THE UNIVERSITY OF CALGARY

Flexible Data Sharing in a Groupware Toolkit

by

Theodore O'Grady

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

NOVEMBER, 1996

0-612-20846-X

Abstract

Synchronous groupware applications let users collaborate over distance through their computers. Since these applications are difficult to build, groupware toolkits have been constructed to help developers with their tasks. Some of the building blocks supplied by these toolkits are a set of abstractions for sharing data between sites. Yet different toolkits use different strategies to share the data: some replicate the data at all sites, while others store it at a central site. The correct choice of data sharing strategy is not obvious, as different strategies affect data consistency as well as the performance of applications built using the toolkit. We argue that data sharing should be flexible and that the developer should be able control the data sharing by selecting from default strategies or creating new ones to meet the requirements of their application. We use a technique called open implementations to provide this control. We have built a prototype groupware toolkit called GEN that demonstrates the feasibility of flexible data sharing. Using GEN, six different forms of data sharing have been constructed, more than any other toolkit currently available.

Acknowledgements

Very little creative work is done in a vacuum and there have been many people that have made this thesis possible. First and foremost I must thank Amy, my wife, who has been supportive and caring through all of it and who is my closest friend. Saul Greenberg's supervision has fostered an environment where it was possible to explore and organize my ideas, and has taught me a lot about writing.

Earle Lowe(e) and Mark Roseman have both been instrumental in this thesis, not only are they close friends but also co-workers with whom ideas can be discussed and examined. Sean Brandenberg, man of many bad sayings, is owed a special thanks for helping me survive Category Theory. Other lab-mates including Carl & Gwen, Linda, Doug & Ronnie, and now Don have made the lab an interesting place to work in, adding both enthusiasm and stimulating conversation to graduate life.

Calgary would not have been the same without Carol, Ian & Amy, and Kirt. They made the whole trip worthwhile! A lot of fun has been had with them, and a there's a lot more to come. Kim & Rob, and Leslie back in Ontario are close friends and were supportive throughout. Gus, who is a life-long friend, many years ago explained to me that science was fun, I got hooked and ended up writing this thesis.

Paul Dourish provided stimulating conversation and challenging ideas. Mark Roseman, Paul and myself expect to market the GOMSercisor any day now. Object Technology International not only funded this thesis, but also supported the research with enthusiasm. Great thanks in particular go out Dave Thomas, Brian Barry and the VM Team.

Finally, I've got to say thanks to my family Paul & Francis (that would be Dad & Mom), Chris & Stuart O'Grady, and the newer bits (in-laws) Mary & Alan, Peggy & David, Norman & Anne, Ben, Katie, & Nick. Thanks people!

Table of Contents

List of Figures

List of Tables

1. Introduction

Real time groupware applications give users the ability to collaborate over distance through their computers. While these systems are now commercially available, they are notoriously difficult to build. Developers must not only deal with defining the semantics of the application, but also deal with the technical issues of how to distribute data and processes around the network. Consequently, toolkits have emerged that allow developers to construct groupware applications more easily. These toolkits provide common building blocks such as inter-process communication, distribution of events and data, mechanisms that allow participants to enter and leave conferences, and specialized user interface widgets. A runtime architecture supports these building blocks by managing process creation and destruction, communication connections, and fault tolerance.

Runtime architectures are an integral part of groupware development, and consequently most toolkits are classified based on how their architecture is configured. The runtime architecture determines the way the system distributes the processes and data of an application across machines, and the way messages are routed between them. Runtime architectures lie between two extremes, from completely centralized to completely replicated. In a *centralized architecture*, there is a single primary process and a single copy of the data residing on a single machine, and all communication is routed to this process. In a *replicated architecture,* each machine involved in the conference has its own local copy of the process and data, and communications between the processes keep replicas synchronized with one another. *Hybrid architectures* are also possible, containing, both replicated and centralized components. There is no clear choice of architectural style, as the type of architecture chosen results in trade-offs between performance and ease of implementation (Greenberg & Roseman, 1996).

We claim that developers need to control data sharing in groupware toolkits in two ways: by selecting from existing methods of sharing data on a per object basis; and by constructing new data sharing mechanisms that specify how data is distributed and what concurrency control mechanisms are used. Control over data distribution lets the developer specify where the data is located (e.g. whether it is centralized or replicated). Management of concurrency control mechanisms lets application developers determine how data is kept consistent (e.g. through locking mechanisms).

It is our goal to provide flexible data sharing to groupware application developers. We achieve this goal using *open implementation* (Kiczales et al., 1995), a technique that allows us to provide developers with default data sharing strategies and the ability to create new ones. We have built a toolkit called GEN which demonstrates the feasibility of flexible data sharing. With GEN we show three default data sharing implementations and how three additional sharing strategies can be constructed.

Although we have targeted groupware toolkits for exploring our ideas about how developers should control the way data is shared, this work can also be seen as having a broader scope and applying to other areas of distributed computing. For example, the idea of giving developers control over the sharing strategy may be useful in applications such as distributed agents. This will discussed further in the final chapter when we consider future work.

To set the scene for the rest of the thesis, the next section identifies the components common to many runtime architectures in groupware toolkits. The subsequent section then narrows the focus by considering the predominant types of runtime architectures seen in current toolkits. Finally, I restate the purpose of this thesis and outline of the remainder of the thesis.

1.1 Runtime Architectures of Groupware Toolkits

Many developers of groupware toolkits have examined in detail the technical features required to build groupware toolkits. These toolkits include GroupKit (Roseman, 1993), ObjectWorld (Tou et al., 1994), Touring Machine (Arango et al., 1993), Rendezvous

(Patterson et al., 1990), MEAD (Bentley et al., 1994), MMConf (Crowley et al., 1990), and Clock (Graham & Urnes, 1996a,b). These toolkits typically provide specialized groupware widgets such as multi-user scroll bars, an application programming interface (API), and an underlying runtime architecture. The groupware widgets fall outside the scope of this thesis, and the reader is referred to Greenberg and Roseman (1997) for further information in this area. The application programming interface is closely tied to the runtime architecture, so this discussion in this sense subsumes them. This thesis concentrates on the structure of runtime architectures, which are typically classified into two separate components: session management and data sharing.

Session management is the set of building blocks that allow developers to create conferences and connect all of their participants (and their programs) together. The session management layer is responsible for starting up and tearing down processes on different machines, establishing and maintaining inter-process communication (IPC), handling fault tolerance, and providing a persistence mechanism for saving information (Greenberg and Roseman, 1997).

Data sharing mechanisms provide abstractions for sharing information between sites. The data sharing mechanism controls how the data is distributed (e.g. centralized or replicated), how it is kept consistent (e.g. by locking), how it is organized when there are large amounts of data, and how other objects (such as user interfaces) are notified of changes made in the data. In most groupware toolkits the runtime architecture intertwines the data sharing model and the process (although they need not be). This will be discussed further in Chapter 2.

1.2 Data Sharing Strategies for Runtime Architectures

As mentioned earlier, the data sharing strategies of groupware toolkits generally fall between two extremes: centralized or replicated (Greenberg & Roseman, 1997). In a centralized architecture, there is single server machine through which all input and output requests are funnelled (Figure 1-1a). Client processes are only responsible for sending requests to the central program, and for displaying the results. Because all data resides in the central server, the data sharing model is also centralized. Only one copy of the data exists, and all transaction requests must be funnelled to it. A centralized server simplifies implementation since synchronization is implicit in a single process model.

Replicated architectures have a copy of the program at every site (Figure 1-1c). Each program maintains a copy of the data. While this means that requests for data can be handled locally, it also implies that each site must coordinate with every other site to ensure consistency. Concurrency control algorithms must be used to maintain an adequate level of synchronization between the various copies of data.

