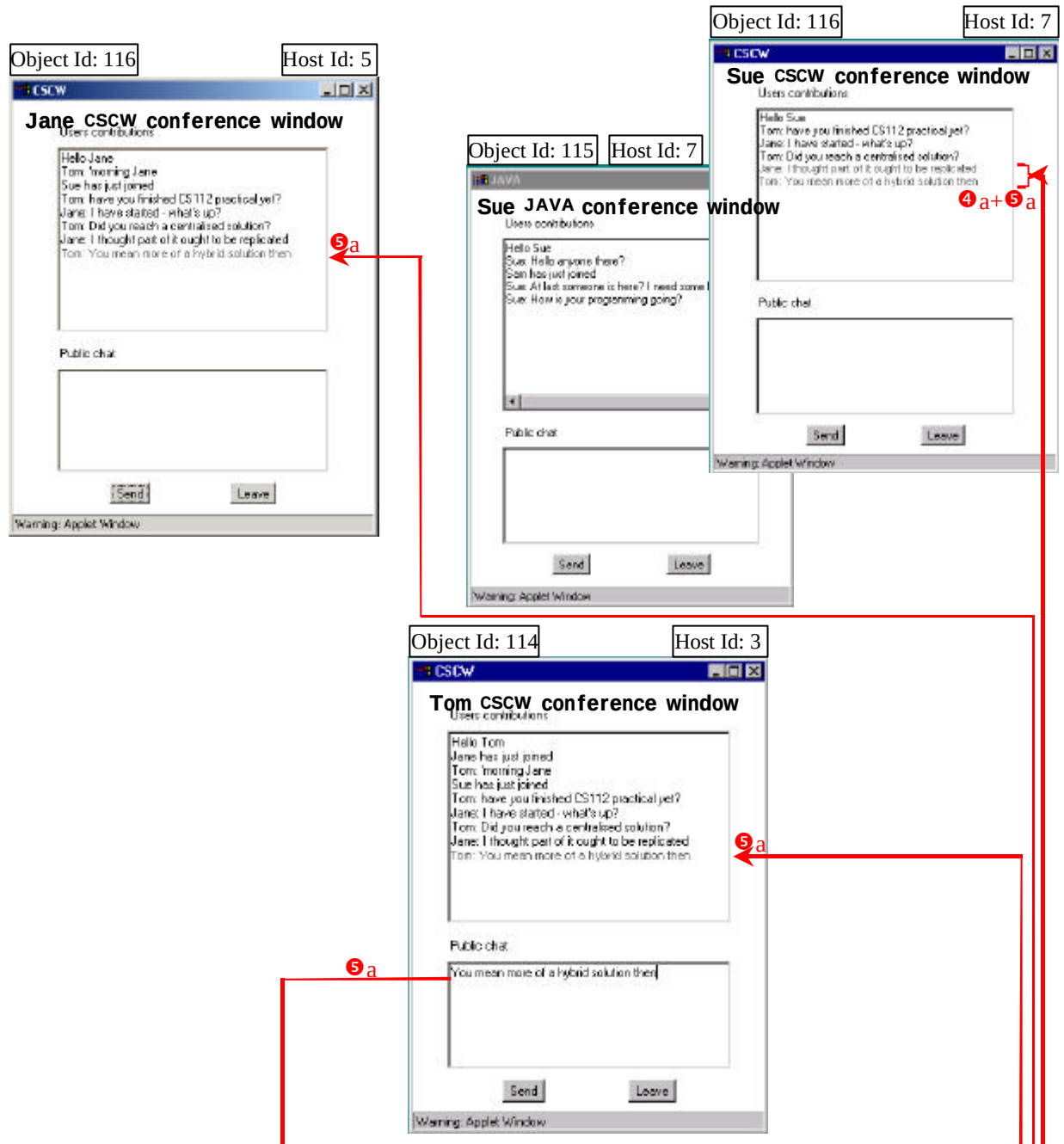
Figure 9.23 (e) Adding contributions

Figure 9.19 (e) shows Jane's contribution ❹a to the CSCW conference and Sue's contribution ❹b to the JAVA conference. GtK broadcasts input ❹a to Tom and Jane CSCW conference clients immediately but adds input ❹a to Sue's empty CSCW conference queue. GtK also broadcasts input ❹b to Sue and Sam JAVA conference clients straight away.

In the final scenario, Tom replies to Jane on the `CSCW` conference with contribution ❺a. Once again, GtK sends out input ❺a to both Tom and Jane `CSCW` conference clients but adds it to Sue's `CSCW` conference queue.

However, if Sue suddenly decides to catch up on the thread of conversation in the `CSCW` conference, she simply clicks on the conference window to bring it into her focus, as shown in figure 9.19 (f). This action triggers Sue's `CSCW` conference client object to register a high pace interest with the Notification Manager by sending a `"change frequency"` event type with default pace parameters. As a result, the frequency level for Sue's `CSCW` conference is reset and any outstanding contributions waiting in the queue now become overdue. Consequently, GtK flushes out contributions ❹a and ❺a from Sue's `CSCW` conference queue and sends them out to her `CSCW` conference window immediately, thus leading to a 'catch up' behaviour.

Sue's `JAVA` conference window has now moved to the background, hence her `JAVA` conference client object alters her interest in the `JAVA` conference with the Notification Manager by sending a `"change frequency"` event type with low pace frequency parameters `(3,3000)`. As a result, GtK will only broadcast Sam's contribution ❺a on the `JAVA` conference to his own conference client object. GtK adds contribution ❺a to Sue's `JAVA` conference queue and it will remain there until the flush time is reached or Sue's focus changes back to the `JAVA` conference.



Transcript object	source id	target id	eventtype	frequency	queue
CSCW conference	10005:0	116:5	"newline"	(0, -1)	"You mean more of a hybrid solution then"
		114:3	"newline"	(0, -1)	"You mean more of a hybrid solution then"
		116:7	"newline"	(0, -1)	"I thought part of it ought to be replicated" "You mean more of a hybrid solution then"

Transcript object	source id	target id	eventtype	frequency	queue
JAVA conference	10006:0	115:7	"newline"	(3, 30000)	"I am having trouble with it too"
Sam's input on JAVA 5b		118:6	"newline"	(0, -1)	"I am having trouble with it too" 5b to Sam's JAVA

Figure 9.24 (f) Change in conference focus

The scenario discussed in this section has demonstrated pace impedance matching within the example real-time Web conferencing application. Pace impedance matching is very useful for providing an effective level of feedthrough to group members collaborating through different artefacts. The pace of interaction (how often one interacts) is generally more important than the bandwidth (how much one communicates) when providing feedthrough in collaborative systems. This is even more obvious on the Web as faster modem and network speeds have meant that large pages and graphics can download very rapidly, but these download times are usually dominated by the time it takes to establish a connection with the remote server.

However, a fast pace of feedthrough is not only unnecessary, but also undesirable. For awareness purposes, rapid change at the periphery of our vision becomes distracting. Indeed, if updates were propagated too quickly, applications would need to suppress some updates or 'smooth' the output. Consequently, it makes sense that if an application object or artefact is the focus of the user's attention then the user requires high pace, high fidelity notification about that object. But if the object is in the background within the user's peripheral awareness, the user will require a much lower pace and lower fidelity notification.

9.5 Summary

The discussion of the real-time Web conferencing application in this chapter demonstrates the practicality of the GtK framework. The conferencing exemplar was designed in order to investigate the issues surrounding impedance matching and it has been implemented to execute specifically within the GtK framework. The focus of the design is on those aspects that facilitate the provision of a controlled pace of feedthrough and not on developing a fully functional system with enhanced features that exploit server-side technologies. In some ways, the implementation described here is representative of more complex issues than a general conferencing system.

The behaviour of the example application at run-time was first examined with a view to analyse how users interact with the different functionalities offered by the interface. The application offers functionalities that are common to many chat systems, but its novel feature lies in providing collaborative users, who are interacting with multiple conferences simultaneously, with a pace of feedthrough that matches their rate of interests. The way in which the example application uses the GtK framework to support its interface behaviour was also thoroughly explored.

This was later augmented with a description of how the functionalities of the GtK notification server were applied to provide users with a pace of feedthrough that matches their interest levels. Users are informed of the changes to the conferences depending on the pace interest frequency that their client objects have registered with the notification server. As the users focus change from one conference to another, the client objects alter the conference pace interest frequency accordingly, thus allowing GtK to readjust the user's pace of feedthrough, eventually leading to a catch up behaviour.

In the example application, users' client objects register a high pace interest with the changes to the top-most conference window but only a low pace interest with the background conference windows. Users therefore see the updates to the top-most conference window, which is also their focus object, almost immediately (limited by network latency). However, updates to the background conferences, which lie in the users' periphery, are placed in a queue and only sent out to them when either one of the pace impedance parameters is reached.

The example application has demonstrated that the GtK notification server can indeed act as an impedance matcher both in terms of the temporal dimension, by restricting the pace of feedthrough and in terms of the bandwidth, by not sending all the information across simultaneously. The matching of the rate of feedthrough with the users' pace of interaction (in this case, the users' interest levels in particular conferences) may not necessarily result in a remarkable inprovement in the user interface, but the gain in performance is significant. Information sent in chunks or batches reduce overheads considerably as the transmission of message headers and process swapping is minimal. Besides, it is usually more efficient to send data over a network in bursts.

The real-time Web conferencing system is not the only application that can apply impedance matching. It was chosen merely as an example to show the potential of pace impedance matching in providing a controlled pace of feedthrough through the GtK framework. The use of the top-level window to manage the rate of feedthrough is just an application of the example but it is not limited to it. The demonstration presented here is complemented with an analysis of the GtK framework from an architectural point of view.

Chapter 10 Architectural Evaluation

The design space for notification servers in Chapter 6 showed that the notification server should ideally be an independent component that separates notification issues from the data. Chapter 7 proposed that a separable notification server could indeed facilitate the provision of an effective rate of feedthrough through impedance matching. This approach was applied in the implementation of the GtK experimental notification server that also supports pace impedance matching in Chapter 8.

The practicality of the GtK notification server as a pace impedance matcher was further explored within a distributed collaborative environment via the construction of an example real-time Web conferencing application. Chapter 9 described how the example application had been built on the GtK framework and how pace impedance was achieved by matching the frequency of notification with the users interest levels in the different conferences.

This chapter complements the previous assessment of the GtK framework by evaluating it from an architectural viewpoint. The benefits and limitations the GtK architectural framework are analysed and a number of potential issues for further consideration are raised. The conferencing exemplar is used to support the discussion accordingly. Also, the notification server taxonomy is employed to assess ways of change discovery and propagation.

Section 10.1 assesses the flexibility of the GtK framework within the design space for notification servers. Section 10.2 explores the possibility for migrating the components within the GtK framework to a distributed platform. Section 10.3 investigates whether the GtK framework can execute in a dynamic mobile environment. Dynamic mobile interaction causes implicit pace changes and opens up the possibility for impedance matching further. Section 10.4 examines the event management scheme of the GtK framework and considers whether impedance matching has any effect on the ordering of events within the existing infrastructure. Finally, Section 10.5 briefly looks at the possibility of using the GtK notification server for handling existing forms of data.

10.1 Flexibility

An important criterion for any framework that facilitates distributed interaction lies in the ease of supporting flexible architectures. This section revisits the notification server taxonomy (Section 6.5) to analyse the role that the GtK framework plays in the design space of notification servers. The number of possible arrangements in reaching a notification cycle (Section 6.5) was represented as a 4x2 matrix (figure 10.1).

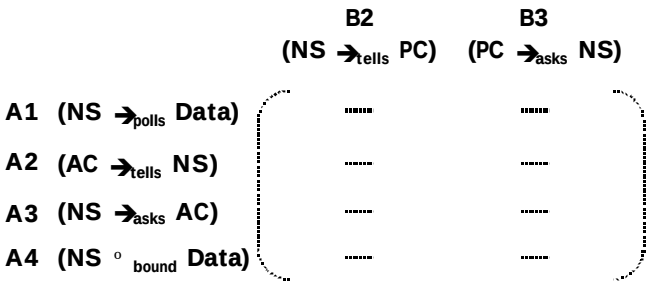


Figure 10.1 Revisiting the 4x2 matrix

A full notification cycle consists of:

A: the way in which the notification server (NS) is made aware of the changes from the active client (AC) and

B: how those changes are then broadcast to the passive client (PC).

This method of analysing change discovery and change propagation will be applied to examine the type of notification supported by the example conferencing application. The possibility for any further arrangements in the taxonomy of notification server types will also be investigated.

10.1.1 Current notification arrangement

The three main components of the example conferencing application are the clients, the Conference Manager and the Notification Manager (figure 10.2).

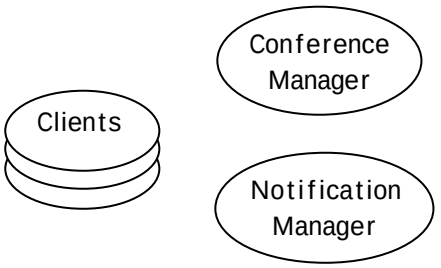


Figure 10.2 Main components of conferencing exemplar

The Notification Manager handles the functions of the GtK notification server while the Conference Manager mainly acts as the data repository. Both the Conference Manager and the Notification Manager are located on the server. Although all the components are independent of each other, the clients are aware of the Conference Manager and the Notification Manager.

The propagation of updates usually relies on issues such as the location of the data, the location of the control and who takes the initiative to send the updates. In the example conferencing application, the active client (client who initiates the change) takes the initiative to send update events to the Conference Manager (figure 10.3). The Conference Manager

then tells the Notification Manager to broadcast the changes to all the passive clients (clients who view the changes). This action triggers the GtK notification server to invoke its `tellAll()` method (Section 8.4.3.1). The Conference Manager thus assumes an active role in change propagation; however, it is separate from the Notification Manager.

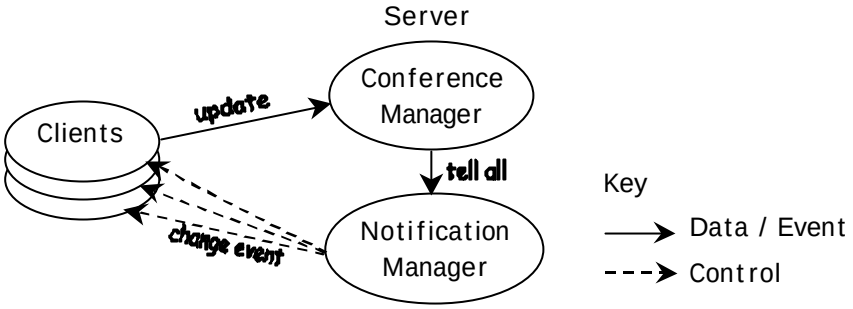


Figure 10.3 Flow of events during change propagation

The flow of events between the different components of the example conferencing application therefore satisfies the A4–B2 arrangement (figure 10.4).

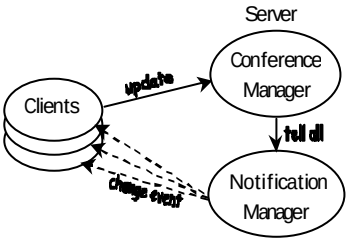
	B2 **(NS → tells PC)**	**B3** **(PC → asks NS)**
A1 (NS → polls Data)		
A2 (AC → tells NS)		
A3 (NS → asks AC)		
A4 (NS ° Data)		

Figure 10.4 GtK within the conferencing exemplar

The GtK notification server 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 to broadcast updates by using the 'tell all' function, but they need not be aware of other remote or local interested objects. GtK is thus only loosely coupled to the data repository.

As the GtK framework already achieves a separation of concern between the notification server and the data repository, one could ask whether there is a possibility that the framework may actually function in any other arrangement within the notification server taxonomy.

10.1.2 GtK as a pure notification server

The taxonomy of notification server types emphasised that a 'pure' notification server arrangement A2–B2 was desirable for promoting feedthrough on the Web, where the protocols that access data are fixed, thus forcing notification to be added at a separate level. The example conferencing application does not exploit the GtK framework in such a way that allows the GtK notification server to function as a 'pure' notification server. However, this is not a fundamental restriction on the GtK framework. Because the data service is not tied to the notification service, GtK can in fact function in different ways, including that of a 'pure' notification server.

If GtK were to satisfy a 'pure' notification server arrangement within the conferencing exemplar, the flow of events between the different architectural components would need to follow the model shown in figure 10.5.

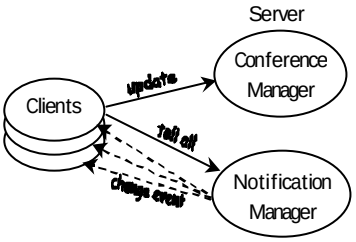
	B2 **(NS → tells PC)**	**B3** **(PC → asks NS)**
A1 (NS → polls Data)		
A2 (AC → tells NS)		
A3 (NS → asks AC)		
A4 (NS ° Data)		

Figure 10.5 GtK as a pure notification server

The active clients still send update events to the Conference Manager, as was the case in the A4–B2 arrangement (figure 10.4). However, the Conference Manager will no longer trigger updates to be broadcast. The active client itself will do so by sending ‘tell all’ events directly to the Notification Manager. GtK can then inform the passive clients who are interested in the updates that some changes have taken place. This will enable the interested passive clients to find the changes in the data straight from the Conference Manager. The Conference Manager therefore has a very passive role in the A2–B2 arrangement – it merely acts as a data repository.

The existing GtK architecture does not need to undergo any major alterations if the conferencing exemplar is to operate in the A2–B2 location. The Conference Manager should merely function as a data repository that has no knowledge of the Notification Manager. In addition, when an active client makes any changes to the data, it has to take the initiative to inform the Notification Manager directly of this occurrence.

10.1.3 Further architectural possibilities

This section will now explore any further arrangements that the GtK framework may support in the notification server taxonomy and assess their efficiency.

A1-B2 arrangement

Figure 10.6 shows the flow of events between the different components of the conferencing exemplar under the A1-B2 arrangement.

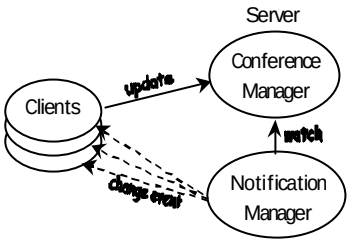
	B2 (NS → tells PC)	**B3 (PC → asks NS)**
A1 (NS → polls Data)		
A2 (AC → tells NS)		
A3 (NS → asks AC)		
A4 (NS ° Data)		

Figure 10.6 Additional location for GtK

The active clients still send update events to the Conference Manager. But the Notification Manager has to watch for changes in the data, for instance by polling the Conference Manager. GtK does not currently support polling, thus it will not function in the A1-B2 arrangement within the existing framework, unless it is modified.

Although the Conference Manager assumes a less active role, the Notification Manager has to be aware of it. In order to send the updates directly to the clients after polling the data repository, the Notification Manager needs to be aware of each client's location. For instance, clients could inform the Notification Manager of their URL when they register at launch time.