A hybrid architecture (Figure 1-1b) has components of both a replicated and a centralized architecture. Here, a copy of the data can be kept in a central process as well as having copies in the replicas. Toolkits that use hybrid architectures typically place the data at a single site based upon performance criteria.

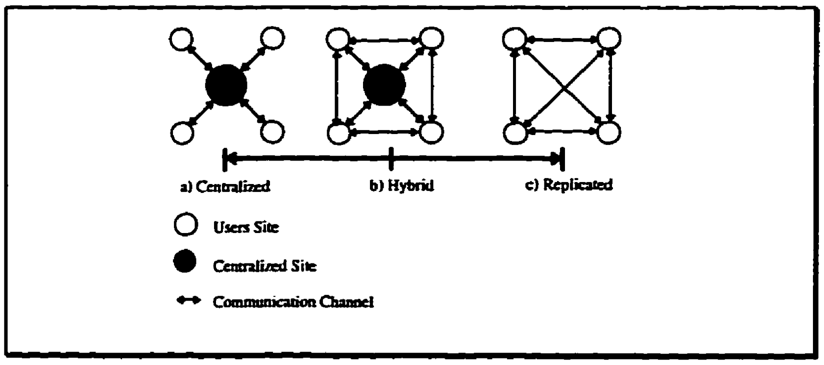

Figure 1-1. Conceptual structure of centralized, hybrid and replicated processes.

In most of today's groupware toolkits, the runtime architecture and its implementation is generally not modifiable. Toolkits typically supply only a single strategy for data sharing, either centralized, replicated or hybrid. Only a handful of new toolkits allow developers to choose from a limited set of data-sharing strategies and these choices are hard-wired (Graham & Urnes, 1996a).

Each of these architectures (replicated, hybrid and centralized) have benefits and drawbacks for the application developer (Crowley et al., 1990). These trade-offs include the ease of implementing consistency, persistence, latecomers, different application versions, connecting users, and heterogeneous environments. Tradeoffs also include runtime issues such as reliability, performance over slow networks, parallelism, and scalability. Chapter 2 will discuss these further.

In addition to the centralized to replicated architectures used by groupware toolkits to distribute data, there are also other possibilities, such as migration (Nascimento & Dollimore, 1992). Migration also has its own set of trade-offs, in terms of performance and ease of implementation. However, no toolkit currently allows data migration.

When using today's toolkits, application developers currently cannot control how their data is distributed around the network and kept consistent. They must design around the runtime architecture provided by the groupware toolkit they select. While the toolkit's runtime architecture may be appropriate for some components of their application, it may be a poor fit for others. All these points are indicators of the inflexibility in the current state of the art.

1.3 Purpose of this Thesis

The purpose of this thesis is to demonstrate that groupware toolkits can provide flexible data sharing. Unlike today's implementations, toolkits can give the application developer fine grained control over how data is shared. This includes the way data is distributed across the network, and how it is kept consistent for each piece of data. Such a toolkit should allow application developers to both select from a set of default strategies for sharing data (provided as options by the toolkit), or to define new ones.

1.4 Overview of this Thesis

The thesis will show how groupware toolkits can be made flexible through the several steps outlined in the chapters below.

Chapter 2 provides background. It considers data sharing techniques used in the runtime architectures in currently available toolkits. An examination of the trade-offs between the various sharing strategies reveals that an application may require multiple sharing strategies.

Chapter 3 describes the philosophy of our approach to making groupware toolkits flexible. It introduces a technique now being applied to toolkits (Kiczales, 1995) called open implementation and discusses how it can be applied to groupware toolkits to give the developers both the ability to choose from default data sharing implementations and to create new ones. The technique of open implementation suggests that a toolkit can be broken into two components: a programmer's interface which contains useful default implementations for sharing data; and a meta-interface which the programmer can use to create new techniques for sharing data.

Chapters 4 and 5 present a prototype called GEN that uses open implementation techniques to give developers fine grained control over the distribution and location of objects. A programmer's interface is presented that provides an API for session management and data sharing. A second level interface, called the meta-interface, gives application developers direct control over data distribution and concurrency control mechanisms. Although GEN is just a prototype, it illustrates the principles and details of how open implementation can be applied to groupware toolkits.

Chapters 6 and 7 demonstrate how GEN's meta-interface can be used to create six different strategies for sharing data between different sites. These implementations include replicated, centralized, replicated locking, selective message broadcasting, migration, and a form of optimistic locking. These case studies are used to demonstrate GEN's capabilities. While no claim is made that GEN can cover all situations, they illustrate that GEN is far more flexible than any other groupware toolkit currently available.

Chapter 8 concludes the thesis. It summarizes the arguments presented in the thesis, critiques the work, and suggests future research in the area of open implementation for groupware toolkits. Finally, it describes the contributions that this thesis has made to computer science.

2. Choices for Sharing Data: Runtime Architectures in Groupware Toolkits

Application developers choose between groupware toolkits for many reasons. Obviously, their choice is based partially upon the functionality the toolkit provides, such as the particular groupware widgets supplied, for these are the primary building blocks that the designer must use to create their application. A second and perhaps more subtle aspect of their choice is the underlying strategy the toolkit uses to manage shared data. The way data is shared can affect the type and performance of applications that are built using a particular toolkit. In this section, we explore some of the trade-offs between various data sharing strategies, and how they are influenced by the actual runtime architecture used by the toolkit. This elaborates some of the ideas already introduced in Chapter 1.

When researchers discuss runtime architectures of groupware, it is typically in terms of its process structure (Crowley, 1990; Greenberg & Roseman, 1997). For example, a *replicated architecture* is said to have a process at every site, with data being shared by maintaining a copy of it in each process. In contrast, a *centralized architecture* is said to have a single process on a single site, with the shared data residing within that single process. However, these terms are used fairly loosely in practice, and most groupware toolkits only tend toward one of these two extremes. For example, although toolkits such as Clock (Graham & Urnes, 1996), MEAD (Bentley et al., 1994) and Weasel (Graham & Urnes, 1992) claim to be forms of centralized architectures, they all have multiple processes running at each of the sites. What the creators of these toolkits usually mean is that it is the *shared data* that is centralized, although it is sometimes so tightly bound up to a centralized server process model that it is difficult to differentiate between the two. Consequently, in this chapter, we will emphasise the location of the data, rather than the processes, to differentiate between various groupware architectures.

The first section begins with a summary of replicated and centralized architectures, examining some of the arguments that have been made for and against each of them. The subsequent section continues by describing and contrasting how particular groupware toolkits implement their process/data-sharing architectures. The chapter concludes by discussing the requirements for a flexible toolkit that separates the process model from the data sharing model, and allows developers to create or choose between different strategies for sharing data.

2.1 Centralized and Replicated Architectures

One of the primary design decisions made by developers building a groupware toolkit is whether its architecture should be centralized or replicated (Crowley et al., 1990). They are usually concerned with the trade-offs between the two architectures, such as how easy they are to implement, and what they believe application developers will need. We begin with a look at how these architectures are generally implemented.

Centralized architectures have a main process with a single copy of the shared data residing on a central server machine. A centralized architecture is usually composed of one server process and multiple client processes (usually 1 per participant). The client process simply receives input from the user and forwards it to the centralized server. The centralized server is the application program, which acts on the input received from the clients and then updates each of the client displays.