The main problem with the A1-B2 arrangement lies with deciding on the frequency with which the notification server should poll the data repository. Frequent polling increases both network and computational load. However, this opens up the prospect of using impedance matching for determining the frequency of polling. Also, the updates can remain in a queue for a certain length of time, thus offsetting any delays that the notification server may introduce when polling for the changes.

While it is possible to modify the GtK framework to allow GtK to function in the A1-B2 arrangement in the example conferencing application, there will be no gain in performance because the exemplar largely supports synchronous interaction. The A1-B2 arrangement is more efficient during asynchronous communication on the Web.

A3-B2 arrangement

In this arrangement, the data repository has to be completely separate from the notification server and the notification server has to ask for changes directly from the clients.

This implies that the notification server will have to send control messages all the clients to query for changes, even if only a few of them may have actually made those changes. Furthermore, the notification server may not necessarily discover the changes from the clients in the same order as they initially happened, thus generating the possibility of race conditions during change propagation. Consequently, the A3-B2 arrangement is not viable under normal circumstances, as it requires some complex computational mechanism to synchronise the order of events.

B3 options

The arrangements under the B3 options involve the passive clients asking for changes directly from the notification server. This is undesirable both in terms of network and computation load because each client will need to continually poll the notification server.

However, if the network suffers from frequent disconnections, then this method of change discovery is useful. Also, it is recommended on the Web that applets should not maintain a permanent socket connection with the server as the latter may suddenly run out of file descriptors. In the example conferencing application, the client applets do hold on to their socket connection, after registering with the server, for the whole duration of the exchange. If polling is to be supported within the GtK infrastructure, the notification server has to associate each client with an identifier. So, at each polling interval or after a disconnection, the notification server can perform a handshake operation to establish the nature of the identifier.

The discussion in this section has established the flexibility of the GtK framework by showing that GtK can occupy different arrangements within the notification server taxonomy. Although, GtK was used in the A4–B2 arrangement in the example real-time conferencing application, GtK is in fact a 'pure' notification server. The exemplar can execute in the A2–B2 arrangement after some minor changes to the GtK framework. GtK can also reside in other viable locations in the taxonomy but the existing framework will need to be modified.

10.2 Distribution

Another important architectural criterion lies in the ease of moving components to different physical locations. This section will examine whether the GtK framework can operate in a distributed layout and assess any related performance issues that may arise.

10.2.1 Existing physical location

Although the notification server is logically distinct from the data source in the example conferencing application, both components currently reside in the same physical address space. The Conference Manager represents the data source in figure 10.7 and the Notification Manager implements the functions of the notification server.

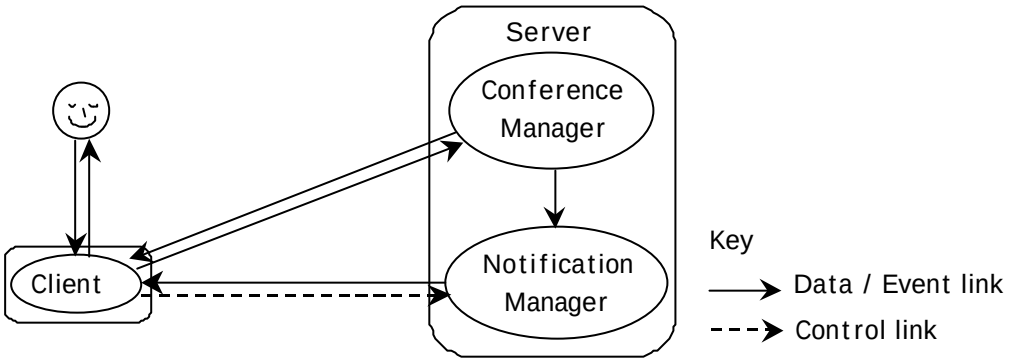


Figure 10.7 Physical structure of conferencing exemplar

The data link allows messages to be transferred between components. Note the arrow direction starts from the agent who initiated the transfer. The event link is more like an event control that also carries some data, such as the `tellAll()` method call from the Conference Manager to the Notification Manager (Section 8.3.3.1). The control link is a form of meta control (Section 5.5.2) that exists between the clients and the Notification Manager. The clients use the control link to inform the notification server of the different types of updates they are interested in (data) and the rate at which they want to be notified of those updates (event).

The separation of concern between the notification server and the data source is an inherent property of the GtK framework (Section 10.1.1). This facilitates the GtK notification server to run in a different address space as the data objects. Consequently, the functions of the notification server (Section 8.3.1) can be invoked both locally via method calls and remotely through asynchronous messages.

Let us consider the behaviour of the components of the GtK framework in a distributed environment, with possibly multiple data sources. Although the issue of distribution and the use of multiple data sources are independent, they do tend to be linked together.

10.2.2 Possibility for supporting multiple data sources

Figure 10.8 shows the physical structure of the GtK framework with several heterogeneous data sources located on different servers.

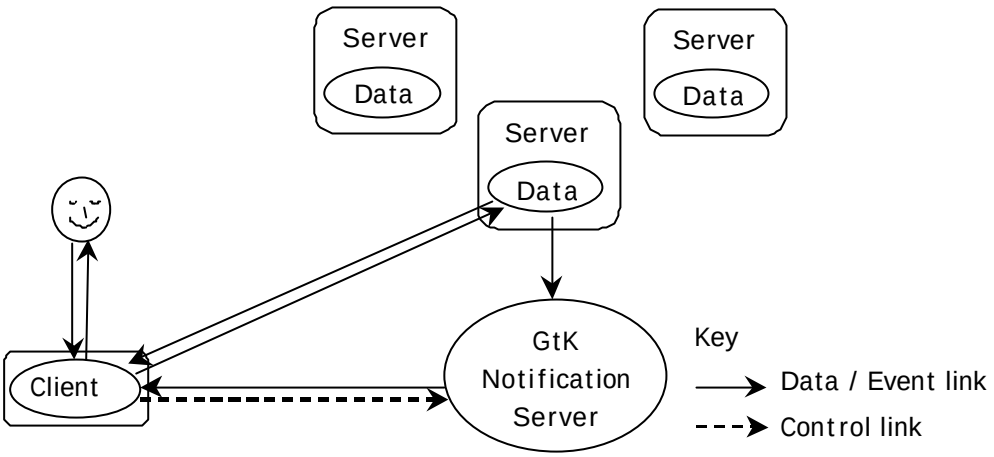


Figure 10.8 GtK framework with heterogeneous data servers

Options A1–A4 from the notification server taxonomy (Section 10.1) will now be used to assess the feasibility of change discovery and propagation between the components of the GtK framework within a distributed setting.

In the conferencing exemplar, the data repository triggers the GtK notification server to broadcast updates to the clients through the 'tell all' function. But when the architectural components are physically distributed, the load transfer between the data store and the remote notification server becomes significant. Therefore change discovery through option A4 can only be supported if the data repository knows about the notification server, in other words, the data repository must behave like an active database.

Similarly, if the notification server were to discover the updates by polling the data repository directly (option A1), this would involve a large number of network exchanges and high computational load between the data store and the remote notification server. Consequently, the rate of change propagation may be affected and the rate of feedthrough may slow down. Change discovery through option A1 will depend heavily on where the complexity lies.

However, in both options A4 and A1, the active clients do not have to be aware of the notification server when they make any changes. So, even if the clients are not very cooperative, change propagation can still take place. Also, this prevents clients from finding out changes directly from the data repository, which would otherwise amplify the volume of data transferred over the network and subsequently increase both computational and network load.

Unlike the link between the notification server and the data store, change discovery between the client and the notification server does not involve much transfer of data. The notification server can either ask the clients directly if they have made any changes in the shared data (option A3) or the active clients can independently inform the notification server after they have made the changes (option A2).

The problem with option A3 is that not all the active clients will necessarily respond at the same rate when the notification server queries them, thus affecting the timeliness in which passive clients will receive the updates. The task of synchronising the order of events discovery is more daunting in a distributed environment. However, option A3 optimises on the number of control messages exchanged and reduces the message overheads. But if the notification server ends up acting as a database, the message load will definitely increase.

Option A2 is the most efficient solution for change propagation on a distributed setting with a separable notification server. It will show a similar behaviour as the locking mechanism used in the UNIX file system where applications explicitly request locks on remotely stored files from the lock daemon. The lock daemon has no control over the files it is referred to and is thus logically distinct from the file store. The GtK framework can support this form of change discovery, but all the clients need to cooperate with each other and they will have to be aware of the notification server.

This section has highlighted the various issues that arise with change discovery and propagation within a distributed GtK framework. The discussion has also shown that the use of separable notification server like GtK is most effective within a distributed environment.

10.3 Mobility

The GtK infrastructure is primarily a static framework. It may enable some form of mobile communication, for instance a PDA could interact with a notification server that sits remotely from the data. However, the GtK framework itself does not generate components that are dynamically mobile. Dynamic mobility poses two main problems: firstly, the components usually have a roaming IP address and secondly, the network is very often subject to disconnections, hence the clients' connections are not persistent. It is therefore more problematic for the notification server to handle the timing and delivery of events to the clients; consequently, the system's efficiency may be affected.

This section will first examine how dynamic mobility can influence change discovery through the GtK framework and then consider some important pace issues that are inherent within a mobile environment.

10.3.1 Introducing mobility in the GtK framework

The GtK framework consists of the following logical components: the client interface, the data repository and the notification server (figure 10.9).

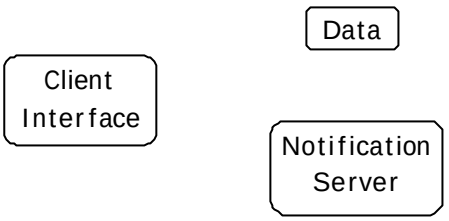


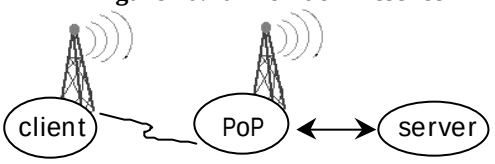
Figure 10.9 Logical components of GtK framework

But in order to support a dynamic mobile environment, a *Point of Presence* is required within the framework.

10.3.1.1 Point of Presence

A previous work[15] that investigated the architectures for mobile user interfaces proposed the need for a *Point of Presence* (PoP) in a mobile network, which acts as an important additional site for computation (Dix et al., 2000). The PoP was defined as the point where a client machine has its connection to the physical network (figure 10.10).

Figure 10.10 Point of Presence



For example, with a hand-held PDA, the PoP may be the local cell's base station. The PoP is not necessarily the first point of contact. It also satisfies some functionality criteria, such as the closest point with greater computational power or better network connectivity. Consequently, the PoP is able to engage in a faster pace of interaction than a server-based interaction.

10.3.1.2 Interaction through the PoP

The logical components the GtK framework (figure 10.9) can be mapped onto the PoP layout (figure 10.10) to produce the logical components of the GtK framework within a mobile environment (figure 10.11).

[15] Interfaces and Infrastructure for Mobile Multimedia Applications research project – as part of the EPSRC MNA programme, GR/L64140 & GR/L64157

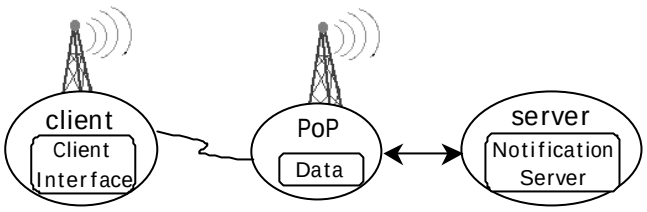


Figure 10.11 Logical components in mobile environment

The client interface still sits on the client, but the presence of the PoP opens up the possibility of having either the data store or the notification server distributed in the PoP. Each case will now be analysed in turn by applying options A1–A4 from the notification server taxonomy.

Data repository distributed in PoP

When the data repository is distributed in the PoP, the PoP acts as a cache and the notification server remains centralised. A centralised notification server will facilitate options A2 (the active client tells the notification server) and A3 (the notification server asks the active client) for change discovery. However, because the PoP only holds a cached copy of the data, the interested passive clients may take longer to find out the real changes in the data.

In order to perform change discovery though option A1 (the notification server polls the data), the notification server has to be aware of the data location prior to polling. Also, the success of this option relies on how far the data is cached and on the state of the network connection. Option A4 (the notification server is bound to the data) is more practical because the notification server is already aware of the data location. So, even if the data is mobile, the notification server can still discover the changes.

Notification server distributed in PoP

This is an interesting layout, which can be useful in a scenario where two users are working together in the same meeting room through their digital pads. For feedthrough reasons, it is more efficient if the users actions get propagated through a local notification server instead of linking with a remote central notification server. The purpose of a notification server is to know when the data has changed but not its actual content. So, if the data repository is kept centralised when the notification server is distributed in the PoP, the notification server will take longer to retrieve the updates, thus slowing down feedthrough. Hence, the data too has to be distributed when the notification server is distributed in the PoP.

Change discovery via option A1 (the notification server polls the data) is impractical with a distributed data repository. However, change discovery through option A4 (the notification server is bound to data) may still be possible as the local notification server can be closely bound to some locally cached data.

Change discovery via options A2 (the active client tells the notification server) and A3 (the notification server asks the active client) can also be supported when the notification server is distributed in the PoP. However, if one of the clients in the above meeting room example leaves the room, this will generate some significant notification issues. For instance, how will the notification server know the new location of the client and more importantly, should the client keep a persistent connection with the local notification server within the PoP or should it connect to the central notification server.

Another difficulty with using a distributed notification server and data repository lies with data synchronisation. If the data gets lost, the notification server can at least contact the central repository, but if notification events get lost (e.g. there is a sudden network disconnection or the same network connection is not maintained during an exchange), it is more problematic to retrieve those events. These issues need further consideration.

10.3.2 Pace issues in mobile interaction

The very nature of mobile interaction introduces some implicit pace changes – the hosts are mobile, the context of execution is dynamic and above all, the network connection is intermittent. If users are provided with a uniform pace of feedthrough in such an environment, the load on the network will be too high. Therefore the users context of interaction has to be taken into account and impedance matching can be applied to provide a controlled pace of feedthrough.

Consider the following scenario. John and Mary are interacting in the same room through their digital pads, for instance copying data to each other. As they are both in the same room, they require a high pace of feedthrough. But if Mary moves to another room shortly after to carry on working offline for a while, she does not require the same pace of feedthrough until she meets up with John later to resynchronise their work.

Consequently, the users' pace of feedthrough can vary depending on their location – pace impedance. Furthermore, the volume of data can be reduced as the users move to lower the network load – volume impedance. Given that the GtK framework already enables pace impedance matching, it could be applied in a mobile environment to adjust the pace of feedthrough that users receive. But unlike the conferencing exemplar, where the feedthrough rate matched the users interests (Section 9.4.2), the pace of feedthrough in a mobile environment can be adjusted depending on the users location.

There are in fact three ways in which the pace of feedthrough can be managed remotely:

(a) application driven – the application can change the pace by informing the notification server that the user has moved.

(b) user driven – the user can send a request for a change in the pace when moving to a new location.

(c) via a generic component – a generic component can be used to track the users location and send location events to the notification server.

The GtK framework requires an additional generic component within its infrastructure – the Location Manager, to provide pace impedance in a mobile environment. The Location Manager will be responsible to send information such as the type of application and the type of location to the notification server (figure 10.12). The latter can then use this information to provide a pace of feedthrough that matches both application type and location type. This will not only enhance location awareness but it will also provide users with a better service.

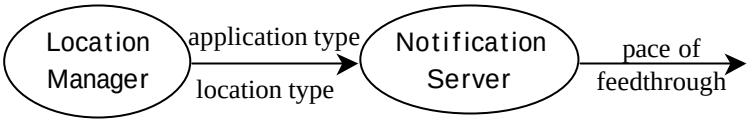


Figure 10.12 Pace impedance matching in mobile environment

This section has shown that the GtK framework requires a PoP within its infrastructure to support dynamic mobile interaction. As the PoP is an additional computational component, it will influence the ways in which the notification server discovers and propagates changes to the users. Mobile interaction causes implicit pace changes, thus impedance matching can be applied to improve the timing and delivery of events to the users. The GtK framework can be enhanced to provide pace impedance in a mobile environment.

10.4 Event Management

Event management is a key architectural concern with impedance matching as it introduces delays in change propagation. As a result, certain types of ordering problems that would not normally arise can in fact surface here. Chapter 7 gave an example that explored the effects of impedance matching on the ordering of events and the resulting impact on the users interaction (Section 7.6.3). This section will analyse the event management scheme within the GtK framework and consider the impact of impedance matching on the existing infrastructure.

10.4.1 Event ordering in the GtK framework

Event ordering in the GtK framework is based on a star configuration (figure 10.13). This simplifies the serialisation of events, as the notification server is the single locus of control. The central notification server facilitates the synchronisation of client events when they are broadcast even if the notification server initially accepted the events in a different order or the events hit the server at the same time. Note that, the events may not necessarily reach the notification server in 'real-time' order due to network latency. It is more complex to ensure that events are broadcast in 'real-time' order, as each client event will need to be time stamped with some 'global' time.

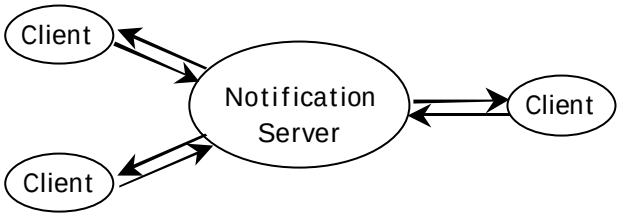


Figure 10.13 Star configuration in GtK framework

An alternative method for event propagation is through a decentralised peer-peer network. This generates multiple paths for event propagation. Also, when a client performs an action it may trigger changes in other clients, which is a direct consequence of the causality effect. The peer-peer network does not guarantee the ordering of events as illustrated by the scenario in figure 10.14.

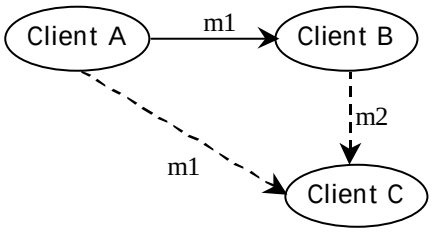


Figure 10.14 Possibility of race condition with peer-peer network

Client A sends message m1 to both client B and client C. Message m1 reaches client B almost instantly but for some reason, it gets delayed in transit to client C. When client B receives message m1, it processes it and generates message m2, which is then sent to client C. If the network connections are equally fast along both paths, client C will receive the messages in the right order; that is m1 followed by m2. But because there are different paths for change propagation, there is a possibility that client C receives message m2 before message m1. So, if client C processes message m2 and then receives message m1, race condition will arise.

The star configuration in the GtK framework guarantees the order in which the Event Manager transmits events at the lowest level. However, at the higher Notification Manager level, impedance matching introduces delays as the events are placed in a queue when they reach the notification server (Section 8.4.3). Hence there is a risk that the notification server may not necessarily broadcast the events in the same order that it received them originally.