For example, consider a simple brainstorming tool that presents participants with a visible list of ideas, and that allows any participant to enter a new idea to the list. In a centralized architecture, a single data structure containing the list of ideas is kept in the central process. Now consider the sequence of events in Figure 2-1, where four people are using this tool. The participant at Site 1 is about to enter a new idea, called 'C', to a list that already contains ideas 'A' and 'B'. The participant types some text and hits 'return', which initiates the sequence. The text, which is generated at the local site, is forwarded to the central server. The central server receives the event, adds 'C' to the shared list, and then updates each of the displays on the client machines. The work of adding the element to the shared list and initiating the display updates is done at server. The clients merely forward events to the server, and redraw the interface as directed by the server.

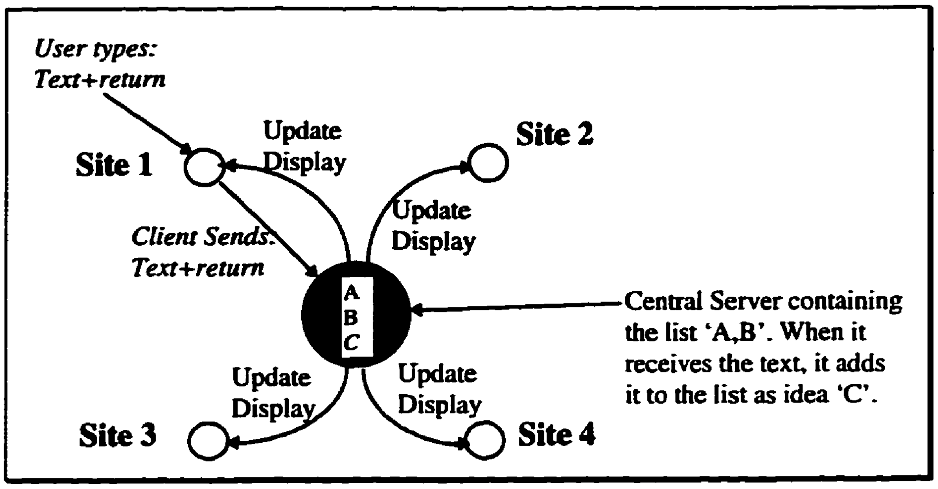

Figure 2-1. Sequence of events in a centralized architecture.

Replicated architectures have a process running at each site as well as a copy of the shared data at each site. Rather than forwarding the events, the local process handles the event locally, and then tells others about it by broadcasting a procedure call or message to all the other sites. The other sites receive the procedure call or message, and update their internal copies of the data as well as the display.

For example, reconsider the problem of adding 'C' to a shared list in a replicated architecture. Figure 2-2 illustrates how this architecture distributes the information. Site 1 will receive the text + return, add 'C' to the local copy of the list, and then update the local display. It then broadcasts a procedure call (including the text as an argument) to sites 2, 3, and 4. These remote sites will then execute the procedure that adds the idea 'C' to the local copy of the shared list. The remote sites then update their displays.

Which architecture is better? Researchers have argued the merits of these two architectural extremes, and adopt the one they feel is best for their toolkit. Bentley et al. (1994), Hill (1992), Patterson et al. (1996), Wilson (1995), and Ahuja et al. (1990) all argue the merits of different forms of centralized architectures for groupware. On the

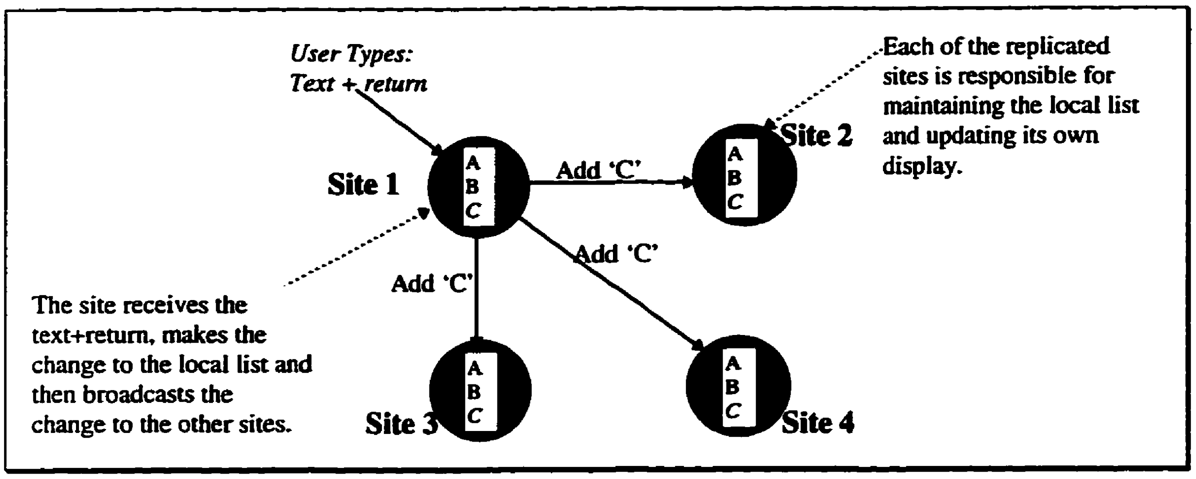
User Types:
Text + return
Site 1
A
B
C
Add 'C'
Site 2
Each of the replicated sites is responsible for maintaining the local list and updating its own display.
Add 'C'
Add 'C'
The site receives the text+return, makes the change to the local list and then broadcasts the change to the other sites.
Site 3
Site 4

Figure 2-2. Sequence of events in a replicated architecture.

other hand Craighill et al. (1993), Crowley et al. (1990), Bonfiglio et al. (1989) and Anupam & Bajaj (1993) argue the merits of various forms of replicated architectures.

A centralized architecture is easier to implement in that the developer does not have to worry about handling consistency, persistence, latecomers, or different versions of the applications. A replicated architecture on the other hand gives superior performance in the cases where there are slow networks, a high need for concurrent activity, a large number of users, users requiring different views, or when the network of computers is heterogeneous. In the following section we will discuss these differences, which are summarized in Table 2-1.

	Centralized	**Replicated**	**Best**[1]
Consistency	Inherent in the architecture.	Toolkits must implement concurrency scheme.	C
Persistence	Copy centralized data to repository to transmit state info.	Unclear which site should store the data.	C
Latecomers	Use same consistent data repository.	Toolkit dependent strategy must be created.	C
Application Versions	Only one version of the application running.	Must ensure every site has the same version.	C
Connecting Users	Every site connects to a single machine.	Each site must explicitly connect to all other sites.	C
Slow Networks	Slow local feedback.	Rapid feedback for local actions.	R
Distributed Execution	All operations on data are serialized.	Multiple sites can operate on data simultaneously.	R
Scalability	Single process can become a bottleneck.	Multiple processes can distribute the work.	R
Heterogeneous Environments	Hard to make work in heterogeneous environments.	Simpler to implement in heterogeneous environments.	R
Multiple Views	Main program must contain all variations.	Each site implements its own view.	R
Reliability	Only as reliable as the central server the application runs on.	Graceful reliability in principle as the loss of a single machine will not stop the entire system.	?

Table 2-1. Differences between replicated and centralized architectures.

[1] C = Centralized is superior, R = Replicated is superior, ? = Unclear which strategy is superior.

2.1.1 Advantages of a Centralized Architecture

Consistency is a concern in any distributed system, because parallel operations on data can leave it in different states at different sites. Centralized architectures eliminate this problem by keeping all the shared data at a single site. With a single copy of the data, there is no danger of state inconsistency because there is only a single process and only a single copy of the data.