10.4.2 Maintaining event ordering with impedance matching

The incorrect ordering of events can lead to inconsistencies in the interpretation of messages exchanged between users (see example in Section 7.6.3). The conferencing exemplar avoids such inconsistencies by being a simple application that stays in a quiescent state. Also, the fact that the data is managed centrally is a contributory factor.

The following example illustrates event ordering with pace impedance in the GtK framework (figure 10.15). A client object generates two types of events: A and B. The queue length (Section 8.4.1) for type A event is 3 and that for type B event is 0.

Action	Event	Effect of action
`tellAll()`	A1	place in queue
`tellAll()`	B	send immediately ⟶ B
`tellAll()`	A2	add to queue
`tellAll()`	A3	flush queue ⟶ A1, A2, A3

Figure 10.15 Event ordering with impedance matching

After a `tellAll()` action (Section 8.4.3.1) for broadcast, GtK sends out the type B event to the interested clients immediately after receiving it. However, GtK places the type A event in a queue and subsequent type A events are added until the queue has reached its maximum length of 3. GtK then flushes the queued events and broadcast them to the interested clients in a single stream (A1, A2, A3). So as long the events are of the same type, the notification server will broadcast them to all the interested clients in the right order, even with impedance matching.

The conferencing exemplar treats all user contributions as a `"newline"` event type (Section 8.2.3) to ensure that they reach their destination in the right order. Furthermore, the clients set the same frequency of updates with the notification server. This guarantees that the events are removed from the queue and broadcast to the users in the same order. If inconsistencies still occurred despite these measures, users can simply click on the relevant conference window to bring it into focus and they will receive any outstanding events instantly (Section 9.4.3).

10.4.2.1 Limitations

Although the star configuration helps to preserve the order of events in the GtK framework with impedance matching, it does have certain limitations. A single path for event ordering through a central notification server will promote consistency within the application and provide users with a fast rate of feedthrough. But the rate of feedback rate may not be so fast. However, if an application uses GtK as a 'pure' notification server, the users will still receive a rapid feedback.

The ordering of events becomes more problematic when different modes of data exchange are allowed, such as images. The conferencing exemplar only supports data transmission in text form. Because images usually take longer to send than text, the clients will need to set different frequencies of updates with the notification server. This will obviously generate events of different types and it will be more problematic for the notification server to broadcast them in the right order. Event ordering in such cases, have to be dealt with by using fairly complex algorithms. Another possible solution is to let the notification server decide when it wants to flush the event queue by synchronising the events from the same objects.

This section has examined the event management scheme within the GtK framework. Although the star configuration can preserve the ordering of events even with impedance matching, it does have some limitations.

10.5 Interacting with existing data

Whereas other studies have implemented notification servers that are tightly bound to the data they regulate, GtK is a 'pure' notification server (Section 10.1.2). Any application built using the GtK framework will require a bespoke data manager, such as the Conference Manager in the conferencing exemplar. Because GtK is separate from the data service, it has the advantage that it can handle many types of data – from a standard database to third party databases including multiple data sources (Section 10.2.2) and more importantly, legacy data.

NSTP (Patterson et al., 1996) is an example of a bundled solution that combines the data service and the notification service together. The notification service cannot be easily integrated with existing applications, thus reducing its scalability considerably. The lack of a clear separation between the functionalities of the notification server and the data restricts NSTP capability for reuse in a different environment with different types of data. Also, because the data is shared with the notification server, the quality of the code will be affected during reuse.

In contrast, GtK is solely a notification service that can be easily plugged with other components in a different environment, thus making it easily portable and adaptable for reuse with legacy data.

10.6 Summary

This chapter has provided a systematic critique of the GtK framework by focussing on architectural issues such as flexibility, distribution, mobility, event management and data interaction. The options for change discovery and propagation from the notification server taxonomy were applied throughout this analysis to show how they influence the different architectural criteria.

The flexibility of the GtK framework was assessed by examining the effects of placing the components of the example real-time Web conferencing application in the different arrangements of the notification server taxonomy. The conferencing exemplar does not exploit the GtK framework in such a way that allows GtK to function as a 'pure' notification server. But as the notification service is only loosely coupled with the data service, GtK can in fact function in different arrangements, including that of a 'pure' notification server. Basically, the client objects will need to inform GtK after they have made some changes in the conferencing exemplar. GtK can then pass on this information to all the interested clients or recipients and the latter can pull the updates straight from the data repository. GtK can also satisfy other arrangements within the design space, such as poll for changes, but the existing framework will need to be modified.

The separation of concern between the notification server and the data source can enable GtK to function in a distributed environment with multiple data sources. The notification server can discover changes in the data either directly from the data repository or from the clients themselves. The former option involves data transfer over the network, thus issues such as network and computational load become significant. The latter option is more efficient especially when the clients themselves inform the notification server when they make the changes, which implies a 'pure' notification server arrangement. However, all the distributed clients should be aware of the notification server and their peers to improve user feedthrough.

The GtK infrastructure is primarily a static framework. In order to support dynamic mobile interaction, it is desirable to have a *Point of Presence* (PoP) within its infrastructure. The PoP is an additional site for computation, which can hold either the data or the notification. This will influence the way in which the notification server discovers and propagates changes to the users. The nature of mobile interaction causes implicit pace changes. Impedance matching can be applied to provide users with timely feedthrough depending on their context of interaction. The GtK framework can be used to provide pace impedance in a mobile environment by matching the rate of feedthrough with the users' location. However, an additional generic component for location awareness is required within the infrastructure.

The event management scheme within the GtK framework was also analysed. The GtK framework uses a star configuration and all client events are managed through the central notification server. The star configuration helps the marshalling of events at the lower levels of the GtK framework. However, impedance matching does not consolidate the ordering

of events at higher levels, when propagating changes to the clients, because it introduces delays. The alternative peer-peer network for event management increases the likelihood of race conditions. The star configuration at least ensures that events of the same type will be broadcast in the right order, even with impedance matching.

The scope of the star configuration is however limited to applications with simple event management schemes. It will be more problematic for the central notification server to preserve the ordering of events in applications that support different modes of data exchanges, as they will generate different types of events. Event ordering in such cases, have to be dealt with by using more substantial algorithms.

The separation of the notification service from the data service is a desirable feature of the GtK framework for handling different types of data. The GtK notification server can interact with a standard database or third party databases including multiple data sources. But more importantly, GtK can be easily plugged with other components in a different environment, thus making it easily portable and adaptable for reuse with legacy data.

Chapter 11 Conclusion

This thesis has presented an architectural framework that allows the construction of temporally coherent collaborative applications. The analytic focus of this work was on gaining a deep understanding of how collaborative applications can be built within a distributed environment in order to provide a desirable temporal behaviour. The emphasis throughout this work has been on the architectural aspects of application building through a number of analytical studies.

The temporal issues of interaction were analysed in Chapter 2. Chapter 3 examined the interface and architectural issues involved in single-user interaction. A similar investigation was carried out for multi-user interaction in Chapter 4. Chapter 5 provided an analytical framework for analysing collaborative architectures on the Web. The design space of notification servers were analysed in Chapter 6 and a taxonomy of notification server types was presented. Chapter 7 proposed impedance matching as a method for providing collaborative users with a controlled pace of feedthrough.

The results of the above studies were applied to develop the Getting-to-Know (GtK) separable notification server that provides pace impedance matching. Chapter 8 described the implementation details of the GtK notification server. Chapter 9 demonstrated the practicality of the GtK framework through a real-time Web conferencing exemplar. Finally, Chapter 10 provided an architectural evaluation of the GtK framework.

This chapter reflects on the work described above. Figure 11.1 shows an overall structure of the discussion that now follows.

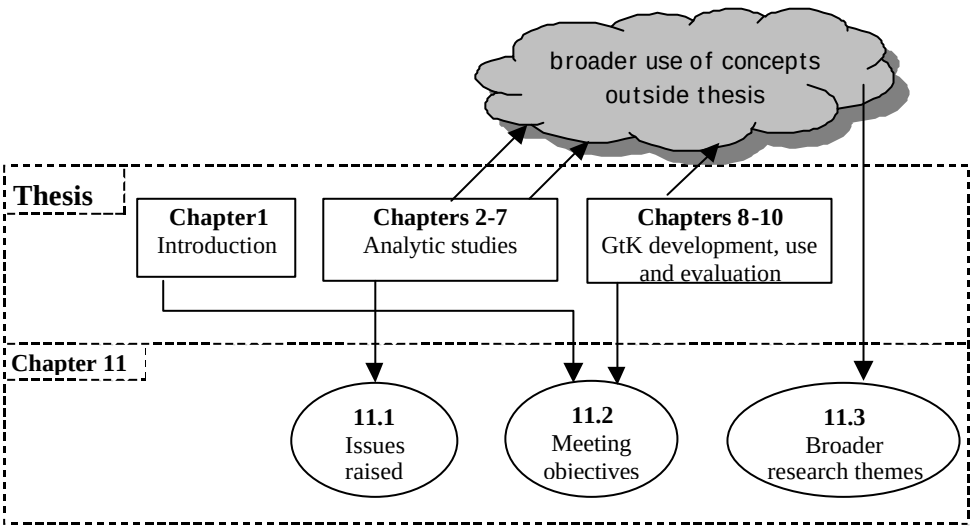


Figure 11.1 Chapter structure

Section 11.1 considers the salient issues raised by the analytical studies described in Chapters 2 – 7. Section 11.2 then reviews the objectives set out in Chapter 1 to show how these have been met through the development of the GtK framework, its use and evaluation

in Chapters 8 – 10. The work described in the thesis is already being developed in several areas of research. So instead of discussing the possibility of future work, Section 11.3 describes the ongoing work as broader research themes.

11.1 Issues raised by analytical studies

Chapter 2 discussed the importance of temporal issues and their effects on users' interaction. Delays and interruptions generate inappropriate timing and increase user frustration and application errors. Temporal properties have traditionally been linked to the system response time. Feedback is the dominant temporal property during single-user interaction. But in collaborative interaction, feedthrough is another vital temporal property. Cooperative users require both timely feedback of their own actions and feedthrough of others actions to enable successful collaboration. Temporal problems become more significant in distributed collaborative applications as delays are likely to arise from both network-related issues and the nature of collaborative work. The theoretical foundations of interface behaviour and pace of interaction were applied to analyse the temporal problems that users perceive at the interface.

Chapter 3 investigated the interface and architectural concerns for single-user applications. The discussion showed that the underlying architecture of a system affects its external behaviour. A number of desirable requirements for single-user interfaces were identified; among which separation, direct manipulation and rapid semantic feedback have a major influence on the temporal properties of the interface.

Some common architectural models and interface development tools were also reviewed. Most of these architectures promote the separation of the application semantics from the user interface functionality. However, such a degree of separation is sometimes difficult to achieve in practice and often ignored. Separation conflicts with the needs of rapid semantic feedback in direct manipulation interfaces. Consequently, aspects of the user interface may 'leak' into the application and vice versa in single-user applications.

Chapter 4 extended the architectural analysis to the context of multi-user collaborative applications. Separation was found to be an essential architectural requirement in such an environment for providing effective user-level behaviour. In addition to rapid feedback, the requirements of timely feedthrough, awareness and sharing are critical in meeting the needs of collaborative users. Furthermore, the shared data should be kept consistent and an effective control mechanism is required to handle change propagation.

Some multi-user architectural models and interface development tools were also reviewed. Most systems use a networked solution with access to information through either centralised or replicated window architecture. A hybrid architecture is employed in some cases, where certain parts of the system are centralised and others are replicated. However, there is always a tension between the responsive nature of replicated architectures that allow rapid local feedback and the need for a centralised component to promote feedthrough.

The provision of feedthrough is a major concern in collaborative applications. Most collaborative solutions tend to assume 'control' over the entire system, with bespoke software running at the users' own workstations and at various servers. In addition, they also implicitly assume that the machines are connected to a single local area network and the properties of this network are stable. These assumptions are challenged by the dynamic nature of the Web.

Chapter 5 presented an analytic framework for constructing collaborative applications on the Web. The examination of behavioural requirements identified the key architectural components of collaborative systems. The placement decisions revealed the conflicting needs of feedback and consistency on the Web. This is commonly dealt with by using either caching or replication to bring the shared data 'closer' to the user. The Web also forces the concern between *where* the data is stored and *where* the control lies, thus generating various alternatives for the location of architectural components.

For example, Web applications use Java applets to download code to users' own machines. This implies that both code and data can be stored in a permanent location while having an ephemeral location for execution or use. The mobility issues associated with data and code generated a storage/use matrix for data and a storage/execution matrix for code. The combinations for code and data placement showed that dynamically downloaded code of which applets are the most common together with caching is a truly Web-based option. Although this favours rapid feedback, as the real data is located centrally, it does conflict with the needs of feedthrough.

The analysis in Chapter 5 narrowed the focus of this work on facilitating the important behavioural requirement of feedthrough. Feedthrough is an intrinsic limitation of Web-based collaborative applications. Also, the provision of feedthrough is more demanding, as both the pace of group users interaction and network-related delays have to be taken into account. This work proposed an architectural solution to address this problem by using a suitable notification mechanism, which not only manages the rate of feedthrough, but also optimises on the temporal performance. The standard Web protocol offers some weak forms of notification, which are essentially polling mechanisms that are largely semantics free.

Chapter 6 explored the different ways in which notification services can be managed in a collaborative system by applying the foundation of Status–Event analysis. The analytic study generated a framework and vocabulary to compare and discuss different notification mechanisms. The taxonomy of notification server types showed that a notification server

should ideally allow a separation of concern between notification and data, thus behaving as a 'pure' notification server. This is particularly important on the Web as the protocols that access data are mostly fixed; hence notification has to be added at a separate level.

Chapter 7 presented an analytic framework for providing collaborative users with timely feedthrough and awareness through impedance matching – the matching of the required and supplied of update events. The notification server, through its central mediating position, is ideally placed to support impedance matching, by adjusting the frequency of notification to meet the users pace of interaction. Users should however inform the notification server of their required pace interests in the shared objects. The communication between the user clients and the notification server does not require the latter to have any knowledge of the application semantics; hence the notification server can still remain separate from the data.

Impedance matching enhances both goal-directed feedthrough – by allowing users to see the changes in the objects they are highly interested in almost instantly, and awareness – by informing them about the changes to the peripheral objects, albeit at a lower pace. This exploits the limited availability of computer resources and network bandwidth. Chapter 7 also explored the different ways of achieving impedance matching, namely through pace impedance and volume impedance. Pace impedance policies were analysed in details. Different triggers for regulating pace were identified and their effects on the flow of events were shown through the use of time-space diagrams.

Because impedance matching controls the pace of feedthrough to the clients by delaying the updates events, this may affect the order of event propagation. The incorrect ordering of events will not have a major impact on a system where there is no causality or dependency between the users' exchanges. But in systems where the ordering of events is crucial, the semantics and interconnections between those events have to be dealt with by using some complex algorithms.

The successive analytical studies led to the construction of a separable notification server called Getting-to-Know (GtK) that provides pace impedance matching. Table 11.1 summarises the important issues raised in Chapters 2 – 7.

Chapter	*Main issues*
2: Time and Interactivity	• temporal problems traditionally linked to response time • single-user interaction => feedback dominant temporal property • BUT collaborative interaction also need feedthrough * • delays arise due to network + nature of group work * • applies analysis of interface behaviour + pace of interaction *
3. Single-user Interface and Architecture Issues	• application architecture affects its behaviour * • need for separation, direct manipulation, rapid semantic feedback • separation conflicts with needs for rapid semantic feedback • separation often ignored in direct manipulation interfaces *
4: Multi-user Interface and Architecture Issues for Collaboration	• separation is vital in multi-user collaborative applications * • also require timely feedthrough + awareness in addition to rapid feedback • need consistent data + effective control mechanism • replicated vs. centralised architecture ≡ feedback vs. feedthrough * • overall 'control' + stable local area network connection commonly assumed • BUT assumption challenged by dynamic nature of the Web *
5: Why, What, Where, When: An analysis of Collaborative Architectures on the Web	• behavioural analysis => identified key architectural components * • placement decisions => feedback conflicts with consistency on the Web * • code and data storage location ≠ execution location on the Web • presents framework for analysing location options for data and code * • true Web-based option: dynamically downloaded code with caching • favours rapid feedback BUT conflicts with needs of feedthrough • provision of feedthrough is problematic and lacking on the Web * • proposes an architectural solution through suitable notification mechanism *
6: Exploring the Design space for Notification Servers	• uses Status–Event analysis to analyse design space for notification services • generated framework + taxonomy to discuss notification mechanisms * • 'pure' notification server is ideal – separates notification issues from data * • separate notification vital on the Web due to fixed protocols for data access
7: Impedance Matching: Coping with Limited Resources	• presents impedance matching framework for timely feedthrough * • use notification server to match updates with users pace of interaction * • BUT users have to inform notification server of their pace interests • impedance matching supports both goal-directed feedthrough + awareness * • exploits limited availability of computer resources + network bandwidth • examines volume impedance + pace impedance issues * • identifies pace triggers and showed effects on time-space diagrams * • event ordering is a main concern that may require complex algorithms

Table 11.1 Summary of issues raised in analytical studies

11.2 Meeting the objectives of the work

The overall objective of the work was broken down into three sub-goals in Chapter 1 (Section 1.2). The way in which each sub-goal has been met is examined below.

Objective 1

> *"To develop an architectural framework that enables the construction of collaborative applications that satisfy appropriate temporal properties"*

The GtK architectural framework that was constructed supports the GtK separable notification server, which provides pace impedance matching to manage the temporal behaviour at the user interface level. Chapter 8 described the implementation of the GtK framework. GtK does not provide volume impedance matching and it does not handle the ordering of events. A solution to the latter problem lies either at the underlying system level or at the programmer level in understanding the semantics of the infrastructure. GtK only supports pace impedance matching based on the 'volume of messages' and 'fixed time interval' triggers.

GtK is a separable notification server that has been built on a distributed object layered infrastructure. The Event Manager controls the exchange of messages and events within the infrastructure by using an asynchronous messaging protocol. This gives GtK a uniform, generic location-independent event model. The Notification Manager handles the main functions of the GtK notification server. GtK allows a client object to add an interest in another object or remove some or all of its interests for a certain object. GtK maintains a list of interested clients for specific objects and their recipients, and broadcasts notification events to all the interested clients' recipients. These functions together with a few housekeeping operations, allow the expression of a wide range of different application specific notification strategies.

Pace impedance was implemented by introducing two pace parameters to control the frequency of updates namely, *queue length* and *maximum delay time.* GtK maintains a queue of outstanding events for each recipient. Events are flushed from the queue and broadcast to the recipients when either pace parameter is reached. An alarm process sets and resets the *maximum delay time* parameter. Client objects can request GtK to change their frequency of notification through a 'change frequency' event. Subsequently, GtK modifies the rate of feedthrough for each recipient of that client object accordingly.