In a replicated architecture, two users can change a piece of shared data at the same time (Greenberg & Marwood, 1994). If no consistency control mechanisms are used, these two copies of the data may become inconsistent. Figure 2-3 illustrates how state inconsistency may occur when different sites shade a circle with different colours at the same time. In Step 1, two copies of a single circle are simultaneously sent a message by their local sites to change its colour. In Step 2, we can see that the change is first applied locally and then the colour change is broadcast as a message to the other site. Finally, in Step 3 the circles have received the broadcast message and have applied it. However, the circles' colours are now inconsistent because the local and broadcast message were executed in different orders at each site.

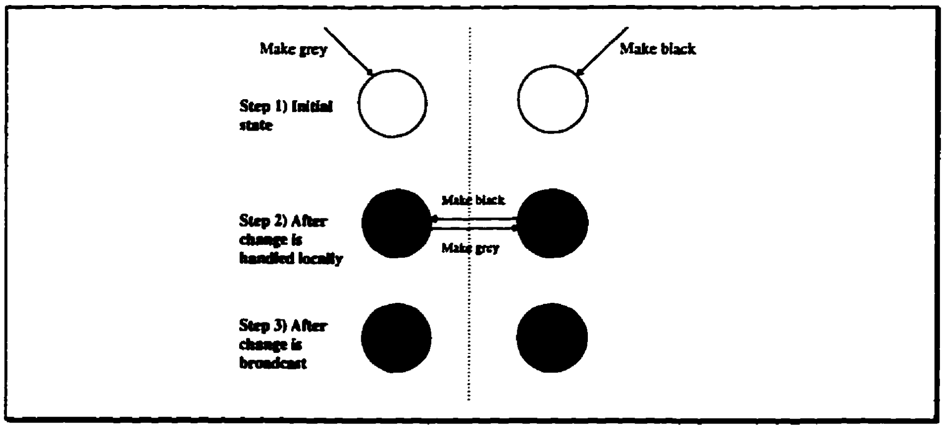
Make grey
Make black
Step 1) Initial state
Make black
Step 2) After change is handled locally
Make grey
Step 3) After change is broadcast

Figure 2-3. The result of two simultaneous changes in a replicated architecture.

In a replicated architecture, there are many different consistency schemes (such as locking, or optimistic locking) that can be applied to overcome this problem. The choice of these has different implications for the user in terms of speed and recovery behaviour

(Greenberg & Marwood, 1994) - the toolkit developer must decide which consistency mechanisms best meet the needs of the application. Because of their importance, we shall revisit the trade-offs between these schemes in detail in Section 2.1.3.

Persistence allows users to save the state of a conference between sessions. It is useful because users may temporarily suspend a conference that they wish to resume at a later time. Persistence is easily handled by a centralized implementation because the centralized server can save the state of the shared data at the server's site and restore itself to that state when the conference is resumed (Tou et al., 1994). The user re-enters the conference by connecting to the server, and the saved conference can be resumed.

In a replicated implementation, each of the sites is an equal peer. Determining where to save the data is more complicated because it is not guaranteed that any of the peers will be part of the conference when it is resumed. As such, the toolkit cannot decide where to save the data, and replicated architectures must come up with alternative strategies. In the case of some toolkits, that problem is left up to the application developer (Roseman, 1993).

Latecomers are defined as sites where users enter a conference after it has begun (Crowley et al, 1990). In this case, many of the shared data structures will have changed from the initial state. For example, an initially empty whiteboard may now contain a drawing. The latecomer will somehow have to update itself with the current state of that drawing. In a centralized architecture this is not a problem. The shared data is contained in the server, and the new site can request the current server to bring it up to date (Patterson et al., 1996).

In a replicated architecture, the new site must get a copy of the current state of the data from one of its peers. Some toolkits do provide hooks for the developer to write application specific code for updating latecomers (for example see Roseman & Greenberg, 1996; Tou et al., 1994).

Different application versions. Applications are frequently modified and there is a risk that participants may have different and incompatible versions of the software.

Centralized architectures do not have to worry about this because the server is the only site running the application. The clients themselves are usually simple display programs or virtual terminals, such as network window systems (e.g. XServer systems, Patterson et al., 1996).

In a replicated architecture a copy of the application program must be run on each site. Problems appear when the local site does not have the application, or when it has a version that is out of date. In this case the system must be able to check the version of the application (Crowley et al., 1990), perhaps uploading the newest one if possible.

Connecting users. Each user must have a way of connecting up to the conference from their local site. In the centralized architecture, the user only has to connect up to the single server site. If they get the site wrong, they will be excluded from the conference. However, their failure to connect will not interfere with the other participants.

In replicated architectures, the user must know the address of each of the other sites connecting to the conference. If the user gets the name of a site wrong, or excludes one of the sites, changes made by one site may not be broadcast to all the other sites.

2.1.2 Advantages of a Replicated Architecture

Slow networks are always possible in an environment where connections are maintained over long distances, over congested networks, or over networks with restricted bandwidth. A slow connection can affect the speed at which changes made by one site can be seen at another site. Remote actions may appear sluggish on the local site as changes are propagated across the network (Crowley et al., 1990). Sluggishness can be annoying when actions performed by other sites are delayed. It can become devastating if your own local actions are delayed, giving poor feedback and inadequate interface responsiveness (Greenberg & Roseman, 1997).

A replicated architecture can help compensate for a slow network by giving rapid feedback for local operations. When the user makes a change to a shared piece of data, that change can be displayed locally extremely quickly. While the change may still appear sluggish at other sites as it takes time to propagate across the network, the system will at least be responsive to the user's own interactions (Greenberg et al., 1992).

In a centralized architecture, both local changes and remote changes will be affected by a slow network. A local action must be propagated across the network, executed at the central node, and then the changes to the display must be propagated back. A slow network will create a significant delay between the time the action is sent to the server and the time the user who initiated the action receives feedback.

Distributed Execution allows several sites to make changes simultaneously by utilizing the computing power of all the sites. A replicated architecture allows users working in different areas of the same application to perform work in parallel. If a particular operation requires a lot of computation power, a single site can do the work locally and then broadcast the result to all the other sites (Anupam & Bajaj, 1993).

In a centralized architecture, there is a single process that performs all the work. The single process serializes all the operations from the various sites and can become a performance bottleneck for applications where there are operations that take a long time.

Scalability concerns how many users can be present in a single conference. An application can quickly become unusable as communication times and processing times increase. A replicated architecture can reduce the amount of information transmitted between sites. For example, when the idea 'C' was added to the list in Figure 2-2, only the add command needed to be transmitted across the network. The cost of adding the additional site is the extra time it takes to do the broadcast (Greenberg & Roseman, 1997).

In contrast, the centralized version had to handle every primitive key press, as the server handles both the input and the output. When a message comes in from one of the sites, the central server must not only process the request, but also update each of the displays. This can be an expensive operation.

Heterogeneous environments are common, and it is possible that several members of a team will be working on different hardware platforms. For example, one user may have an Apple Macintosh, while the other has a Sun workstation. A groupware application must be able to run on all the various platforms.

A replicated architecture simplifies this problem, because machine specific code can be written for each platform. The developer recodes and compiles the application for a particular platform. It is only the communication protocols and data structures that must be consistent between sites (Roseman & Greenberg, 1993).

In a centralized architecture this is much more difficult as the centralized process must be able to invoke commands in the windowing system. While protocols such as X allow developers to do this, however they are not available on all platforms.

Multiple Views. Sometimes users will require different views of the same data. For example, a supervisor and a worker who are collaborating on a project may need different levels of detail. The worker's view may show detailed information about a particular component of the project, while the supervisor's view may show general information about the project.

In a replicated architecture, new views can be created by programmers without requiring recompilation of the entire application. As with heterogeneous environments, a new site-specific client can be written which uses the new view, as long as it adheres to the same shared data structures and protocols as other sites (Roseman & Greenberg, 1993).

A centralized architecture requires the developer to rebuild the entire centralized application and to contain all the possible variations of the interface. If there are many different views to construct the application can become quite large.

Reliability. Sometimes both replicated and centralized architectures have good solutions for a problem. However the trade-offs are simply different. This is the case with reliability. Reliability is a problem in groupware applications because individual sites may become unstable or fail in a distributed environment. If one of the sites goes down, the toolkit must recover the data and keep the conference running. In a centralized architecture, this is relatively easy as the centralized site simply closes the connection to the aberrant site. Similarly, it is unlikely that the centralized site will fail because it is likely to reside on a stable machine (Patterson et al., 1996). However, if the server does go down, the entire conference will be halted. On the other hand, replicated architectures handle reliability in a different way. If a single site goes down, the rest of the conference can keep running, losing only a single participant (Greenberg & Roseman, 1997). The trade-off is that there may be complicated recovery techniques that are required as various sites are left in inconsistent states (Patterson et al., 1996). Indeed no replicated architectures currently handle reliability, even though they can do so in principle.

No one architecture is superior to the other in this case and developers must choose the type of reliability that their application requires. If there is a stable centralized server available a centralized architecture would be easier to implement; however, when no stable server is available, a replicated approach might be more appropriate.

2.1.3 Issues in Choosing a Concurrency Control Scheme for Replicated Architectures

Concurrency control was raised as a potential problem in the previous section. Because this is a critical aspect of replicated architectures, it has received much attention and deserves more discussion. Concurrency problems arise in groupware applications that have data distributed among several processes. In a centralized approach, there is a single copy of the data and all the operations on the data occur in a single process, effectively serializing the user's operations on the data and eliminating the need for concurrency control. In a replicated approach, however, the same piece of data can be accessed simultaneously at two different sites, as we saw in the example in Figure 2-3 of Section 2.1.1. With multiple applications communicating over a network, commands that change the data may arrive at different sites in different orders and the data may become inconsistent. In this case a strategy is needed to ensure that the programs can stop or control these inconsistencies. Greenberg & Marwood (1994) identify four common strategies:

- *Non-optimistic serialization* ensures that all messages execute in the same order on all sites. It means that the message cannot be executed locally until the application is assured that all messages which executed before it on any other site have arrived and have executed. One strategy for implementing this is by using timestamps as discussed by Lamport (1978). However, the delay caused by waiting to ensure that all messages have been received may disrupt the work of the user, as they may not receive feedback quickly.

- *Optimistic serialization* assumes that messages will arrive in the order in which they were sent. As such, it executes each message when it is received. However, messages that arrive out of order must be detected and then the damage must be repaired. In one strategy, the site repairs out of order sequences by undoing the out of order messages and then redoing the messages in the correct order. This will have the side-effect of the user seeing things undone and then redone occasionally, as message execution is repaired.

  A second strategy uses transformation. The transformation algorithm ensures that the objects end up in a consistent state even if messages arrive in different orders at different sites. It does this by modifying the data, using a set of rules, when messages sent to it arrive out of order (Ellis & Gibbs, 1989). By being able to execute messages as they arrive, the site will minimize the time between when a message is broadcast and when it is executed at a remote site, making remote operations appear fast. Additionally, the user will see quick local feedback as their messages are applied to the data immediately. This is appropriate when order conflicts are rare.

- *Non-Optimistic Locking* enforces mutually exclusive access to data. In order to change data a site must acquire a lock. Once the lock has been acquired, the site can make the required changes, and then it must release the lock to allow other sites to make changes. This may disrupt work, as the user must now wait for the lock on networks with long latencies.

- *Optimistic Locking* works by using provisional locks. A provisional lock gives a temporary approval of the lock whenever the lock is requested. However, the temporary approval may be revoked later if another site had previously requested and received the lock. When approval is revoked, the application must undo any changes made during the period where it had the provisional lock. The advantage to this scheme is that the user will see quick local updates on his display, but may also see unusual behaviour in the application as their actions are undone if the lock is refused.

Further, it is not always clear that concurrency needs to be managed. For example, Greenberg & Marwood (1994) point out the example of a shared white board, where inconsistencies of a few pixels at different sites do not matter as users may not notice these inconsistencies. Additionally, concurrency may be handled by the users without the need for software intervention, through social protocols (Greenberg & Marwood, 1994). Stefik et al. (1987) noticed that people using groupware do not usually interfere with one another's work. For example, it would be rude to destroy an object someone is currently using. In certain circumstances, social protocols may mitigate the need for concurrency control.

There are many other concurrency strategies, such as transaction mechanisms, immutable objects (Moran et al., 1995), and read/write locks. Various different implementations of these strategies can result in different trade-offs between speed and consistency. Barghouti and Kaiser (1991), Greenberg & Marwood (1994), and Ellis & Gibbs (1986) provide comprehensive surveys of the available techniques for concurrency control.

In summary, optimistic strategies are useful when network response times are slow, but may confuse the user when the data must be 'rolled back' because of a missed lock or serialization event. Non-optimistic strategies will always ensure that the data is always in a consistent state, at a cost of higher latency. With no concurrency control, inconsistencies may exist between sites. However, in certain circumstances (such as a whiteboard), users may not notice small differences and the speed gains may be substantial.

2.2 Examples of Data Sharing Strategies in Runtime Architectures

The earliest toolkits developed had process architectures close to the models mentioned previously, centralized or replicated. MMConf (Crowley et al., 1990), Conference Toolkit (Bonfiglio et al., 1989) and ShareKit (Jahn, 1995) use replicated designs, while Rendezvous (Patterson et al., 1990) and Rapport (Ahuja et al., 1990) use centralized designs. Other implementations, including GroupKit (Roseman & Greenberg, 1992), ObjectWorld (Tou et al., 1994), Notification Server (Patterson et al., 1996), Clock (Graham & Urnes, 1996), and MEAD (Bentley et al., 1994), all use variations that combine some elements of the replicated architecture with elements of a centralized architecture. Finally, Prospero (Dourish, 1996) examines the aspect of flexibility in concurrency in groupware architectures. In this section we will examine some examples of these toolkits.

Rendezvous (Patterson et al., 1990; Hill, 1992; Hill et al., 1993) is primarily a centralized architecture with the majority of the application residing on a single machine. Rendezvous is made up of three major components: virtual terminals, for displaying output and collecting input; the application thread, which executes the application code; and interaction threads, which determine how the information should be displayed for each user. The majority of the work is done by the interaction thread and the application thread, which reside in a single centralized process.

By using different interaction threads the application can present multiple different views to the data. This interaction thread serves as an intermediary between the application and the display, by interpreting the shared data and updating the display of its associated virtual terminal. For example, in Figure 2-4 we can see how different users have different displays. In this case, the interaction threads 2 and 3 display the application data as histograms through the virtual terminal. On the other hand, interaction thread 1 displays its data as a line graph. By modifying the interaction thread, developers can create different ways for viewing the same data.

Rendezvous' creators claim the benefits from the centralized architecture eliminates the need for concurrency control mechanisms, eliminates concerns about different application versions, and has good reliability. Rendezvous suffers because a single process handles both the application and the interaction threads. This means that the processor must update all the displays as well as execute the code for the application. When there are multiple sites used in intense graphical applications, the centralized server becomes a bottleneck and performance suffers.