Objective 2

> *"To demonstrate the feasibility of the conceptual framework by using it as a basis for developing an exemplar that provides collaborative users with a temporal behaviour that meets their pace of interaction"*

Chapter 9 demonstrated the practicality of the GtK notification server as a pace impedance matcher through an example real-time Web conferencing application built on the GtK framework. The application offers similar functionalities as many Web chat systems, but its

novelty lies in allowing collaborative users to interact with multiple conferences, while adjusting their pace of feedthrough to match their pace of interaction.

The conferencing application enables users to create conferences on various topics and launch discussion sessions with different participants at the same time. The discussion sessions are mainly held in real-time but late joiners can also catch up with any ongoing session. The conferencing application executes on the server and users connect to the application through an applet interface from any common Web browser. The Conference Manager on the server is responsible for managing the conferences and acts as the data repository. The GtK notification server is only loosely coupled to the Conference Manager.

Each conference session is represented as a separate window on the users' screen. As the top-most window usually indicates the user's focus, users are more likely to have a high interest in the changes to the conference which features on that particular window. Likewise, users may only have a passive interest in the changes to the background conference windows, as they lie within the user's peripheral awareness. The users' clients apply this reasoning to register the relevant pace interest for each conference with GtK.

GtK provides users with the updates to their top-most conference window almost instantaneously (limited only by network latency) but it sends out the changes to their background conference windows less frequently, depending on the interest rate associated with them. However, users can shift their focus to catch up on the thread of conversation in a background conference at any time by simply clicking on the window. GtK subsequently readjusts the pace of feedthrough for that particular conference, thus allowing users to see any outstanding contributions immediately.

The example conferencing application thus provides collaborative users with a pace of feedthrough that match their interest rates on the different conferences. The focus of the design was on those aspects that facilitate the provision of a controlled pace of feedthrough and not on developing a fully functional system. Also, the use of the top-level window to manage the rate of feedthrough is just a function of the exemplar but it is not limited to it.

Objective 3

"To evaluate the effectiveness of the approach embodied by the model"

The GtK framework was assessed through the example real-time Web conferencing application in Chapter 9 by examining the effects of pace impedance matching on the interaction between the different components within the infrastructure. This was then complemented with an architectural evaluation of the GtK framework in Chapter 10. Such a method of evaluation was chosen because it is often problematic to evaluate a framework that is embodied in code. Furthermore, it would have been impractical to cover a more in-depth evaluation within the scope of this work, given its extensive analytic focus.

The example application supports pace impedance matching by allowing client objects to register pace interest frequencies with the GtK notification server for each conference that

the user joins. The pace frequency is related to the pace parameters *queue length* and *maximum delay time*. Client objects therefore register a high pace frequency with default pace parameters for the changes on the conference that features on the user's top-most window but they only register a low pace frequency (using some pre-defined pace parameters) for all the other contributions in the background conference windows.

The GtK notification server maintains each conference contributions in a separate queue. GtK immediately flushes out all contributions that are associated with a high pace frequency conference from the queue and sends them out to the respective client objects. The low pace frequency conference contributions remain in the queue until one of the pace parameter is reached. When the user's focus changes to a background conference window, the client object resets the pace interest frequency and informs GtK about it. GtK subsequently readjusts the user's pace of feedthrough, thus leading to a catch up behaviour.

The exemplar has therefore demonstrated the behaviour of the GtK notification server as an impedance matcher both in terms of the temporal dimension, by restricting the pace of feedthrough and in terms of the bandwidth, by not sending all the information across simultaneously. Although a controlled pace of feedthrough may not necessarily provide a remarkable improvement in the user interface, the gain is performance is significant. Information sent in chunks or batches over a network reduce overheads considerably.

The critique in Chapter 10 evaluated the GtK framework in architectural terms by assessing issues such as flexibility, distribution, mobility, event management, and data interaction. The GtK framework supports a separable model where the notification service is only loosely coupled with the data service and this offers a number of advantages.

Although the conferencing exemplar does not exploit the GtK framework in such a way that allows GtK to function as a 'pure' notification server, the separable nature of the GtK framework increases its flexibility. GtK is in fact a 'pure' notification server within the framework. In addition, GtK can satisfy other feasible arrangements in the notification server taxonomy. However, the GtK framework will have to be modified to satisfy those purposes.

The separation of concern between the notification server and the data source can also allow GtK to function in a distributed environment with multiple data sources. The 'pure' notification server arrangement is the most efficient solution in such a setting, but all the distributed clients will have to be aware of the notification server and their peers to improve user feedthrough.

The nature of mobile interaction causes implicit pace changes. Impedance matching can thus be applied effectively in such an environment for providing users with a timely feedthrough depending on their context of interaction. While the GtK infrastructure is primarily a static framework, its pace impedance matching functionality could for instance, enable the matching of the users' rate of feedthrough with their location. However, this requires an additional generic component for location awareness within the GtK framework.

The GtK framework adopts a star configuration for event management. Although this helps the marshalling of events at the lower levels of the infrastructure, impedance matching does not consolidate the ordering of events at the higher notification level because it introduces delays. However, the star configuration ensures that events of the same type are broadcast in the right order, even with impedance matching.

The scope of the star configuration is limited to applications with simple event management schemes. The ordering of events could in fact be enhanced in two ways – either the application could provide a better control, for example through explicit flush requests, or the framework could itself do so. However, the latter option requires substantial algorithmic and conceptual advances, which involve capturing semantic knowledge directly from the applications.

The separable nature of the GtK framework promotes interaction with different types of data. The GtK notification server can thus interact with a standard database or third party databases including multiple data sources. But more importantly, GtK can be easily adapted for reuse with legacy data in a different environment without affecting the quality of its code.

Table 11.1 summarises how the objectives of this work have been met in Chapters 8 – 10.

Objectives	*Main issues*
1: *develop an architectural framework that enables the construction of collaborative applications that satisfy appropriate temporal properties*	• addressed in Chapter 8 through the GtK architectural framework • supports GtK separable notification server that provides pace impedance • GtK is built on a distributed object layered infrastructure • Event Manager: uses asynchronous protocol for messages and events • Notification Manager: handles main functions of GtK • two pace parameters: maximum event queue length + delay time • GtK maintains a queue of outstanding events for each recipient • GtK flushes events for broadcast when either pace parameter is reached • user clients use 'change frequency' event to change rate of notification • GtK does not address volume impedance and event ordering
2: *demonstrate the feasibility of the conceptual framework by using it as a basis for developing an exemplar that provides collaborative users with a temporal behaviour that meets their pace of interaction*	• addressed in Chapter 9 through Web conferencing exemplar • Conference Manager – on server – holds central data for conferences • user client – applet on Web browser – provide user interface and rapid feedback • GtK is only loosely coupled to the Conference Manager • clients register pace interest for conferences – top conference window (user focus) => high pace – other conferences (users' periphery) => low pace • GtK matches pace of feedthrough with users interest rates – focus conference => almost instantaneous updates – other conferences => less frequent updates

	• GtK readjusts pace of feedthrough as users' focus change – sends outstanding contributions immediately • use of top window to manage feedthrough is only a function of exemplar – but not limited to it
3: *evaluate the effectiveness of the approach embodied by the model*	• addressed in Chapter 9 + Chapter 10 • objective 2 shows GtK can be used for temporally rich exemplar • client objects register pace frequency with GtK – high pace (focus conference) => default pace parameters – low pace (other conferences) =>pre-defined pace parameters • GtK maintains conference contributions in a queue – high pace ⇒ flushes queue immediately – low pace ⇒ remain in queue until either pace parameter is reached • exemplar demonstrates GtK as impedance matcher – restricted pace of feedthrough => temporal reduction – chunking of transmitted information => network reduction • notification and data service only loosely coupled in GtK framework – offers many architectural possibilities • GtK is a 'pure' notification server within framework – GtK can satisfy other arrangements in notification server taxonomy – but framework will need to be modified • GtK can also function in distributed environment with multiple data sources • can use GtK framework in a dynamic mobile environment – match users' rate of feedthrough with their location • GtK framework uses star configuration for event management – correct order for same event types even with impedance matching – but better support requires substantial algorithms • GtK can interact with other types of data without affecting its code – in particular, legacy data

Table 11.2 Summary of how objectives have been met

11.3 Broader research themes

The work described in the thesis has taken place over a number of years. Consequently, it has already led and contributed to a number of projects that are not described in the thesis. This section examines the broader themes that this work has been leading out into research for a long time. This also acts as a test to the value of the concepts and the work within the thesis.

The first contribution originates from the study of temporal problems during long-term interaction. The second stems from the analytic technique employed in investigating collaborative architectures on the Web. The third emerges from the main focus of this work, which was primarily concerned with the underlying computational infrastructure to support feedthrough in collaborative systems.

11.3.1 Trigger analysis

The insights gained from the study into long-term interaction (described in the Appendix) have important ramifications that have reached far afield from the scope of the thesis. Both the case study and its later application[16] reinforced the collaborative aspects of organisational modelling and emphasised the importance of reminders as an enabling mechanism for resuming activities following delays and interruptions.

The 4Rs (Request, Receipt, Response, Release) framework is a generic pattern, which repeats itself with similar triggers and similar failure modes. The 4Rs framework can be applied to any process-oriented task analysis. The existence of generic patterns can uncover potential problems before they actually occur and solutions found in one situation can be adapted and applied to another. Also, any deviation in the generic pattern will indicate possible breakdown points.

The trigger analysis technique has therefore been proposed as a method for task decomposition in HCI (Dix et al., 2003) and it can be used in conjunction with many task analysis and workflow methods. The strength of the 4Rs analysis lies in uncovering triggers that cause each process to occur. Triggers can determine whether a process is robust to interruptions or forgetfulness and if not, identify the cause of the failure and the instance where any problem is likely to arise. The theoretical and practical design implications of the 4Rs analysis can benefit anyone who is investigating the ecology of the workplace.

[16] MaPPiT project for mapping the Placement Process with Information Technology, a HEFCE project. Details available at: http://www.hud.ac.uk/scom/mappit/home2.htm

11.3.2 Analysing architectural options for mobile interfaces

There has recently been a massive growth in mobile communications and mobile computing. Although the end points here may be well understood (but in the case of small mobile devices difficult to design for), the network itself is much more dynamic than even the Internet, with limited bandwidth, temporary disconnection, and an ever changing network topology. The widespread use of mobile devices has increased the importance of designing appropriate user interfaces for mobile environment. This is reflected by the growing research interests in communities such as CSCW, mobile HCI and Ubicomp.

The need to consider the dynamic nature of this infrastructure and its effects on interaction places new demands on the software architecture and the overall role of the architecture. Essentially, software architecture is about 'what goes where'. In stationary networks, the 'where's tend to be fairly obvious and are normally characterised as either clients or servers. Even this can lead to a rich set of architectural alternatives, as described in Chapter 5. However, in mobile systems, the changing network topology suggests a much richer set of possibilities.

The systematic technique used during the analysis of collaborative architectures on the Web in Chapter 5 was applied in a later research project[17] to investigate software architecture options for mobile user-interfaces. This generated a PoP mobile framework (Dix et al., 2000) which clarified the design options of mobile systems with the aim of improving collaborative interaction.

There is an important trade-off between efficiency and locality issues in a mobile environment as people are more likely to move between contexts. These complex issues can be thoroughly analysed by combining the PoP mobile framework (Dix et al., 2000) with that of the notification server taxonomy (Ramduny et al., 1998). Such an analysis remains a very interesting possibility for extending this work.

11.3.3 Requirements for notification mechanisms

The main concern of this work was on the underlying computational infrastructure that enables collaborative systems to support feedthrough, and in particular, in the requirements and design of notification servers. This work has highlighted three main requirements for notification mechanisms to support feedthrough and awareness:

(a) the notification server should be a separate component

(b) it should be possible to control the pace of notification

(c) it should be possible to control the quality/fidelity of notified information

[17] Interfaces and Infrastructure for Mobile Multimedia Applications research project – as part of the EPSRC MNA programme, GR/L64140 & GR/L64157

The issue of separability (a) led to the investigation of the design space of notification servers, as described in Chapter 6. The systematic method applied to categorise the design options of notification servers contributes to the academic credibility of the discipline, as it clarifies the similarities and differences between different example systems and identifies new directions. The framework for notification servers commenced a design vocabulary in CSCW (Ramduny et al., 1998) for the implementation of notification services with the aim of improving design.

The taxonomy of notification server types highlighted the critical features of a 'pure' notification server, separate from the data it regulates and the clients it supports. This work has confirmed, through the development of GtK, that notification servers should be regarded as separate entities – certainly at a conceptual level and often physically distinct.

The possibility for controlling the pace of notification (b) was explored through pace impedance matching. GtK implemented pace impedance based on both 'time delay' and 'number of outstanding updates'. Pace impedance matching facilitates the development of client applications that require rapid detailed feedthrough for goal-directed activities while supporting lower pace and lower granularity notification for awareness purposes.

The last requirement (c) for notification mechanisms corresponds to volume impedance matching. Although volume impedance can be supported within the GtK infrastructure, it has not been implemented in this work. Volume impedance can largely be met by having different forms of application-specific, low-granularity update events. However, more generic approaches to this issue would be a valuable extension to this work.

The GtK framework thus implements both requirements (a) and (b) for a Web-based feedthrough and awareness infrastructure. Table 11.3 compares the main functionalities of GtK with some existing notification systems.

notification system	*functionalities*
GtK	• pure notification service * • pace impedance matching support feedthrough and awareness * • explicit notification to improve user-level performance *
NSTP	• tightly bound notification service • cannot be re-used with different types of data • no consideration for time and pace issues
Elvin	• pure notification service * • only operates at the system level with little explicit user notification • no impedance matching

Table 11.3 Comparing GtK with other notification systems

Although Elvin (Fitzpatrick et al., 1999) supports a 'pure' notification service, it primarily operates at the system level by allowing applications to exchange notifications. It has very little explicit support for user notification and impedance matching is non-inexistent.

NSTP (Patterson et al., 1996) instead offers a bundled solution that combines the data service and the notification service. This restricts its capability for reuse in a different environment with different types of data. Performance considerations of time and pace are also absent.

GtK is the only extant notification service that embodies true separability from data, while also providing pace impedance matching to improve temporal behaviour, thus increasing user-level performance and reducing network load.

11.4 Final remark

Timing issues are becoming increasingly more important on both large scale group work and fine scale networked environments. Delays and temporal properties are vital concerns in the design of ubiquitous and mobile devices. Thus, the understanding of temporal problems and the management of delays are likely to become ever more significant.

As the thesis is being finalised, a whole session was devoted to the issues of time in the recent CSCW conference where some studies (Begole et al., 2002), (Reddy and Dourish, 2002) have applied the notion of rhythm (Zerubavel, 1985) for interpreting the temporal pattern of work iterated over time and for coordinating work. By integrating rhythms in system design, people not only become aware of their current activities but they can also see how those activities are related to past activities and how they may influence future activities.

Current research on notification servers (Shen and Sun, 2002) is also drawing on the work on notification server taxonomy described in the thesis. Timing issues are already playing a major role in the provision of awareness or notification services and their importance will continue in the future.

References

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

(Anderson, 1994) Anderson, R. J. (Ed.) (1994) *Representations and Requirements : The Value of Ethnography in System Design*, Human-Computer Interaction, Vol. 9, Lawrence Erlbaum pp. 151-182.

(Baecker et al., 1995) Baecker, R., Grudin, J., Buxton, B. and Greenberg, S. (Eds.) (1995) *Readings in Human-Computer Interaction:Towards the Year 2000*, Second edition, Morgan-Kaufman pp. 950.

(Barth, 1986) Barth, P. S. (1986) An Object-Oriented Approach to Graphical Interfaces, In *ACM Transactions on Graphics*, **5** (2), pp. 142-172.

(Bass, 1993) Bass, L. (1993) Architectures for Interactive Software System: Rationale and Design, In *Trends in Software Issue on User Interface Software*, **1**, pp. 31-44.

(Beaudouin-Lafon and Karsenty, 1992) Beaudouin-

(Begeman et al., 1986) Begeman, M., Cook, P., Ellis, C., Graf, M., Rein, G. and Smith, T. (1986) Project Nick: meetings augmentation and analysis, *Proceedings of the ACM 1986 conference on Computer Supported Cooperative Work*, Austin, Texas, ACM Press.

(Begole et al., 2002) Begole, J. B., Tang, J. C., Smith, R. B. and Yankelovich, N. (2002) Work Rhythms: Analyzing Visualisations of Awareness Histories of Distributed Systems, *Proceedings of Computer Supported Collaborative Work (CSCW 2002)*, New Orleans, USA, ACM Press, pp. 334-343.

(Benford et al., 1993) Benford, S., Bullock, A., Cook, C., Harvey, P., Ingram, P. and Lee, O. (1993) From room to cyberspace: models of interaction in large virtual computer spaces, In *Interacting with Computers*, **5** (2), pp. 217-237.

(Benford and Fahlén, 1993) Benford, S. and Fahlén, L. (1993) A Spatial model of Interaction in Large virtual Environments, *Proceedings of*

the third European Conference on CSCW, ECSCW'93, Sept., Milan, Italy, Kluwer Academic, pp. 109-124.

(Benford et al., 1994a) Benford, S., Bowers, J., Fahlén, L., Mariani, J. and Rodden, T. (1994) Supporting Cooperative Work in Virtual Environments, In *The Computer Journal*, **37** (8), pp. 653-668.

(Benford et al., 1994b) Benford, S., Fahlen, L., Greenhalge, C. and Bowers, J. (1994) Managing mutual awareness in collaborative virtual environments, *ACM SIGCHI conference on Virtual Reality and Technology (VRST'94)*, Singapore, ACM Press.

(Benford et al., 1997) Benford, S., Greenhalgh, C. and Lloyd, D. (1997) Crowded Collaborative Virtual Environments, *Proceedings of CHI'97*, March 22-27, Atlanta, Georgia, pp. 59-66.

(Bentley et al., 1992a) Bentley, R., Hughes, J. A., Randall, D., Rodden, T., Sawyer, P., Shapiro, D. and Sommerville, I. (1992) Ethnographically-informed systems design for air traffic control, *Proceedings of CSCW'92*, Nov., Toronto, Ontario, ACM Press, pp. 123-129.

(Bentley et al., 1992b) Bentley, R., Hughes, J. A., Randall, D. and Shapiro, S. Z. (1992) Technological support for decision making in a safety critical environment, Computing Department, Lancaster University, CSCW/5/92.

(Bentley, 1994) Bentley, R. (1994) Supporting Multi-User Interface Development for Cooperative Systems, *PhD Thesis*, University of Lancaster, UK.

(Bentley et al., 1994) Bentley, R., Rodden, T., Sawyer, P. and Sommerville, I. (1994) Architectural support for cooperative multi-user interfaces, In *IEEE COMPUTER special issue on CSCW*, **27** (5), pp. 37-46.

(Bentley et al., 1996) Bentley, R., Horstmann, T., Sikkel, K. and Trevor, J. (1996) The BSCW Shared Workspace System, In *ERCIM workshop on CSCW and the Web* (Eds, U. Busbach, D. Kerr and K. Sikkel), GMD/FIT, Sankt Augustin, Germany.

(Bentley, 1997) Bentley, R. (1997) Time and the Web: Experiences from BSCW, *Time and the Web Seminar*, June, Staffordshire University.

(Bentley et al., 1997a) Bentley, R., Appelt, W., Busbach, U., Hinrichs, E., Kerr, D., Sikkel, K., Trevor, J. and Woetzel, G. (1997) Basic Support for Cooperative Work on the World Wide Web, In *International Journal of Human-Computer Studies, special issue on 'Innovative Applications of the World Wide Web'*, **46**, pp. 827-846.

(Bentley et al., 1997b) Bentley, R., Horstmann, T. and Trevor, J. (1997) The World Wide Web as Enabling Technology for CSCW: The Case of BSCW, In *Computer Supported Cooperative Work: The journal of Collaborative Computing*, **6**, pp. 111-134.

(Berners-Lee et al., 1994) Berners-Lee, T., Cailliau, R., Luotonen, A., Frystyck Nielsen, H. and Secret, A. (1994) The World Wide Web, In *Communications of the ACM*, **37** (8), pp. 76-82.

(Bier and Freeman, 1992) Bier, E. and Freeman, S. (1992) MMM: A user interface architecture for shared editors on a single screen, *Proceedings of UIST'91*, Hilton Head, pp. 79-86.

(Brewster, 1994) Brewster, S. A. (1994) Providing a structured method for integrating non-speech audio into human-computer interfaces, *PhD Thesis*, University of York, UK.

(Brewster et al., 1994) Brewster, S. A., Wright, P. C. and A.D.N., E. (1994) The design and evaluation of an auditory-enhanced scrollbar, *Proceedings of CHI'94*, Boston, Massachusett, ACM Press, pp. 173-179.

(Brink and Gomez, 1992) Brink, T. and Gomez, L. M. (1992) A Collaborative Medium for the Support of Conversational Props, *Proceedings of CSCW'92*, Nov., Toronto, Ontario, ACM Press, pp. 171-178.

(Byrne and Picking, 1997) Byrne, A. and Picking, R. (1997) Is Time Out to be the Big Issue?, *Time and the Web Seminar*, June, Staffordshire University.

(Card et al., 1983) Card, S. K., Moran, T. P. and Newell, A. (1983) *The psychology of Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, New Jersey.

(Card et al., 1991) Card, S. K., Robertson, G. C. and Mackinlay, J. D. (1991) The information visualizer: An information workspace, *CHI'91 Conference Proceedings : Human Factors In computing Systems*, 28 April-2 May, New Orleans, LA, ACM Press, pp. 181-188.

(Computer, 1985) Computer, A. (1985) *Inside Macintosh,* Addison-Wesley.

(Conn, 1995) Conn, A. P. (1995) Time affordances: the time factor in diagnostic usability heuristic, *CHI'95 Conference Proceedings: Human Factors In computing*, ACM Press, pp. 186-193.

(Coutaz, 1987) Coutaz, J. (1987) PAC, An Object Oriented Model For Dialog Design, *Human-Computer Interaction - INTERACT '87* (Eds, H.J. Bullinger and B. Shackel), pp. 431-436.

(Crowley et al., 1990) Crowley, T., Milazzo, P., Baker, E., Forsdick, H. and Tomlinson, R. (1990) MMConf: An infrastructure for building shared multimedia application, *Proceedings of CSCW'90*, ACM Press, pp. 329-342.

(Curtis et al., 1988) Curtis, B., Krasner, H. and Iscoe, N. (1988) A Field Study of the Software Design Process for Large Systems, In *Communications of ACM*, **31** (11), pp. 1268-1287.

(Cypher, 1986) Cypher, A. (1986) The Structure of Users' Activities, In *User Centred System Design - New Perspectives on Human Computer Interaction* (Eds, D.A Norman and S. Draper Lawrence), Erlbaum Associates, pp. 243-263.

(Dewan, 1990) Dewan, P. (1990) A tour of the Suite user interface software, *UIST'90: Proceedings of 3rd ACM SIGGRAPH Symposium on User Interface Software and Technology*, ACM Press, pp. 57-65.

(Dewan and Choudhary, 1991) Dewan, P. a

(Dewan, 1992) Dewan, P. (1992) Principles of designing multi-user user interface development environments, *Proc. 5th IFIP Working Conf. on Engineering for HCI* (Eds, J. Larson and C. Unger), August, Ellivuori , Finland, pp. 35-48.

(Dewan and Choudary, 1992) Dewan, P. and Choudary, R. (1992) A High-level and Flexible Framework for Implementing MultiUser User Interfaces, In *ACM Transactions on Information Systems*, **10** (4), pp. 345-380.

(Dewan, 1993) Dewan, P. (1993) Tools for Implementing Multiuser User Interfaces, In *Trends in Software: Issue on User Interface Software*, **1**, pp. 149-172.

(Diaper, 1989) Diaper, D. (1989) Task Analysis for Knowledge Descriptions (TAKD); the method and an example, In *Task Analysis for Human-Computer Interaction* (Ed, D. Diaper), Chapter 4, Ellis Horwood, Chichester, pp. 108-159.

(Dix et al., 1993) Dix, A., Finlay, J., Abowd, G. and Beale, R. (1993) *Human-Computer Interaction*, second edition 1998, Prentice Hall.

(Dix, 1994a) Dix, A. (1994) Que sera sera - The problem of the future perfect in open and cooperative systems., *Proceedings of HCI'94: People and Computers IX*, Glasgow, Cambridge University Press, pp. 397-408.

(Dix et al., 1995) Dix, A., Ramduny, D. and Wilkinson, J. (1995) Interruptions, Deadlines and Reminders: Investigations into the Flow of Cooperative Work, University of Huddersfield, RR9509.

(Dix and Abowd, 1996a) Dix, A. and Abowd, G. (1996) Modelling status and event behaviour of interactive systems, In *Software Engineering Journal*, **11** (6), pp. 334-346.

(Dix and Abowd, 1996b) Dix, A. and Abowd, G. (1996) Delays and Temporal Incoherence Due to Mediated Status-Status Mappings, In *SIGCHI Bulletin*, **28** (2), pp. 47-49.

(Dix et al., 1996) Dix, A., Ramduny, D. and Wilkinson, J. (1996) Long-Term Interaction: Learning the 4Rs, *CHI'96 Conference Companion Proceedings: Human Factors In computing Systems*, Apr., Vancouver, British Columbia, ACM Press, pp. 169 -170.

(Dix, 1997) Dix, A. (1997) Challenges and Perspectives for Cooperative Work on the Web: An Analytical Approach, In *CSCW: The Journal of Collaborative Computing*, **6** (2-3), pp. 135-156.

(Dix, 1998) Dix, A. (1998) Finding Out - event discovery using status-event analysis, *Formal Aspects of Human Computer Interaction FAHCI98*, 5-6th September 1998, Sheffield.

(Dix et al., 1998) Dix, A., Ramduny, D. and Wilkinson, J. (1998) Interaction in the large, In *Interacting With Computers, Special Issue on Temporal Aspects of Usability*, **11**, pp. 9-32.

(Dix et al., 2000) Dix, A., Ramduny, D., Rodden, T. and Davies, N. (2000) Places to stay on the move: software architectures for mobile user interfaces, In *Personal Technologies - Special Issue on Human Computer Interaction with Mobile Devices*, **4** (2), pp. 171-181.

(Dix et al., 2003) Dix, A., Ramduny, D. and Wilkinson, J. (2003) Trigger Analysis: understanding broken tasks, In *The Handbook of Task Analysis for Human-Computer Interaction* (Eds, D. Diaper and N. Stanton), Lawrence Erlbaum Associates.

(Dix, 1987) Dix, A. J. (1987) The Myth of the Infinitely Fast Machine, *Proceedings of the Third Conference of the BCS HCI SIG: People and Computers III*, Cambridge University Press, pp. 215-228.

(Dix, 1991) Dix, A. J. (1991) *Formal Methods for Interactive Systems*, Academic Press.

(Dix, 1992a) Dix, A. J. (1992) Pace and interaction, *Proceedings of HCI'92: People and Computers VII*, Sept., York, Cambridge University Press, pp. 193-208.

(Dix, 1992b) Dix, A. J. (1992) Beyond the Interface, *Engineering for Human-Computer Interaction: Proceedings of IFIP TC2/WG2.7 Working Conference*, Ellivuori, Finland, North-Holland.

(Dix, 1994b) Dix, A. J. (Ed.) (1994) *Computer-supported cooperative work - a framework*, Design Issues in CSCW, D. Rosenburg and C. Hutchison (Series Eds.), Springer Verlag, Berlin pp. 9-26.

(Dix, 1995a) Dix, A. J. (1995) Cooperation without (reliable) Communication: Interfaces for Mobile Applications, In *Distributed Systems Engineering*, **2** (3), pp. 171-181.

(Dix, 1995b) Dix, A. J. (1995) LADA-A logic for the analysis of distributed action, *Interactive Systems: Design, Specification and Verification* (Ed, F. Paternó), 1st Eurographics Workshop, Bocca di Magra, Italy, June 1994, Springer Verlag, Berlin, pp. 317-332.

(Dourish and Bellotti, 1992) Dourish, P. and Bellotti, V. (1992) Awareness and coordination in shared workspaces, *CSCW'92*, Toronto, Canada, ACM Press, pp. 107-114.

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

(Edmonds, 1992) Edmonds, E. A. (Ed.) (1992) *The Separable User Interface*, Computer and People Series, Academic Press.

(Ellis et al., 1990) Ellis, C. A., Gibbs, S. J. and Rein, G. L. (1990) Design and use of a group editor, *Proceedings of the IFIP Engineering for Human-Computer Interaction Conference* (Ed, G. Cockton), North-Holland, Amsterdam, pp. 13-15.

(Ellis et al., 1991) Ellis, C. A., Gibbs, S. J. and Rein, G. L. (1991) Groupware: Some issues and experiences, In *Communications of the ACM*, **34** (1), pp. 38-58.

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

(Engelbart, 1975) Engelbart, D. (1975) NLS Teleconferencing Features, In *Proceedings of Fall COMPCON*, pp. 173-176.

(Ensor et al., 1988) Ensor, J. R., Ahuja, S. R., Horn, D. N. and Lucco, S. E. (1988) The Rapport Multimedia Conferencing System: A Software Overview, *Proceedings of the 2nd IEEE Conference on Computer Workstations*, March, pp. 52-58.

(Erickson et al., 1999) Erickson, T., Smith, D. N., Kellogg, W. A., Laff, M. R., Richards, J. T. and Bradner, E. (1999) Socially translucent systems: Social proxies, persistent conversation and the design of 'Babble', *Proceedings of CHI'99*, May 15-20, Pittsburgh, PA, ACM, New York, pp. 72-79.

(Fielding et al., 1997) Fielding, R., Gettys, J., Moghul, J., Frystyk, H. and Berners-Lee, T. (1997) Hypertext Transfer Protocol -- HTTP/1.1, In *RFC 2068*, U.C. Irvine, DEC, MIT/LCS.

(Fitzpatrick et al., 1995) Fitzpatrick, G., Tolone, W. and Kaplan, S. (1995) Work, Locales and Distributed Social Worlds, *ECSCW'95*, 10-14 September 1995, Stockholm, Dordrecht: Kluwer, pp. 1-16.

(Fitzpatrick et al., 1999) Fitzpatrick, G., Mansfield, T., Kaplan, S., Arnold, D., Phelps, T. and Segall, B. (1999) Augmenting the workaday world with Elvin, *Proceedings of the sixth European Conference on Computer Supported Cooperative Work (ECSCW'99)* (Eds, S. Bødker, M. Kyng and K. Schmidt), 12-16 September 1999, Copenhagen, Denmark, Kluwer Academic Publishers, Netherlands, pp. 431-450.

(Flannagan, 1997) Flannagan, D. (1997) *Java in a Nutshell*, 2nd Edition (Java 1.1), O'Reilly.

(Foundation, 1989) Foundation, O. S. (1989) OSF/Motif, Programmer's Reference Manual., In *Open Software Foundation*.

(Garfinkel, 1967) Garfinkel, H. (1967) *Studies in Ethnomethodology*, Prentice Hall, Englewood Cliffs, NJ.

(Gaver et al., 1992) Gaver, W., Moran, T., MacLean, A., Lövstrand, L., Dourish, P., Carter, K. and Buxton, W. (1992) Realizing a Video Environment: EuroPARC's RAVE System, *Proceedings of CHI'92: Human Factors in Computing Systems* (Eds, P.Bauersfield, J.Bennett and G. Lynch), ACM Press, pp. 27-35.

(Gillian and Breedin, 1990) Gillian, D. J. and Breedin, S. D. (1990) "Designers" Models of Human-Computer Interface, *Proceedings of SICCHI'90*, Apr. 1990, Seattle, WA, pp. 391-398.

(Gleick, 2000) Gleick, J. (2000) *Fa:st:er the acceleration of just about everything*, Reissue 6 July, 2000, Abacus, London.

(Goland et al., 1999) Goland, Y. Y., J., W. J. E., Faizi, A., Carter, S. R. and Jensen, D. (1999) HTTP Extensions for Distributed Authoring -- WEBDAV, In *RFC 2518*, Microsoft, U.C. Irvine, Netscape, Novell.

(Gram and Cockton, 1996) Gram, C. and Cockton, G. (Eds.) (1996) *Design Principles for Interactive Software*, Chapman & Hall, UK.

(Grasso et al., 1997) Grasso, A., Meunier, J., Pagani, D. and Pareschi, R. (1997) Distributed Coordination and Workflow on the World Wide Web, In *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, pp. 175-200.

(Gray et al., 1994) Gray, P., England, D. and McGowan, S. (1994) XUAN: Enhancing UAN to Capture Temporal Relationships among

Actions, *Proceedings of HCI'94: People and Computers IX*, Glasgow, Cambridge University Press, pp. 301-312.

(Greenberg, 1990) Greenberg, S. (1990) Sharing views and interactions with single-user applications, *Proceedings of COIS'90*, Cambridge, Massachussets, pp. 227-237.

(Greenberg et al., 1992a) Greenberg, S., Roseman, M., Webster, D. and Bohnet, R. (1992) Issues and experiences in implementing two group drawing tools, *IEEE Proc. 25th Annual Hawaii Intl. Conf. System Sciences*, Vol. 4, pp. 139-150.

(Greenberg et al., 1992b) Greenberg, S., Roseman, M., Webster, D. and Bohnet, R. (1992) Human and technical factors of distributed group drawing tools, In *Interacting with Computers*, **4** (3), pp. 364-392.

(Greenhalgh and Benford, 1995) Greenhalgh

(Greenhalgh et al., 2000) Greenhalgh, C., Purbrick, J. and Snowdon, D. (2000) Inside MASSIVE-3: flexible support for data consistency and world structuring, *Proceedings of the third international conference on Collaborative virtual environments*, ACM Press, pp. 119-127.

(Gust, 1988) Gust, P. (1988) SharedX: X in a distributed group work environment, *2nd annual X Conference*.

(Hall et al., 1996) Hall, R. W., Mathur, A., Jahanian, F., Prakash, A. and Rassmussen, C. (1996) Corona: A Communication Service for Scalable, Reliable Group Collaborative Systems, *ACM Conference on Computer Supported Cooperative Work (CSCW 96)*, Nov. 1996., Boston, MA, ACM Press, pp. 140-149.

(Hammersley and Atkinson, 1995) Hammersle

(Hayne et al., 1993) Hayne, S., Pendergast, M. and Greenberg, S. (1993) Gesturing through cursors: Implementing multiple pointers in group support systems, *Proceedings of the HICSS Hawaii International Conference on System Sciences*, Vol. IV, January 1993, Los Alamitos, Calif., IEEE Computer Society, pp. 4-12.

(Heath et al., 1993) Heath, C., Jirokta, M., Luff, P. and Hindmarsh, J. (1993) Unpacking Collaboration: The Interactional Organisation of Trading in a City Dealing Room, *Proceedings of*

ECSCW'93, Sept., Milan, Italy, Kluwer Academic Publishers, Dordrecht, pp. 155-171.

(Heath and Luff, 1994) Heath, C. and Luff, P. (1994) Crisis management and multimedia technology in London Underground line control rooms, In *Journal of CSCW*, **1** (1-2), pp. 69-94.

(Herskind, 1997) Herskind, S. (1997) Computer support for temporal aspects of coordination of cooperative work, *ECSCW'97 Conference Supplement*, Lancaster, UK, Kluwer Academic Press, Dordrecht, pp. 67.

(Hill, 1992) Hill, R. D. (1992) The Abstraction-Link-View Paradigm: Using Constraints to connect User Interfaces to Applications, *Proceedings of CHI'92*, ACM Press, pp. 335-343.

(Hill et al., 1994) Hill, R. D., Brinck, T., Rohall, S. L., Patterson, J. F. and Wilner, W. (1994) The Rendezvous Architecture and Language for Constructing Multiuser Applications, In *ACM Transactions on Computer-Human Interaction*, **1** (2), pp. 81-125.

(Hix, 1990) Hix, D. (1990) Generations of User-Interface Management Systems, In *IEEE Software*, (Sept.), pp. 77-87.

(Hudson et al., 2002) Hudson, J. M., Christensen, J., Kellogg, W. A. and Erickson, T. (2002) "I'd be overwhelmed, but it's just one more thing to do": Availability and interruption in research management, *Proceedings ACM Conference on Human Factors in Computing Systems (CHI'02)*, ACM Press, pp. 97-104.

(Ingram et al., 1996) Ingram, R. J., Benford, S. D. and Bowers, J. M. (1996) Building Virtual Cities: Applying Urban Planning principles to the Design of Virtual Environments, *Proceedings VRST'96*, July 1-4, Hong Kong, ACM Press, pp. 83-91.

(Johnson and Gray, 1995) Johnson, C. and Gray, P. (Eds.) (1995) *Workshop on Temporal Aspects of Usability*, 2, Vol. 28, SIGCHI Bulletin.

(Johnson et al., 1995) Johnson, C. W., McCarthy, J. and Wright, P. C. (1995) Using Petri Nets to support natural language in accident reports, In *Ergonomics*, **38** (6), pp. 1265-1283.

(Johnson, 1997) Johnson, C. W. (1997) The impact of time and place on the operation of mobile computing devices, *Proceedings of*

HCI'97: People and Computers XII, Bristol, UK, pp. 175-190.

(Joosten, 1994) Joosten, S. (1994) Trigger modelling for workflow analysis, *Proceedings of CON'94: Workflow Management* (Ed, R. Oldenbourg), Vienna, pp. 236-247.

(Knight and Munro, 1998) Knight, C. and Munro, M. (1998) Using an Existing Game Engine to Facilitate Multi-User Software Visualization, In *Second Annual Workshop on System Aspects of Sharing a Virtual Reality.*

(Krasner and Pope, 1988) Krasner, G. E. and Pope, S. T. (1988) A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, In *Journal of Object Oriented Programming*, **1** (3), pp. 26-49.

(Kuhmann et al., 1987) Kuhmann, W., Boucsein, W., Schaefer, F. and Alexander, J. (1987) Experimental investigation of Psychophysical Stress-Reactions induced by different Response Times in Human-Computer Interaction, In *Ergonomics*, **30** (6), pp. 933-943.