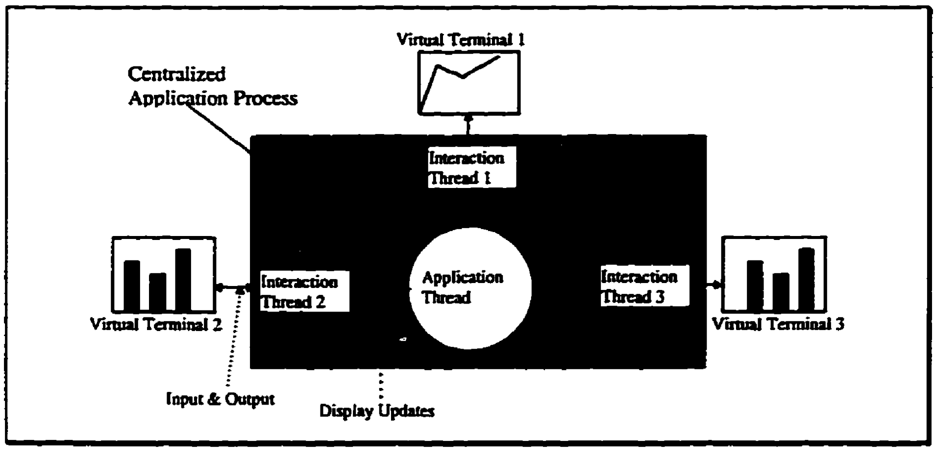

Figure 2-4. Rendezvous' centralized architecture.

MMConf (Crowley et al., 1990) uses a purely replicated architecture and supports several applications executing together in a single conference (such as a whiteboard, text editor, and brainstormer). Rather than having each application connect individually to all the other applications, MMConf uses a conference manager at each site to handle the problems associated with maintaining the communications between sites. Figure 2-5 illustrates how a message to draw a line from the whiteboard at site 1 is sent to the conference manager, which broadcasts it to all the other sites. When the conference manager at the other site receives the message, it forwards the message on to the local copy of the whiteboard application that is running. In essence the conference manager acts as a router and a multi-casting agent.

MMConf allows each site to process the local changes before they are broadcast, so that users can receive quick feedback for local operations. However, the replication introduces several problems. First, in order to ensure that all the data is kept consistent, MMConf only allows a single user at a time to make changes in the application at a time using floor control (Greenberg, 1991), which is overly restrictive. Second, users may have different

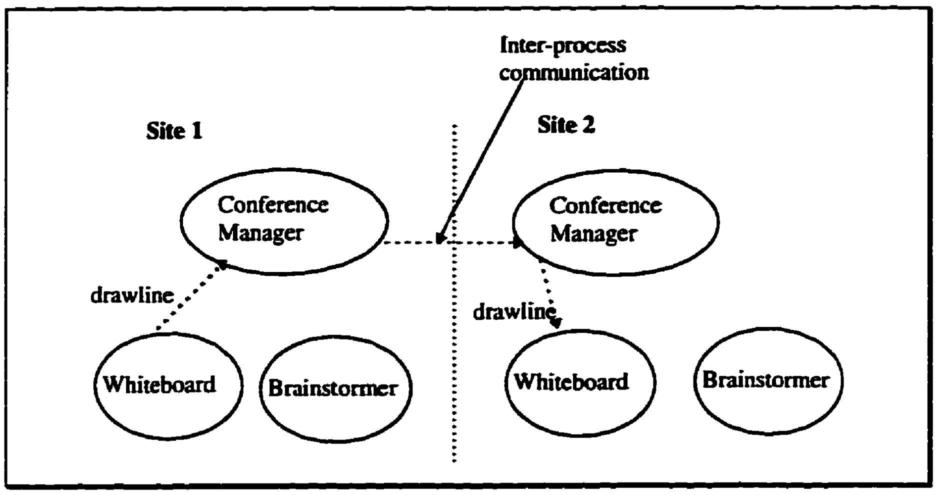

Figure 2-5. MMConf's replicated architecture.

versions of the application. In order to get around this problem, MMConf uses a file transfer protocol that allows each site to request an updated version of the program. Third, the user interface to connect with other users is unwieldy. When starting the conference the user must specify the address of all the other sites, if they are to connect to them.

2.2.1 Modified Replicated Architectures

Although the replicated and centralized architectures are often referred to in the literature as strategies used for implementing toolkits, new architectures combine them to best suit their needs.

Several toolkits combine aspects of centralized systems into a predominantly replicated architecture. The idea is to mitigate the shortcomings of the replicated architecture, such as the problems of connecting to multiple different sites, adding persistence, handling latecomers, and managing consistency.

GroupKit (Roseman, 1993) is a predominantly replicated architecture, where applications and the session management are completely replicated. However, GroupKit employs a centralized component (called the registrar) to help solve the complex problem of identifying all the sites to connect to. The user connects to the registrar, which then manages the connections to the other sites for the client. The registrar accomplishes this by keeping a list of all the sites participating in a conference and automatically establishes connections with them. GroupKit has also used this feature to allow users to browse existing conferences and join them while they are in progress.

GroupKit does not supply a default form of concurrency control but rather leaves it up to the developer. By default, replicated data in GroupKit has no concurrency control, and messages may arrive at different sites in different orders. As discussed previously, this is reasonable when a shared application (such as a whiteboard) does not need to be entirely consistent.

GroupKit does provide the option of serialization for concurrency control, which developers can selectively apply to data. Internally, GroupKit uses a special multi-casting procedure to route all messages through a single arbitrary site, ensuring that the messages arrive at each of the sites in the same order. In effect, the site through which messages are routed becomes a centralized server which dispatches messages to all the other sites.

ObjectWorld (Tou et al, 1994) adds persistence to a replicated architecture by adding a centralized server that acts as a repository for the shared data. The server records the current state of the conference in a file for later recall. When the conference is resumed after having been terminated at an earlier point, each site queries the server for the saved state of the data which then transfers it to them. Additionally, ObjectWorld uses this technique to update latecomers to the conference: when a new participant joins the conference they are automatically updated from this central repository.

ObjectWorld uses several strategies to control the consistency of the objects. *Dependency detection* controls the misordered arrival of messages by uniquely identifying the state of the objects they operate on before the message executes. If the object is in the wrong state, the system automatically copies the most recent version of the object from the centralized store. *Non-optimistic locking* allows the developer to prevent simultaneous access to objects by explicitly acquiring and releasing locks. Finally, a *special broadcast protocol* checks to ensure that all objects are available in the process when a message is received. If the object is not present, the system copies it from the broadcasting site. This concurrency control strategy is built in and all shared objects automatically use it whether it is required or not. Unlike other methods that require developers to explicitly request locks for data, or explicitly funnel data through a single process such as in GroupKit, ObjectWorld makes consistency implicit. Whenever an object is shared, its consistency is automatically guaranteed and maintained by the system.

2.2.2 Modified Centralized Architectures

Centralized architectures are often modified to reduce the amount of work that is done in the centralized server, as well as to reduce the amount of communication between the client sites and the server itself.

To reduce the amount of computation that is done in the centralized server, **MEAD** (Bentley et al, 1994) separates the shared application data and operations on that data from the user interface. The shared data and application are contained in the server, while the interface and display are performed by each of the client sites. This reduces the workload of the centralized site by having the client machines interpret the shared data and update the display on their own. For example, when there is a change in the application data, the server informs the client of the change. The client then responds to that change by querying the current state of the shared data, and updating its display appropriately. This type of implementation reduces the load on the server, by removing all responsibility for maintaining the display, which then becomes the responsibility of the clients.

Patterson's **Notification Server** (Patterson et al., 1996) reduces the computation done in the server by having clients perform the work for the application. The server only maintains the state of the data. The Notification Server has two functions: 1) to store a consistent copy of the data and manage locks for that data and 2) to notify sites using a piece of data when a change is made to that data.

To modify a piece of data, the developer must go through five steps: 1) get the lock for the data; 2) copy the data from the repository; 3) make the change to the data; 4) copy the data back to the repository; and 5) release the lock. The other function of the notification server is to inform sites when a change is made to a piece of data. Each site must register itself with the Notification Server, indicating what information it is interested in. When a change is made to that data, the registered clients are notified. Each client then has to get the current state of the data from the Notification Server.

The Notification Server reduces the computation bottleneck of other centralized architectures. However, it also increases the amount of network traffic as the shared data must be transmitted between the Notification Server and the client site whenever a change is made.

Clock (Graham & Urnes, 1996a; Graham & Urnes, 1996b) attempts to reduce both the amount of computation done on the server, as well as the time it takes for communication to be transmitted across the network. As with MEAD, the client handles the display updates. Unlike these toolkits, the client and the server also co-ordinate with each other to speed up communication by caching data that is normally stored in the server at the client site. The cache contains results of requests that have been made by the client to the server on the assumption that same request will be made frequently. If the result of a request is available in the local cache, the cost of a remote request is saved. The cache entries of the clients are invalidated by the centralized server whenever a change is made to shared data that affects a cache entry.

Clock uses two additional techniques to speed up the delivery of messages: *request prefetch* and *request presend*. The assumption is that the cache predicts the requests that the application is going to make, so during idle points in the program the client and server can attempt to keep the cache from going stale. In request prefetch, the client examines its cache, finds stale entries, and asks the server for updates for those entries. The request presend is performed by the server, which has knowledge about the cache at each site (remember it must invalidate the entries). When the server determines that a cache entry on a client is "stale" (has been invalidated), it automatically sends an update to the site.

The Clock implementation brings the centralized architecture closer to the replicated architecture by copying frequently used data out to each of the client sites, thus reducing communication times and allowing fast local updates.

2.2.3 A Flexible Architecture

In work parallel to this thesis, the **Prospero** groupware toolkit (Dourish, 1996) uses open implementation (presented in Chapter 3) to let application developers manage multiple concurrency control implementations in their application. Dourish recognizes that concurrency control strategies depend on the needs of a particular application. To accomplish this, he explores how consistency is handled in groupware systems. He claims that most groupware systems use *inconsistency avoidance* rather than *consistency management*. In inconsistency avoidance, the system focuses on raising barriers to prevent parallel work from being done on a single piece of data. For example, a lock only allows a single user to access a particular piece of data at a time. On the other hand, consistency management re-synchronizes data after multiple users have worked on it in parallel. Dourish creates a consistency management model that allows data to diverge (become inconsistent) and then converge (or re-synchronize). Using this, developers can specify arbitrary consistency schemes that are suitable to their applications.

In order to support this work, Dourish also proposes a simple open implementation for distributing data, to give the developer a degree of control over how data is distributed. As we shall discuss in Chapter 8, Dourish's work complements the work presented here by emphasizing concurrency control, while we emphasize data distribution.

2.3 Requirements for Data Sharing

The current generation of toolkits define the sharing and concurrency strategy for the entire application and assume that the strategy provided for sharing data will be useful for all the data across all groupware applications. Researchers have not reached a consensus on which particular data sharing strategy should be used in groupware applications, however they make valid arguments for and against each strategy. Currently, the way data is shared in groupware applications built using a toolkit is based on what the *toolkit developer* supplies.

We argue that the *application developer* should be given control over the technique used to share data, where the particular properties of the data being shared and how it is used by the application should determine which techniques should be employed. Additionally, the data sharing strategy should not apply to the entire application, but rather to particular pieces of data. The developer needs to control the distribution of the data and the concurrency control mechanism.

Distribution of the data. The application developer needs to control how each piece of data is distributed. For example, different pieces of data may require different levels of feedback when the user makes a change. To illustrate, when a user draws a line on a drawing surface, they need immediate feedback so they can see their pencil stroke, which replicated data is well suited to. Conversely, when a user makes a query about the participants in the conference, they may not require immediate feedback and a centralized implementation could be used.

Additionally, techniques that are unwieldy across the entire application may be useful when considering individual pieces of data. For example, data migration (Nascimento & Dollimore, 1992) could be used in the case where one user accesses a particular piece of data frequently. In this case, rather than have the data remain at a single location (where the user might have to access it remotely) the data could be moved to the site using it most frequently to reduce bandwidth and transmission times.

Consistency. Users need control over consistency because various different consistency strategies may be appropriate. Consistency can impact the performance of the application, and the time it takes users to get feedback on their actions. Some data may require strong consistency, with little regard to performance, such as a bank balance. Other data may accept inconsistencies at the expensive of responsiveness, such as a whiteboard.

We believe that the application developer needs to be able to control how each piece of data in the application is shared by determining both the concurrency mechanism and the way the data is distributed. A toolkit should give the developer the choice. However, no toolkit can implement all the possible combinations of concurrency control and data distribution. Thus the application developer should be able to define new strategies, as well as select from existing ones, to meet their particular requirements. In the next chapter we will see how open implementations can afford a degree of control while minimizing coding complexity.

2.4 Summary

In this chapter we have shown some of the different ways toolkits hardcode the sharing of data. However, an application may have need for different sharing strategies for different pieces of data. We argue that the application developer should be able to choose from different strategies, as well as define new strategies for sharing individual pieces of data.

3. Adding Flexibility to a Runtime Architecture

The previous chapter revealed several different strategies for implementing both centralized and replicated runtime architectures. The examples illustrated that every toolkit chooses different ways of implementing its components, which ultimately produces trade-offs in performance, ease of implementation, and consistency. Yet we expect that groupware applications built on these toolkits will become more complex and diverse. Consequently, the particular implementation choices hard-wired into these toolkits may prove a poor match to the demands of future applications. Runtime architectures in groupware toolkits must become more flexible to let application developers control aspects of the underlying implementation, including the ability to select how and where trade-offs are made.

In this chapter, we will examine a technique called *open implementation* (Kiczales et al., 1995) for adding flexibility to toolkits. We will apply this technique to groupware toolkits to give the application developer fine grained control over the runtime architecture to choose how data is located, and how data is distributed across the network.

In the first section, we argue that runtime architectures in toolkits need to be more flexible. I then show how the 'API' black box strategy now used in toolkits restricts the set of applications that can be reasonably constructed. We then introduce the concept of open implementation, where the developer is given the power to control a selected subset of the internal implementation details of the toolkit. I will conclude the chapter by applying the principles of open implementation to runtime architectures of groupware toolkits.

3.1 A Call for Flexibility in Groupware Toolkits

Current groupware toolkits, although aiding the development of applications, restrict the application developer to the trade-offs inherent within a particular runtime architecture.

Many toolkit developers recognize that their designs are somewhat rigid and that new toolkit should provide greater flexibility in the runtime architecture. Greenberg & Roseman (1997) point out:

> "Perhaps what is required is a dynamic and reactive groupware architecture, where the decision of what parts of the architecture should be replicated or centralized can be adjusted [by the developer] at run time to fit the needs of particular applications and site configurations."

Similarly, Cortes (1994) after examining toolkits concludes:

> "We consider that designers and programmers should be able to define the internal process structure according to the needs of each application."

Bentley et al. (1994) when discussing the design of MEAD strike a similar chord:

> "Because neither architecture fully meets multi-user interface requirements, a hybrid solution is needed wherein components of the co-operative system are either centralized or replicated, depending on the application requirements."