(Kutar, 2001) Kutar, M. S. (2001) Specification of Temporal Properties of Interactive Systems, *PhD Thesis*, University of Hertfordshire, UK.

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

(Lantz, 1986) Lantz, K. A. (1986) An Experiment in Integrated Multimedia Conferencing, *Proceedings of Conference on Computer-Supported Cooperative Work*, December, pp. 267-275.

(Lauwers and Lantz, 1990) Lauwers, J. C. and Lantz, K. A. (1990) Collaboration Awareness in support of Collaboration Transparency: Requirements for the next generation of shared window systems, *CHI'90 Conference Proceedings: Human Factors computing Systems*, Apr., Seattle, Washington, ACM Press, pp. 303-311.

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

(Leland et al., 1988) Leland, M. D. P., Fish, R. S. and Kraut, R. E. (1988) Collaborative document production using quilt, *Proceedings of CSCW'88*, Sept., Portland Oregon, ACM Press, pp. 206-215.

(Lewis, 1995) Lewis, S. (1995) *The Art and Science of Smalltalk*, Hewlett-Packard, Prentice Hall.

(Linton, 1993) Linton, M. A. (1993) Making User Interfaces Easy-to-Build, In *User Interface software* (Eds, L. Bass and P. Dewan), pp. 45-59.

(Long, 1976) Long, J. (1976) Effects of delayed irregular feedback on unskilled and skilled keying performance, In *Ergonomics*, **19** (2), pp. 183-202.

(Macedonia et al., 1994) Macedonia, M., Zyda, M., Pratt, D., Barham, P. and Zeswitz, S. (1994) NPSNET: A Network Software Architecture for Large-Scale Virtual Environments, In *Presence: Teleoperators and Virtual Environments*, **3** (4), pp. 265-287.

(McManus, 1997) McManus, B. (1997) Compensatory Actions for Time Delays, *Time and the Web Seminar*, June, Staffordshire University.

(Microsoft, 1993) Microsoft (1993) Microsoft Windows, Version 3.1.

(Microsystems, 1996) Microsystems, S. (1996) Java: Programming for the Internet

(Miller, 1956) Miller, G. A. (1956) The magical number seven, plus or minus two: some limits on our capacity to process information., In *Psychological Review*, **63** (2), pp. 81-97.

(Miller, 1968) Miller, R. B. (1968) Response time in man-computer conversational transactions, 33, Proceedings of the AFIPS Fall joint Computer Conference, pp. 267-277.

(Miyata and Norman, 1986) Miyata, A. and Norman, D. A. (1986) Psychological Issues in Support of Multiple Activities, In *User Centred System Design - New Perspectives on Human Computer Interaction* (Eds, D.A. Norman and S. Draper), Lawrence Erlbaum Associates, pp. 265-284.

(Myers, 1985) Myers, B. A. (1985) The importance of percent-done indicators for computer-human interfaces., *Proceedings of CHI'85: Human Facors in Computing Systems*, 14-18 April, San Francisco, CA, ACM Press, pp. 11-17.

(Myers, 1989) Myers, B. A. (1989) User-Interface Tools: Introduction and survey, In *IEEE Software*, pp. 15-23.

(Myers, 1990) Myers, B. A. (1990) A New Model for Handling Input, In *ACM Transactions on Information Systems*, **8** (3), pp. 289-320.

(Myers et al., 1990) Myers, B. A., Guise, D. A., Dannenburg, R. B., Vander Zanden, B., Kosbie, D. S., Pervin, E., Mickish, A. and Marchal, P. (1990) Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces, In *IEEE Computer*, **28** (11), pp. 71-85.

(Myers, 1991) Myers, B. A. (1991) Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs, *SIGGRAPH Symposium on Users Interface Software and Technology (UIST'91)*, Hilton Head, South Carolina, pp. 211-220.

(Myers, 1995) Myers, B. A. (1995) User Interface Software Tools, In *ACM Transactions on Computer-Human Interaction*, **2** (1), pp. 64-103.

(Newell and Simon, 1972) Newell, A. and Simon, H. A. (1972) *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, New Jersey.

(Newman, 1968) Newman, W. M. (1968) A System for Interactive Graphical Programming, *Proceedings of the Spring Joint Computer Conference*, Atlantic City, NJ, AFIPS Press, pp. 47-54.

(Nielsen, 1993) Nielsen, J. (1993) *Usability Engineering*, ACM Press.

(Nielsen, 1995) Nielsen, J. (1995) *Multimedia and Hypertext: The Internet and Beyond*, AP Professional, Boston, MA.

(Nielsen, 1997) Nielsen, J. (1997) The need for Speed, Alert box, http://www.useit.com/alertbox/9703a.html

(Nigay and Coutaz, 1993) Nigay, L. and Coutaz, J. (1993) A Design Space for Multimodal Systems: Concurrent Processing and Data Fusion, *Proceedings of INTERCHI'93*, April 1993, pp. 172-178.

(Norman, 1984) Norman, D. A. (1984) Stages and Levels in Man-Machine Interaction, In *International Journal of Man-Machine Studies*, **21**, pp. 365-375.

(Norman, 1986) Norman, D. A. (Ed.) (1986) *New views of information processing: Implications for intelligent decision support systems*, Intelligent Decision Support Process Environments, E. H. et al. (Series Eds.), Springer-Verlag.

(Norman, 1988) Norman, D. A. (1988) *The Psychology of Everyday Things*, Basic Books, New York.

(Olsen Jr, 1992) Olsen Jr, D. (1992) *User Interface Management Systems: Models and Algorithms*, Morgan Kaufmann.

(OSF, 1995) OSF (1995) *OSF/Motif Programmer's Guide*, Revision 2, Open Software Foundation, Prentice Hall.

(Ousterhout, 1994) Ousterhout, J. K. (1994) *An Introduction to Tcl and Tk*, Addison-Wesley.

(Palanque and Bastide, 1995) Palanque, P. and Bastide, R. (1995) Formal specification and verification of CSCW, *Proceedings of the HCI'95 Conference: People and Computers X*, Huddersfield, UK, Cambridge University Press, pp. 213-231.

(Palay, 1988) Palay, A. e. a. (1988) The Andrew toolkit: An Overview, *Winter USENIX Technical conference*, Dallas, Texas, pp. 9-12.

(Palfreyman and Rodden, 1996 Palfreyman

(Paternó and Faconti, 1992) Paternó, F. and Faconti, G. (1992) On the use of LOTOS to describe graphical interaction, *Proceedings of the HCI'92 Conference: People and Computers VII*, Cambrigde University Press, pp. 155-173.

(Patterson et al., 1990) Patterson, J. F., Hill, R. D., Rohall, L. and Meeks, W. S. (1990) Rendezvous: An architecture for synchronous multi-user application, *Proceedings of CSCW'90*, ACM Press, pp. 317-328.

(Patterson, 1991) Patterson, J. F. (1991) Comparing the Programming Demands of Single-User and Multi-User Application, In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp. 87-94.

(Patterson et al., 1996) Patterson, J. F., Day, M. and Kucan, J. (1996) Notification Servers for Synchronous Groupware, *Proceedings of CSCW'96*, Nov. 1996, Boston, Massachusetts, ACM Press, pp. 122-129.

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

(Pausch et al., 1992) Pausch, R., Conway, M. and DeLine, R. (1992) Lessons learned from SUIT, the Simple User Interface Toolkit, In *ACM Transactions of Office Information Systems,* **104** (4), pp. 320-344.

(Payne, 1993) Payne, S. J. (1993) Understanding Calendar Use, In *Human-Computer Interaction*, **8** (2), pp. 83-100.

(Pfaff and Hagen, 1985) Pfaff, G. and Hagen, P. J. W. (Eds.) (1985) *Seeheim Workshop on User Interface Management Systems*, Berlin, Springer-Verlag.

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

(Ramduny, 1994) Ramduny, D. (1994) Increasing User Awareness in UNIX, *B.Sc. Project Report*, Lancaster University, UK.

(Ramduny, 1996) Ramduny, D. (1996) Temporal Interface Issues and Software Architecture for Remore Cooperative Work, *CSCW'96 Doctoral Colloquium*.

(Ramduny and Dix, 1997a) Ramduny, D. and Dix, A. (1997) Why, What, Where, When: Architectures for Cooperative Work on the World Wide Web, *Proceedings of HCI'97*, Aug. 1997, Bristol, UK, Springer-Verlag, pp. 283-301.

(Ramduny and Dix, 1997b) Ramduny, D. and Dix, A. (1997) In the Right Place at the Right Time: Placement Options for Web-based Architectures, *ECSCW'97 Conference Supplement*, Sep, 1997, Lancaster, UK, pp. 37-38.

(Ramduny et al., 1998) Ramduny, D., Dix, A. and Rodden, T. (1998) Exploring the design space for notification servers., *Proceedings of CSCW'98*, Nov. 14-18, Seattle, Washington, ACM Press, pp. 227-235.

(Ramduny, 1999) Ramduny, D. (1999) Impedance Matching: Enhancing temporal interactivity on the web, *Proceedings of The Active Web, A British HCI Group Day Conference* (Eds, Dave Clarke, Alan Dix and Fiona Dix), January, Staffordshire University, UK, pp. 227-235.

(Ramduny and Dix, 2002) Ramduny, D. and Dix, A. (2002) Impedance Matching: When You Need to Know What, *People and Computers XVI: memorable yet invisible: proceedings of HCI 2002* (Eds, X. Faulkner, J. Finlay and F. Détienne), London, UK, Springer-Verlag, pp. pp 121 - 137.

(Randall, 1995) Randall, D. (1995) Ethnography for Systems Development: Bounding the Intersection, University of Huddersfield, Tutorial Notes HCI'95.

(Reddy and Dourish, 2002) Reddy, M. and Dourish, P. (2002) A Finger on the Pulse: Temporal Rythms and Information Seeking in Medical Work, *Proceedings of Computer Supported Collaborative Work (CSCW 2002)*, New Orleans, USA, ACM Press, pp. 344-353.

(Reeves, 1996) Reeves, S. (1996) Specifying and reasoning about CSCW, *Design, Specification and Verification of Interactive Systems '96*, Namur, Belgium, Springer Verlag, Berlin, pp. 366-383.

(Rein and C., 1991) Rein, G. and C., E. (1991) rIBIS: a real-time group hypertext system, In *International Journal of Man Machine Studies*, **34** (3), pp. 349-368.

(Rodden and Blair, 1991) Rodden, T. and Blair, B. (1991) CSCW and Distributed Systems: The problem of Control, *Proceedings of the second European Conference on CSCW (ECSCW'91)* (Eds, L. Bannon, L. M. Robinson and K. Schmidt), September 25-27, Amsterdam, The Netherlands, Kluwer Academic Publishers, pp. 49-64.

(Rodden, 1996) Rodden, T. (1996) Populating the Application: A Model of Awareness for Cooperative Applications, *Proceedings of CSCW'96*, Nov. 1996, Boston, Massachusetts, ACM Press, pp. 87-96.

(Rohall et al., 1992) Rohall, S. L., Patterson, J. F. and Hill, R. D. (1992) Go Fish! A Multi-User Game in the Rendezvous System, *SIGGRAPH Video* Review 76, ACM, NewYork.

(Roseman and Greenberg, 1992) Roseman, M

(Rouncefield et al., 1994) Rouncefield, M., Hughes, J. A., Rodden, T. and Viller, S. (1994) Working with 'Constant Interruption' CSCW and the Small Office, *Proceedings of CSCW'94*, Oct., Chapel Hill, North Carolina, ACM Press, pp. 275-286.

(Sandor et al., 1997) Sandor, O., Bogdan, C. and Bowers, J. (1997) Aether: an awareness engine for CSCW, *Proceedings of ECSCW'97*, Lancaster, UK,, Kluwer Academic, pp. 221-236.

(Satyanarayanan et al., 1990) Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H. and Steere, D. C. (1990) Coda: a highly available file system for a distributed workstation environment, In *IEEE Transactions Computers*, **39** (4), pp. 447-459.

(Sawyer and Mariani, 1995) Sawyer, P. and Mariani, J. A. (1995) Database systems: challenges and opportunities for graphical HC, In *Interacting with Computers: the Interdisciplinary Journal of Human-Computer Interaction*, **7** (3), pp. 273-303.

(Scheifler and Gettys, 1986) Scheifler, R. and Gettys, J. (1986) The X Window System, In *ACM Transactions of Graphics*, **5** (2), pp. 79-109.

(Schneiderman, 1983) Schneiderman, B. (1983) Direct Manipulation: A Step Beyond Programming Languages, In *Computer* pp. 57-69.

(Sellen and Harper, 1997) Sellen, A. and Harper, R. (1997) Paper as an analytic resource for the design of new technologies, *Proceedings of the Conference on Human Factors In Computing Systems CHI'97*, ACM Press, pp. 319-326.

(Shen and Sun, 2002) Shen, H. and Sun, C. (2002) Flexible Notification for Collaborative Systems, *Proceedings of Computer Supported Collaborative Work (CSCW 2002)*, New Orleans, USA, ACM Press, pp. 77-86.

(Shepherd, 1995) Shepherd, A. (1995) Task analysis as a framework for examining HCI tasks, In *Perspectives on HCI: Diverse Approaches* (Eds, A. Monk and N. Gilbert), Academic Press, London, pp. 145-174.

(Shneiderman, 1992) Shneiderman, B. (1992) Response time and display rate, In *Designing the user interface: Strategies for effective human-computer interaction*, 2nd ed., Addison Wesley, Reading, Mass, pp. 278-301.

(Smith, 1983) Smith, D. (1983) A business case for subsecond response time: Faster is better, In *Computerworld*, Vol. 17 (16) pp. 1-11.

(Smith et al., 1989) Smith, R. B., O'shea, T., O'Malley, C., Scanlon, E. and Taylor, J. (1989) Preliminary experiments with a distributed,

multi-media, problem solving environment, *Proceedings of ECSCW'89*, pp. 19-34.

(Smith and Mosier, 1986) Smith, S. L. and Mosier, J. N. (1986) Guidelines for designing user interface software, *Mitre Corporation Report*, Mitre Corporation, MTR-9420.

(Stefik et al., 1987a) Stefik, M., Bobrow, D. G., Foster, G., S., L. and Tatar, D. (1987) "WYSIWIS revisited" early experiences with multiuser interfaces, In *ACM Transactions on Office Information System*, **5** (2), pp. 147-167.

(Stefik et al., 1987b) Stefik, M., Foster, G., Bobrow, D., Kahn, K., Lanning, S. and Suchman, L. (1987) Beyond the chalkboard: computer support for collaboration and problem solving in meetings, In *Communications of the ACM*, **30** (1), pp. 32-47.

(Suchman, 1987) Suchman, L. A. (1987) *Plans and Situated Actions: The Problem of Human-Machine Communication*, Cambridge University Press.

(Szekely, 1987) Szekely, P. (1987) Modular Implementation of Presentations, *Proc SIGCHI & GI 87*, ACM Press, New York, pp. 235-240.

(Tang, 1991) Tang, J. (1991) Findings from observational studies of collaborative work, In *International Journal of Man-Machine Studies*, **34**, pp. 143-160.

(Teal and Rudnicky, 1992) Teal, S. L. and Rudnicky, A. I. (1992) A performance model of system delay and user strategy selection, *Proceedings of CHI'92*, pp. 295-305.

(Thau, 1996) Thau, R. (1996) Design Considerations for the Apache Server API, *Computer Networks and ISDN Systems 28: Proceedings of the 5th International World Wide Web Conference*, 6-11 May, Paris, pp. 1113-1122.

(Thomas, 1998) Thomas, R. C. (1998) *Long Term Human-Computer Interaction: An Exploratory Perspective*, Springer Verlag.

(Trevor et al., 1994) Trevor, J., Mariani, J. and T., R. (1994) The use of adaptors to support cooperative work, *Proceedings of CSCW'94*, Oct. 1994, Chapel Hill, North Carolina, ACM Press, pp. 22-26.

(Trevor et al., 1997) Trevor, J., Koch, T. and Woetzel, G. (1997) MetaWeb: Bringing synchronous groupware to the World Wide Web, *Proceedings of the Fifth European Conference on Computer Supported Cooperative Work (ECSCW'97)*, September 7-11, Lancaster, UK, pp. 65-80.

(Vaghi, 2002) Vaghi, I. R. (2002) Augmenting the Virtual: Model, Architecture and Techniques for the Representation of Delay-Induced Phenomena in CVEs, *PhD thesis*, University of Nottingham, UK.

(Warboys, 1994) Warboys, B. (1994) Reflections on the relationship between BPR and software process modelling, *Proceedings of ER'94*, Berlin, Springer Verlag, pp. 1-9.

(Welie and Eliëns, 1996) Welie, V. M. and Eliëns, A. (1996) Chatting on the Web, In *Proceedings of the 5th ERCIM/W4G workshop on CSCW and the Web*, GMD/FIT, Sankt Augustin, Germany.

(Whitehead, 1997) Whitehead, J. E. J. (1997) World Wide Web Distributed Authoring and Versioning (WEBDAV) -- An Introduction, In *ACM Standard View*, **5** (1), pp. 3 - 8.

(Whitehead and Y., 1999) Whitehead, J. E. J. and Y., G. Y. (1999) WebDAV: A network protocol for remote collaborative authoring on the Web, *Proceedings of the sixth European Conference on Computer Supported Cooperative Work (ECSCW'99)* (Eds, S. Bødker, M. Kyng and K. Schmidt), Copenhagen, Denmark, Kluwer Academic Publishers, Netherlands, pp. 291-310.

(Winograd and Flores, 1986) Winograd, T. and Flores, F. (1986) *Understanding computers and cognition : a new foundation for design*, Addison-Wesley Publishing Company, Inc, New York.

(Zelezny and Langley, 1999) Zelezny, P. and Langley, A. (1999) XChat 1.2 http://xchat.linuxpower.org/doc/xchat.html

(Zerubavel, 1985) Zerubavel, E. (1985) Hidden rhythms: schedules and calendars in social life, Berkeley: University of California Press.

Appendix Case Study of Long-term Interaction

Long-term interaction poses even more problems than fast pace interaction, as the communication is more likely to be interrupted at different intervals through the process and short-term memory is volatile. This appendix describes an empirical case study that was carried out to analyse the temporal problems that users face during long-term collaborative interaction, where the tight cycle between action and feedback is broken.

Several problems may arise when the expected responses do not occur during long-term interaction. Users will for instance have to remember that they have things to do, that others should do things and also they need to infer why things happen when they do. In order to provide an insight into the ways in which team-based interactions operate on both a co-located and a remote basis, a case study was carried out based on the long-term cooperative processes associated with the running of a past HCI conference. This scenario was especially interesting as it involved many procedures that crossed organisational boundaries.

The theoretical foundation of the case study lies in the study of pace of interaction (Dix, 1992a), (Dix, 1994a), (Dix, 1995a). Although the main focus was on triggers – events that initiate the occurrence of activities, a recurrent pattern of activities and triggers was discovered and it was named the 4Rs. Issues discussed here have been reported in (Dix et al., 1995), (Dix et al., 1996), (Dix et al., 1998), (Dix et al., 2003).

Section 1 identifies the problems faced during long-term interaction. Section 2 describes the approach adopted for analysing triggers and the method used to represent the flow of work through processes and activities. Section 3 gives a detailed account of the case study. The emerging classes of triggers for activities and in particular the 4Rs – a recurrent pattern of long-term work, are then analysed in Section 4. Section 5 compares the method used in this analysis with other approaches like ethnography, business process re-engineering and work flow. Finally, Section 6 proposes some general results based on potential design heuristics. It also shows how the trigger analysis and 4Rs pattern have been validated through a practical application.

1 Problems of long-term interaction

Long-term interaction takes place at a much slower pace. The lack of short-term memory and sequenced communication can make long-term interaction even more complex than a fast pace interaction. Stretching the pace of interaction poses many problems as compared to those studied by traditional researchers in HCI.

The pace of interaction (see Section 2.4.2) is defined as the rate at which users interact with computer systems, the physical world and with one another. In many collaborative situations, the pace of communication takes place over a longer time scale. This may be

partly due to the nature of the communication medium, such as normal postal delays, or partly due to the nature of the task, for example a doctor waiting for X-ray results.

The critical point is that standard models of interaction, as typified by Norman's interaction cycle (figure 1a) concentrate on a tight cycle between action and feedback (Section 2.2). But when interaction is considered over a long-term scale, such models eventually break down.

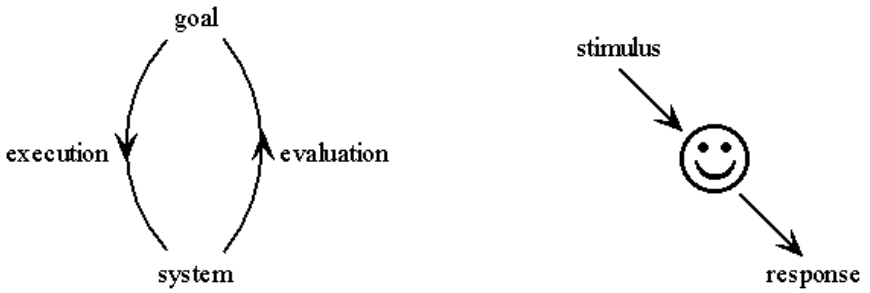


Figure 1. (a) Norman's interaction cycle (b) stimulus–response model

Another model of interaction often applied in industrial settings is to treat the worker in a stimulus–response manner (figure 1b). Commands and alarms act as stimuli and workers respond to these in the appropriate manner. However, in its pure form, this model does not allow workers to formulate any long-term plans or goals. The worker is treated in a mechanistic manner, merely a cog in the machine.

In order to incorporate both of these perspectives, the user interaction with the environment should be examined over a protracted timescale. The term environment here includes interactions with other users, computer systems or the physical environment. Such interaction is typically of a turn-taking fashion: the user acts on the environment, the environment 'responds', the user sees the effects then acts again and so on.

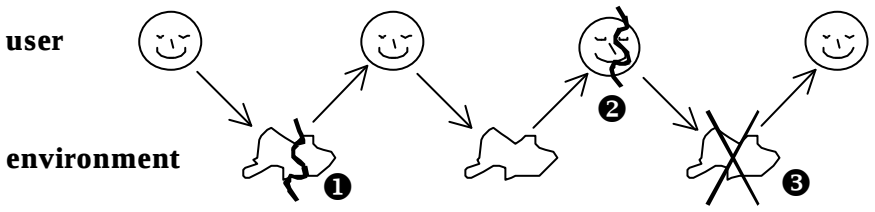


Figure 2. Problems of long-term interaction

This process is illustrated in figure 2. The Norman loop concentrates on the user–environment–user part of the interaction whereas the stimulus–response model centres on the environment–user–environment part. Long-term interaction will affect this diagram in various ways:

❶ action–effect gap – The user performs an action, but the effects of that action only becomes apparent after a long delay. The main problem here is loss of context. For instance, someone sends you an email and you respond to it but you do not receive a reply back from the sender until some days later. How do you recall the context of the message when the feedback eventually arrives? You should not only remember the reason why the original message was sent but also what sort of reply was expected. Email systems tend to address this problem by including the sender's message in the reply. In paper communications, the use of 'my ref./your ref.' fulfils a similar purpose.

❷ stimulus–response gap – The user must respond to some event, but for some reason cannot do so immediately. For example, at a chance meeting in the corridor, someone asks you to do something. The problem here is that you may forget, hence the need for to-do lists or other forms of reminders. In the psychological literature this has been called prospective memory (Payne, 1993).

❸ missing stimulus – The user performs an action, but something goes wrong and there is never a response. For example, you send a letter to someone, but never get a reply. For short-term interactions this is immediately obvious – if you are waiting for a response and nothing happens then you know that something is wrong. However, for long-term interactions you cannot afford to do nothing for several days waiting for a reply to a letter! Therefore you need a reminder that someone else needs to do something – a to-be-done-to list!

All the above problems have a negative effect on long-term interaction and they can cause potential failures in the work process. A previous work (Dix, 1992a) focussed on the problem of missing stimuli and proposed some potential design solutions. Although it was clear from that work that some of the problems could theoretically occur, it was difficult to assess how prevalent they were without any empirical evidence.

A case study was therefore carried out to validate the analysis in a real situation. The problems due to missing stimuli and long-term interaction are closely linked to issues of interruptions on the work process as both cases cause disruptions in the flow of activities within a task. Hence, the techniques used in the case study were designed to expose these problems as well.

2 Analytic method

A long-term cooperative process is first divided into activities performed by either individuals or groups and the interdependencies between these activities are recorded. The activities and their interdependencies are then catalogued in a traditional workflow fashion but this only acts as the superstructure of the analysis. The focus is on *when* activities are performed and *whether* they happen at all. The main distinguishing aspect of this work is the emphasis placed on triggers that initiate activities.

Triggers

A trigger is essentially an event, which makes the activity happen when it does. The method used for analysing triggers is influenced by the status and event phenomena (Section 2.4.1). Events aim at informing, but more often initiating actions, which in turn may generate further events. The actions of agents may change the status of the agent or the world, but changes in status are themselves events that may trigger further actions.

Triggers ensure the transition between activities. The dependencies between activities imply that one activity is a pre-condition for another. This is precisely the sort of dependency that is captured in a workflow or process model (Warboys, 1994).

However, there will typically be a gap between the completion of one activity and the start of the next – an event is therefore required to trigger each activity. Depending on the nature of the trigger, one can determine the possibility or likelihood that an activity will be missed or if the activity fails to occur, whether those failures will be noticed. For instance, if someone has to remember to perform an activity, this involves short-term memory and it can be regarded as a fragile part of the process, especially if it is performed in a complex and busy environment.

The crucial aspect of this analysis is not to capture the events that enable an activity to carry on – these are the preconditions. Instead, the focus is on the trigger – the event that made the activity happen when it did.

Processes and activities

Processes are recorded as a series of circles or bubbles, each one representing an activity. The bubble is labelled with the agent(s) who perform(s) the activity and the nature of the activity. Lines between the bubbles record dependencies and arrows at the beginning of each bubble record the trigger for the activity (figure 3).

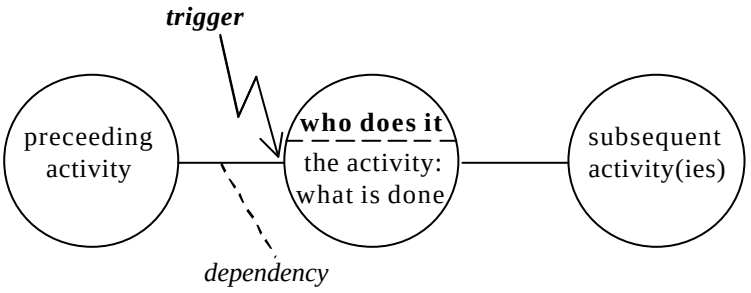


Figure 3. Recording processes

The case study adopted a minimalist approach for recording processes. Instead of recording all the complexities of real processes in a single diagram, a number of separate diagrams are used, often concentrating on a specific scenario. The crucial point is that for each activity we look for the corresponding trigger.

In general, activity boundaries are placed wherever there is the likelihood of a delay or gap. The most obvious situation that shows such a break is when subsequent activities in a process are performed by people at different sites. However, there are often distinct activities performed sequentially by an individual. In principle, such an exercise could go down to the full detail found in Hierarchical Task Analysis (HTA) (Shepherd, 1995). This would be reasonable if, say, interruptions were possible in the middle of typing a letter. But for the purpose of this analysis, such fine-grained tasks are ignored in order to retain a tight focus on long-term interaction.

The term activity rather than action is deliberately used to indicate that the lowest level of this analysis is far from atomic. Activities may be shared between individuals, for example having a meeting or dictating a letter would still be regarded as a single activity involving several people. Again, one could dissect such an interaction further, but this would be the remit of conversational analysis. Furthermore, if an activity is of no interest or if there is not enough knowledge about it, its details are ignored. For instance, if a firm issues an order to an external organisation and then waits for the goods to arrive, the internal processes of the external organisation may not be of any interest to us.

Finally, certain activities that are normally omitted in a traditional process model are included here. In particular, the receipt of a message is treated as a distinct activity to emphasise the gap that may occur between receipt and response.

3 Details of the study

The case study was based on a thorough investigation of the flow of work during the administration and organisation of a past HCI conference. A number of activities had to be carried out prior to the actual conference and most of them required the coordination of information among several people at various sites. Ann, the conference organiser acted as the central coordinator in many of these activities. She was the first point of contact in any enquiry, but this was only part of her work for the duration of the conference. Although the study covered an extensive range of activities that Ann had to coordinate, the activities relating to the flow of work during the life cycle of a paper were examined in detail (for a longer report see (Dix et al., 1995)).

Most of the processes encountered in this case study were in lock-step and only made up a small part of Ann's overall work. Data collection methods generally used for traditional task analysis or requirement elicitation such as direct observation and documentation were impractical for this study due to the long-term, ecologically rich and cross-organisational nature of the processes. Although documentation of long-term processes is likely to be relatively accurate, it may omit the activities beyond organisational boundaries, and above all, most of the triggers. However documentation can be used as an initial framework and later supplemented by observation or subsequent interviews.

Given that the processes of interest were geographically dispersed, direct observation was inappropriate. The necessary protracted field studies would not be acceptable as a part of normal commercial design practice. However, the lock-step nature of a conference is not

typical of office processes. In many office situations, there are several instances of the same process at different stages of completion, for instance, in an insurance office many claims are processed, each at a different stage. In these cases, a day-in-the-life observation may be sufficient as long as each activity seen during the study period can be pieced together afterwards, even if the process in question is never seen to run from end to end.

The most effective way to gather information for the purpose of this case study was through in-depth interviews. Interviewing is often regarded as problematic since the accounts people give of their actions are frequently at odds with what they actually do. However, the interviewing exercise adopted here was governed by the analytic focus – the structure imposed by the process flow and the specific interest in triggers. This allowed omissions and inconsistencies to be traced back to produce reliable results from the interviews. Studies of this nature should normally be sustained by some additional direct observation but it is important that practical design should rely principally on more directed and less intrusive techniques.

Finally, the importance of environmental cues gives a vital source of information – the work environment itself. Typical items found in an office include papers, files on the desk, post-it notes, contents of an in-tray, annotated wall calendar. So by just looking at those items one can ask several questions. For instance, why is that file sitting on the desk? What will happen to it? What would happen if it were not there? Each item in the environment fulfils a specific role and by its very presence one can determine the activity it triggers. At the very least, a piece of paper left on the desk is saying, "file me please".

Example 1 – paper submissions

The rest of this section will now consider some of the processes during the life cycle of a paper and they are typical of any scientific conference.

The paper submission sub-process starts when the author sends the paper to the conference office (figure 4). Ann receives the paper through the post, records its details in a database and then files a copy of the paper, ready for subsequent review.

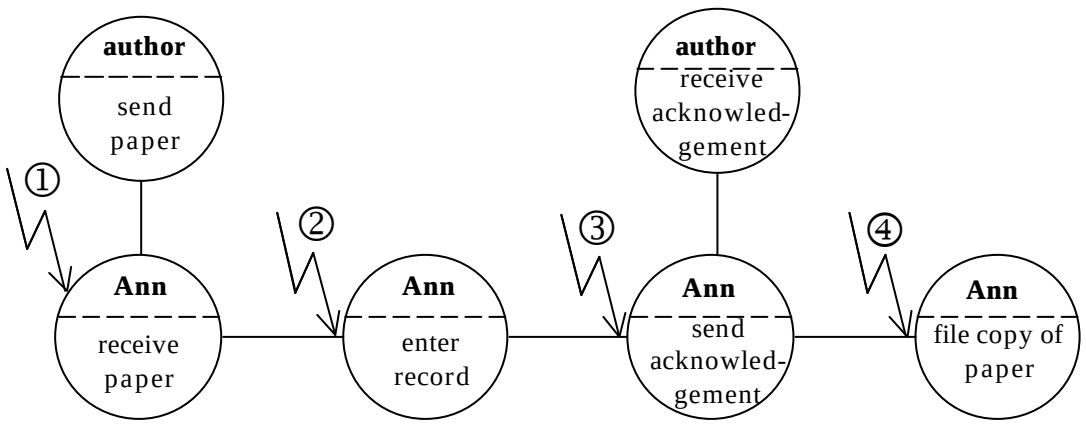


Figure 4. Paper submissions

Each activity above is triggered by a certain event.

Trigger ① is simply when the packet containing the paper reaches Ann via a communication channel, in this case it was by post. The postal system could be investigated in detail but since it is an external organisation, it is ignored. However, the possibility of a failure due to the unreliability and timeliness of the medium of interaction is recorded, as this will affect the whole system's operation. A possible solution to guard against such a failure is to augment the existing communication channel with a more reliable protocol. For instance, electronic mail could be used in parallel with postal mail, but this might result in a situation where humans, unlike software, may find the additional protocol too time consuming to maintain. Moreover, the reliability of electronic mail could be questioned as well.

Ann did not immediately enter the paper's details in the database. Instead, she waited until a small pile had accumulated before entering all the details together. Trigger ② is therefore the pile of papers on the desk. This trigger is an environmental cue that allows Ann to pick up the threads of her activities. Environmental cues are important triggers that serve as reminders. As soon as the paper details were recorded, Ann sent an acknowledgement to the authors and filed a copy of the papers.

Both triggers ③ and ④ are such that in an interruption-free environment, the end of the previous activity acts a trigger for the next activity. However, Ann may be interrupted for some length of time while she is in the midst of sending an acknowledgement and filing a copy of the paper. In case of an interruption, the secondary or fall-back trigger is examined. The fall-back triggers for ③ and ④ are the same as ②, in other words, the unfiled papers on the desk.

Because the activities have the same trigger, an activity will potentially either be repeated after an interruption or omitted entirely (if Ann mistakenly thought an interruption had previously occurred). Clearly, it is a mental strain to keep track of all the tasks one is engaged in. If someone fails to complete or close tasks held in short-term memory, or is prevented from doing so by interference, the subject is liable to lose track of what she is doing and can consequently make errors. Luckily, Ann's memory was good enough and these problems were not encountered in this case. However, interruptions can have major consequences on the flow of work within a collaborative system (Rouncefield et al., 1994) and its likelihood should not be discarded.

Example 2 – referee allocation

The next stage after the paper submission process was the referee allocation exercise (figure 5). The process starts with Ann sending the authors papers to the conference committee. The committee holds a meeting to decide which paper is assigned to which referee. The papers are afterwards sent back to Ann together with the decisions reached. Ann then updates the database and dispatches the paper copies to the relevant referees.

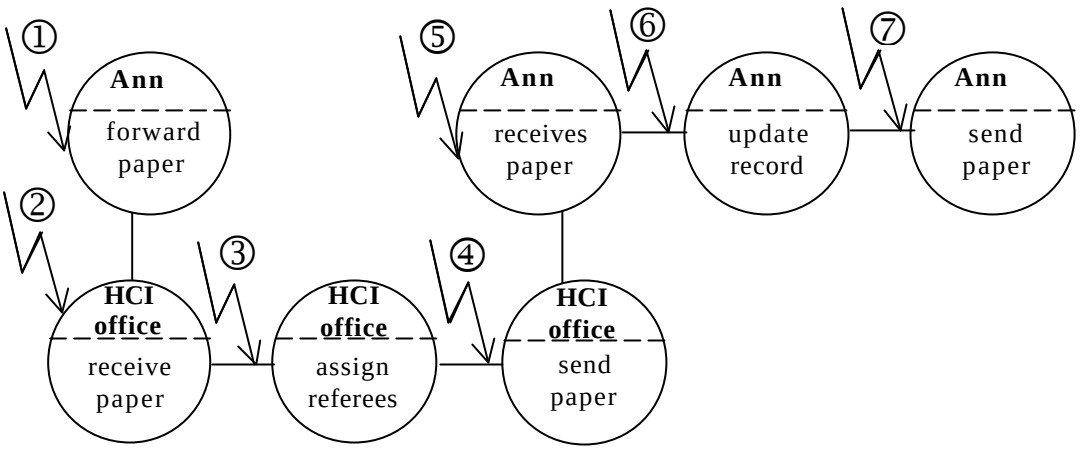


Figure 5. Referee allocation

This is an interesting scenario as it highlights some new types of triggers. Trigger ① is the deadline that prompted Ann to forward the papers to the conference committee. As there was only one deadline for all the papers, it was not too difficult to remember the date.

Triggers ② and ⑤ represent the communication channel through which Ann send the papers to the committee and receives them back. In this case it was via internal mail, which was a relatively reliable medium.

Trigger ③ is an external event – here a meeting session, which allowed decisions to be made regarding referee allocation to papers.

Trigger ④ directly follows from the previous activity, so as soon as the referees were nominated, the conference committee sent the papers back to Ann.

Trigger ⑥, the papers on the desk, reminded Ann to update the database and allowed pick up the thread of her activities.

Trigger ⑦ directly follows from the previous activity and when faced with such a trigger, interruption can disrupt the flow of the activities as highlighted in Example 1. For instance, if Ann was interrupted in between updating the records and sending the papers and if the paper was left lying on the desk she could enter the record twice; or else if she did not see the paper, she could assume that the paper was already sent and omit sending it altogether, although the paper could have been mislaid.

Example 3 – refereeing process

As a final example, let us consider the refereeing process, which involves referees annotating the papers they have received and sending the reviewed papers back to Ann (figure 6).

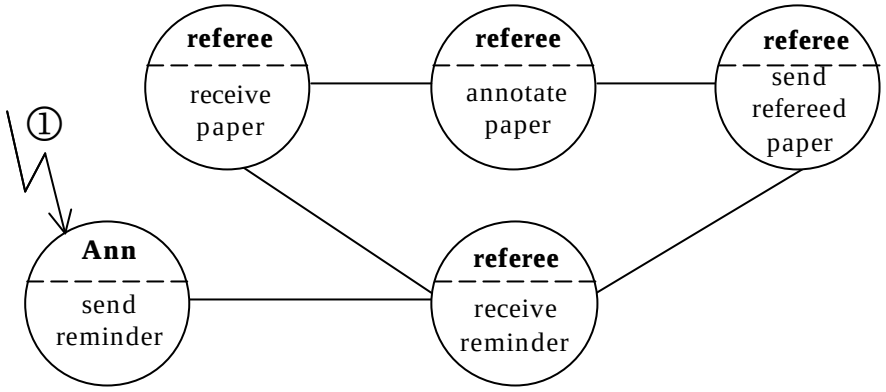


Figure 6. Part of refereeing process

In this case, the agents involved no longer reside within a single organisation. Organisational boundaries have been crossed and the success of the whole process is entirely dependent on the referees based at different locations. So how does Ann coordinate the referees' activities when there is a temporal gap between the dispatch of the papers and the return to the referees' reviews?

Trigger ①, the deadline, enables Ann to regain control. If Ann does not receive the refereed papers by the date set for return, she sends reminders to the referees. As there was only one deadline in this case, the date was easy to remember. However, if each paper were allowed a different date for submission, Ann would have to keep track of deadline dates periodically. When faced with a periodic action, one can ask how does the person remember to perform the action at the appropriate time?

This scenario therefore shows that in a long-term collaborative situation, especially when the control resides among different agents and when there is a gap between an event and its subsequent action, it is vital to prevent activities getting out of synchronisation otherwise a range of failures can occur.

4 Findings of the study

Although the initial aim of this empirical study was to verify the analysis of pace (Dix, 1994a), the techniques employed generated a range of interesting issues. Firstly, the modelling of activities during the work process became clearer. Secondly, based on previous theoretical analysis and refined by the results of the study, different classes of triggers emerged. Finally, a pattern of activities was discovered which might be regarded as a fundamental unit of long-term work.

Types of triggers

From the examples discussed in Section 3, trigger types recurred in various scenarios and some general classes of triggers emerged as follows.

(a) *Completion of previous activity* – This is when one activity begins immediately after the previous activity has reached completion. But we may treat this with suspicion – does the second activity always proceeds immediately? If there is any chance of a gap or interruption, we must look for secondary triggers.

(b) *Memory (sporadic actions)* – Very often, people have to remember that they need to do something. For instance, when a request is made verbally, the recipient has to remember that the request is outstanding until either it can be performed or some record of the commitment is made. In the latter case, the recording of the commitment is itself an important activity.

(c) *Periodic actions* – These are actions that occur at regular intervals, for example, reading email every morning. But when faced with a periodic action, how does one remember to perform a certain task at the relevant period? It may be due to a routine behaviour one has acquired, such as consulting a diary every morning. However, if it is an hourly activity then one may ask how does the person know when it is the hour? Perhaps the clock strikes or the watch beeps on the hour, but this is an external signal trigger (see below).

(d) *Temporal gaps* – Unlike periodic activities, temporal gaps are characterised by a single significant moment or delay. For instance, we may need to perform a generic task by a deadline or expect a response by a certain date. Again we must ask what makes a person notice the actual event has occurred.

(e) *External events* – Very often, periodic actions and temporal events are signalled by a wristwatch or an automatic calendar set to pop up a reminder at a specific time. Also, non-time based events may occur to prompt actions, for instance the completion of an automatic activity, an event in the world or even the (electronic) receipt of a message.

(f) *Receipt of a message* – This is a special kind of external event, which includes a telephone call, a face-to-face request or the receipt of a letter or a fax. Such events can

only be considered to be the trigger for an action if that action occurs immediately after the request is received. If the request is dealt with later (as is more often the case) the receipt of the request and the response to the request are treated as separate activities. Furthermore, we have to record the reliability of the communication media, the possibility of communication delays and the consequences of any failure in the channels.

(g) *Environmental cues* – These are things in our environment that remind us that activities ought to be done. Sometimes this may be explicit, like a diary entry or sometimes implicit, such as a partly written letter in the typewriter. Environmental cues may manifest themselves in paper form, for instance, to-do-lists, diaries, or in electronic form, such as an email waiting in the in-box.

Triggers of type (a) and (b) are insecure as they are liable to interruptions and poor memory respectively. In each case, we look for a secondary or back-up trigger or where this is absent, we look at the process as a whole and assess the consequences should the activity fail to trigger at all.

Other triggers also lead to follow-on questions. For instance, if a temporal event (d) is triggered because it is in a diary, what makes one look in the diary? It may possibly be due to a periodic activity (c) in which case, how does one know when the period occurs? Environmental cues (g) are fundamental but even here one must ask what makes a subject notice a particular cue?

We can carry on asking such follow-up questions indefinitely, but at some point we must stop and either assume that a trigger does always occur as specified, or if not, assess the reliability of the trigger and perhaps any delays associated with noticing it.

The 4Rs

Although the initial focus of this case study was on individual triggers, a pattern in the processes emerged as they were recorded. This pattern was called the 4Rs: Request, Receipt, Response, Release. Figure 7 shows a simplified version of figure 4, which exemplifies the 4Rs.

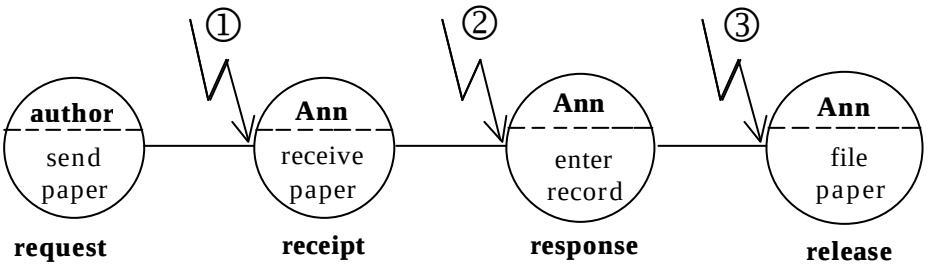


Figure 7. The 4Rs

The pattern of activities shows a general structure: *request* – someone sends a message (or implicitly passes an object) requiring a certain action; *receipt* – the receiver receives the

message through a communication channel; *response* – the receiver performs the necessary action; and *release* – the receiver files or disposes of the things used during the process. At this point, if the functional goal has been achieved, the process can be considered to have reached completion.

Not only is the pattern of activities common between different processes, but a similar pattern can also be seen in the types of triggers. Trigger ① is always some form of communication mode (trigger type (f)), and can be assessed for reliability and timeliness. Trigger ② is most likely to be the presence of a document or another object (trigger type (g)), which activates the *response* activity. Trigger ③ is the completion of the previous activity (trigger type (a)), and it usually removes the presence of the environmental cue but also relies on its existence as a secondary trigger.

The above pattern has several refinements, for example when a note is made of a verbal request, this adds an extra stage to the *receipt* activity. Another interesting variation is linked to the *response* activity, which may involve more than one action. For instance, in figure 4 the response consisted of two activities 'enter record' and 'send acknowledgement'. When faced with such a situation, we need to look very carefully at the triggers for the two parts of the response, as they may in fact be the same trigger. This was the case in figure 4 as both *response* activities were triggered by the presence of the pile of papers on the desk. On the other hand, in some situations, for example receiving information for filing, there may be no separate response as the *response* and *release* activities are merged.

Figure 8 also demonstrates a frequent aspect of the 4Rs – the *response* of one 4Rs pattern forms the *request* activity initiating a new 4Rs pattern. For example, the *response* activity 'send acknowledgement' in figure 4 is itself a message to the author and may generate another 4R sub-process. A chain of such iterative 4Rs patterns can create a format for long-term conversation. However, this generic pattern is at a much lower level than those identified in speech-act theory (Winograd and Flores, 1986) and may thus be considered to be the long-term interaction equivalent of adjacency pairs found in conversational analysis.

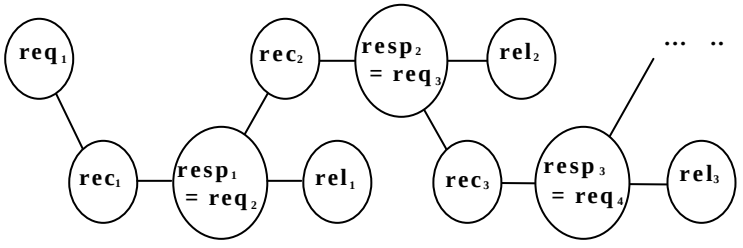


Figure 8. The 4Rs chain

5 Related approaches

The nature of the study discussed bears some similarity with other approaches in the general field of the 'social analysis of work', particularly workflow, speech-act theory, ethnography,

ethnomethodology and formal techniques including Petri Nets. The following points, however, summarise the critical differences between the approach adopted here and the above-mentioned disciplines.

Workflow

Workflow[18] implies technological solutions to improve the nature of work and usually hints at cultural changes. However, the case study was neither targeted at introducing any technological solutions nor at dictating any cultural changes. Most of the processes considered crossed organisational boundaries and consequently they were unpredictable. Unless some formal collaboration was established beforehand, workflow would have been unsuccessful at ensuring that the links of communication and activity do not break down in such an open environment.

Although the initial aim of the study was not to automate the processes of work or even to facilitate them through computerisation, as is the case with workflow, some design implications were reached (see Section 6). So, to avoid confusion or disagreement over the use of the term 'workflow', the approach adopted here is referred to as an investigation of the 'flow of work', with its principal focus being on events triggering activities.

Speech-act theory

The basic structure of speech-act theory (SAT) consists of all possible stages in conversational interaction (Winograd and Flores, 1986) but the approach used in the case study is more abstract. For example, the arrival of an email message may be a potential trigger whereas SAT would analyse the contents of the email itself. In contrast, the 4Rs pattern is at a lower level of granularity than SAT patterns such as conversation for action (CfA) – each action pair in a SAT diagram expands to a complete 4R.

Ethnography

Ethnography is committed to inquiring into patterns of interaction and collaboration, based on the assumption that human activities are socially organised (Hammersley and Atkinson, 1995). The method used here is also enquiring about a particular pattern but with a difference. Ethnography has an open-ended approach of gathering information and is based on the belief that one cannot know in advance of inquiry which elements of organisational life will prove to be of interest, value and importance for work (Randall, 1995). In contrast, the case study started with a precise focus on triggers that initiate activities. This implies that certain aspects that an ethnographer would normally record are ignored. However, a more restricted approach is better suited to inform systems design unlike ethnographers' open-

[18] One of the main centres within the workflow community – the Workflow Management Coalition, 1994 – has defined all the terms relating to workflow in organisations. Details available at <http://www.aiai.ed.ac.uk/WfMC>

endedness, which is seen as a weakness when it is used for requirements capture (Anderson, 1994).

Ethnomethodology

Ethnomethodology has also been used within HCI (Suchman, 1987) as a particular form of sociological analysis (Garfinkel, 1967). It involves observing, collecting and analysing data and deciding what is relevant about work activity as it really is, rather than an idealised conception of work as can be the case with process-modelling and workflow. Ethnomethodology differs from other modes of sociology in that it seeks to describe *from within* the ways in which people actually order their work activities through mutual attentiveness to what has to be done (Anderson, 1994). This case study also aims to describe peoples' work activities, but again, its a priori focus on specific aspects of work makes it distinct. Armed with the knowledge of what work had to be done, the aim here was to discover 'breakdowns' which could affect the completion of that work process.

Several studies (Bentley et al., 1992a), (Bentley et al., 1992b), (Heath et al., 1993), (Heath and Luff, 1994) have considered the importance of the environment for how work is executed. Traditionally, such studies emphasise the social actors and the close teamwork within that environment. However, more recent studies of office work (Herskind, 1997), (Rouncefield et al., 1994) have brought the surroundings in which people work and the artefacts into the limelight, in particular recognising the importance of paper (Sellen and Harper, 1997). This trend is also followed in this empirical work but with a more specific formulation of the purpose of artefacts as triggers for activity.

Formal techniques

Formal techniques have been applied to the study of time and collaboration including Petri Nets (Johnson et al., 1995), (Palanque and Bastide, 1995), various forms of temporal and modal logic (Dix, 1995b), (Johnson, 1997), (Reeves, 1996) and process algebras such as LOTOS (Paternó and Faconti, 1992). Any of these methods could be used to capture precedence relationship between the activities, but not the nature of triggers. However, a study (Joosten, 1994) applied Petri Nets to model workflow and used the word trigger in the same context, that is an event which initiates an activity. But this is where the similarity ended as the ecology of triggers was not investigated at the level of detail found in this study.

6 Design implications

The analysis employed in this empirical study was initially targeted at increasing the understanding of long-term interaction, but in use, it has some direct design implications. It can be used to determine whether a process is robust to interruptions and forgetfulness, and if not, identify where potential problems may occur.

Robustness of work process

The reliability of the work process can be assessed by asking specific questions about the triggers for activities. However, it is inevitable that triggers will fail for some reason, activities may be missed or perhaps the whole process may fail to continue because something goes wrong somewhere. The combination of a process model together with a well-founded assessment of the reliability of each activity can allow us to assess the robustness of the whole process.

If someone fails to complete an activity and consequently, the next activity is never triggered, what happens? Does the whole process seize up, or will the failure be eventually noticed. The approach adopted in this case study is not simply an ad hoc procedure, on the contrary, one can systematically go to each trigger and ask: what happens to the entire process if the trigger fails?

Furthermore, by looking at the process as a whole we can improve our assessment of the reliability of any trigger. For instance, if the trigger for an activity is a report lying in someone's in-tray, we can examine the wider context and assess the likelihood of whether the report will indeed be there when required.

Importance of environmental cues

As expected, the case study confirmed the importance of environmental cues as one of the principal and most robust triggering mechanisms. As mentioned above (Section 5) various studies have recognised the importance of the ecology of the workplace, including whiteboards, calendars, individual papers and piles on desks (Herskind, 1997), (Rouncefield et al., 1994), (Sellen and Harper, 1997). Indeed, in many cooperative processes there may be little direct communication, instead activities are coordinated by implicit communication through the artefact (Dix, 1994b).

This case study focussed on a particular role of these environmental cues, namely their ability to remind and trigger future actions. An understanding of the importance of papers is essential if there are plans to automate parts of an office procedure. One can assess whether automation will break the existing work patterns and if so whether alternative cues could be implemented in the new system.

Whereas many studies have concluded that papers are important, the analysis discussed here takes this a step further by developing an understanding of *why* paper is important.

Applying triggers and the 4Rs

The 4Rs framework was applied in the MaPPiT Project[19] to analyse the existing work processes and inform design decisions in developing a Lotus Notes[20] implementation of the student placement activity. The example scenarios are discussed in detail in (Dix et al., 1998). This evaluation process was particularly interesting in that it raised some new issues and confirmed some existing beliefs and these are summarised below.

Several variations of the 4Rs process were encountered; some with multi-stage responses. The possibilities of breakdowns in the work process as a result of interruptions, forgetfulness and long delays were clearly visible when the analysis was applied to the different scenarios. Different levels of automation were suggested by the 4Rs analysis. At one level this involved the complete bypassing of the human process, but in others only part of the procedures required automation. More importantly, the 4Rs analysis has ensured that the automated solution does not hide existing triggers, as is often the case with electronic filing, but is instead explicitly designed to enhance the triggers through the use of automatic reminders and electronic environmental cues.

Some of the solutions adopted included building navigators or agents into the automated system to act as automatic reminders for triggering activities. Also, automatic submission of data through Web browser forms was opted for to avoid losing data by handling it physically. Another interesting implementation was the introduction of new environmental cues by creating Notes forms to record the receipts of data and monitor the flow of processes.

The 4Rs framework was remarkably successful in describing patterns of activity and prompting appropriate questions to drive the Notes implementation. However the study also highlighted the fact that the salience of certain kinds of triggers could change over time. As a result, environmental cues could fail for exactly the same reasons that our memory finds to-be-done-to items difficult. We cannot therefore assume that the detailed triggers are homogeneous over time. Instead, we should establish by enquiry or observation whether triggers vary in kind or salience.

From the findings of the case study and its validation through the MaPPiT Project, it is clear that the broad techniques would be applicable to any process-oriented task analysis, such as Hierarchical Task Analysis (Shepherd, 1995). Consequently, a trigger analysis technique has been proposed (Dix et al., 2003) as a means for task decomposition, which can be applied in combination with many task analysis and workflow methods. The novelty of this approach lies in uncovering triggers that cause each subtask to occur at each step along the task decomposition process.

---

[19] MaPPiT – Mapping the Placement Process with Information Technology, a HEFCE project. Details available at: http://www.hud.ac.uk/scom/mappit/home2.htm

[20] http://www.lotusnotes.com/home.nsf

7 Summary

This case study has focussed on the problems that arise during long-term interaction. In particular, people may have difficulty in recalling the context of a delayed response (action–effect gap), in resuming activities after an interruption (stimulus–response gap) and in remembering the non-occurrence of anticipated events (missing stimulus). The problem of stimulus-response gap calls for to-do-lists and aide mémoires while missing stimulus requires to-be-done-to lists or similar reminders.

The above considerations led to an in-depth analysis of the importance of triggers in initiating activities. Very often, it is assumed that an activity is triggered by the completion of the previous activity. But this is unlikely to be the case in an office-based situation due to the competing demands and frequent interruptions. Triggers not only determine when a particular activity occurs, but more importantly they also show whether that activity happens at all.

The investigation of the flow of work during the HCI conference was used as a case study to validate the theoretical analysis of temporal problems linked with prolonged interaction such as interruptions and delays. It also provided a deeper understanding of the issues and problems surrounding long-term interaction. The findings were later validated by a second case study, the MaPPiT project. Both studies have highlighted the collaborative aspects of organisational modelling and the importance of reminders as an enabling mechanism for resuming activities following delays and interruptions.

During the analysis of the first case study, a recurrent pattern of activities emerged and it was named the 4Rs – Request, Receipt, Response, Release. This pattern is believed to be a fundamental unit of long-term work. The existence of generic patterns makes it easier to uncover problems situations quickly and to adapt solutions found in one situation to another. Both studies have shown that the same sequence repeats itself with similar triggers and similar failure modes. Any deviation in the 4Rs pattern indicated possible breakdown points.

Problems are most likely to occur when implementing changes in the work process by automating existing paper-based systems. This often leads to the loss of important environmental cues. The 4Rs analysis allows one to ask pertinent questions about the triggers for activities and also assess the reliability of individual parts of a work process. Furthermore, as confirmed by the second case study, the use of the 4Rs enabled the design of semi-automated processes where physical environmental cues were replaced and in some cases, enhanced with electronic cues.

The 4Rs framework can be applied to any process-oriented task analysis as it has some very powerful design implications. Although the nature of the study applied here may at first show some similarity to other approaches in the general field of the ‘social analysis of work’, there are however some crucial factors that distinguish this work. Its strength lies in uncovering triggers that cause each process to occur. Triggers enable us to determine

whether a process is robust to interruptions or forgetfulness and if not, identify the cause of the failure and the instance where any problem is likely to arise.