Patterson et al. (1996) after espousing the benefits of the Notification Server concedes:

> "The difference between GroupKit and Notification Server is a judgement about how often serialization is required. It is perfectly reasonable, however, to use both..."

This problem of inflexibility in toolkits is not unique to groupware, and has been extensively studied by Kiczales (1992). In the next section we explore how this problem arises from black box layering inherent in conventional application programming interface (API's). We also identify two techniques that application programmers use to get around this inflexibility.

3.2 The Black Box Approach to Toolkits

All toolkits abstract away implementation details. This allows application developers to concentrate on learning and applying essential building blocks. This is much easier than requiring an understanding and use of a complex set of primitives, e.g., access to complete library source code or an entire class hierarchy. Toolkits decrease complexity by providing the application developer with an abstract interface to its building blocks through an application programming interface (API). An API is a set of predefined functions, methods and/or objects with well-documented behaviour. Figure 3-1 provides a layered model of a toolkit with three identifiable components: the toolkit implementation, the toolkit API, and the application.

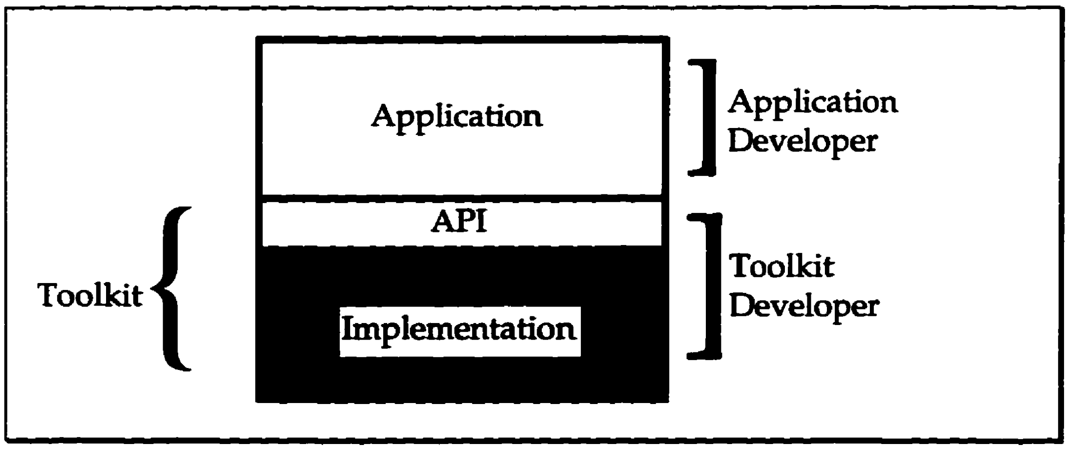

Figure 3-1. The implementation of a traditional toolkit is only available through the API.

The API serves as a barrier, preventing the application developer from having to know about the implementation details of the toolkit (Kiczales, 1992). Through the API, the application developer can manipulate and control the implementation. The internal implementation is seen as a black box, leaving the application developer free to think about their application needs rather than low level details. For example, a window toolkit hides low level issues including how windows are stored in memory and how the mouse is tracked.

When toolkit designers hide implementation details, they make decisions about what implementation strategies will be used to provide functionality. Yet the particular choices may not be suitable for all applications. For example, classic performance decisions in computer science involve the trade-offs between memory requirements and speed. The degree to which a toolkit is useful depends on designers correctly anticipating the needs of its users and choosing an appropriate implementation strategy (Rao, 1993). However, in some problem domains there may not be a single correct strategy. At one extreme, toolkits which must support a wide range of activities may have to create a large API, resulting in an overly complex and difficult programming system, such as the X interface (Rao, 1993). At the other extreme an over-simplified API will make the toolkit much easier to learn, but will be more restrictive. The application developer may not be able to use the toolkit since the simplified abstraction provided may be a poor match to the application requirements.

3.2.1 Working Around the Black Box

Despite the inflexibility of the black box approach, programmers do manage to work around toolkits that do not fit their particular application requirements. Two common strategies include *hematomas* and *coding between the lines* (Kiczales, 1992). With *hematomas,* the programmer reimplements some functionality of the toolkit in the application. With *coding between the lines*, the programmer uses knowledge of undocumented features and characteristics of the toolkit implementation to improve performance in the application.

Hematomas. We will use the example of scheduling strategies in operating systems to demonstrate how programmers use hematomas to get around hidden implementation details that affect their application. Scheduling algorithms must handle the problem of high priority processes blocked on a mutex held by a low priority process. The algorithm developer must decide how to reduce the amount of time that the high priority process blocks. For example, Solaris uses priority boost to overcome this problem, where the lower priority process is boosted to the priority of the high priority process until it releases the mutex. In Windows NT, the lower priority process is scheduled normally, and the higher priority process must wait until the lower one has a chance to execute and release the mutex. If this particular problem is important to the application, the developer must either choose an operating system based on the scheduling strategy, or work around the implementation in the operating system. One workaround is to add a hematoma to the operating system mutex implementation. The developer may create a wrapper around the operating system mutex to provide the desired scheduling behaviour. The problem is that the resulting wrapper may be less efficient than the one supplied by the operating system. As well, this produces more code that must be maintained and debugged by the application developer. Still, the application developer uses the hematoma to extend the operating system to get around the hard-wired design decisions made during its implementation.

Coding between the lines. Coding between the lines uses knowledge of the toolkit's particular implementation to get better performance. Virtual memory systems provide a classic example of this. In a large application, objects are paged out to disk (virtual memory) when physical memory fills. Paging memory out to disk and back takes a relatively long time. To improve an application's speed, programmers can allocate objects that are referenced together in the same page of memory. The application developer codes "between the lines", knowing that the virtual memory system swaps out pages of memory rather than, say, bytes or words. An application programmer causes fewer page faults by allocating objects that are referenced together, close together in the memory address space. The problem is that coding between the lines requires programmers to contort their code, possibly by allocating memory in unusual places, to match the underlying implementation of the virtual memory system. Also, code that relies on undocumented features of the implementation, such as the page size, may break when new versions are released.

In groupware toolkits, a major component in the "black box" is the runtime architecture. These black boxes have arbitrarily chosen the way to distribute data (either replicated or centralized) and the consistency maintenance strategy (locking and serialization techniques). The trade-offs inherent in the choices clearly affect application developers, and consequently force developers to create their own hematomas and/or code between the lines. It would be better to add flexibility to toolkits to give application developers appropriate levels of control over the critical implementation details. A technique called *open implementation* does this.

3.3 Flexibility through Open Implementation

One technique for creating a flexible toolkit is to open up the implementation of the toolkit, allowing developers to modify components of the toolkit to meet their particular needs (Kiczales, 1992). Rather than working around a toolkit that does not entirely meet the requirements of the application, an application developer can directly modify selected components of it.

An open implementation provides two levels at which the programmer can use the toolkit: the *programmer interface* and the *meta-interface* (Kiczales, 1996). The programmer interface is the normal API, the functions through which the application programmer makes use of the underlying default implementation. The *meta-interface* describes the behaviour of the toolkit, and gives the application developer a constrained way to extend the toolkit.

The goal is not to foist the responsibility of toolkit construction on the application developer. In most cases the application developer will find that the standard programmer interface suffices. However, when the programmer interface is not adequate, the meta-interface allows them to modify some of the underlying decisions made by the toolkit developer.

The meta-interface is simply an API to the implementation of the toolkit, designed to give application developers the ability to customize the toolkit to meet their particular needs. With the meta-interface the application developer can modify and add new components to the API by directly manipulating the toolkit implementation, as shown in Figure 3-2.

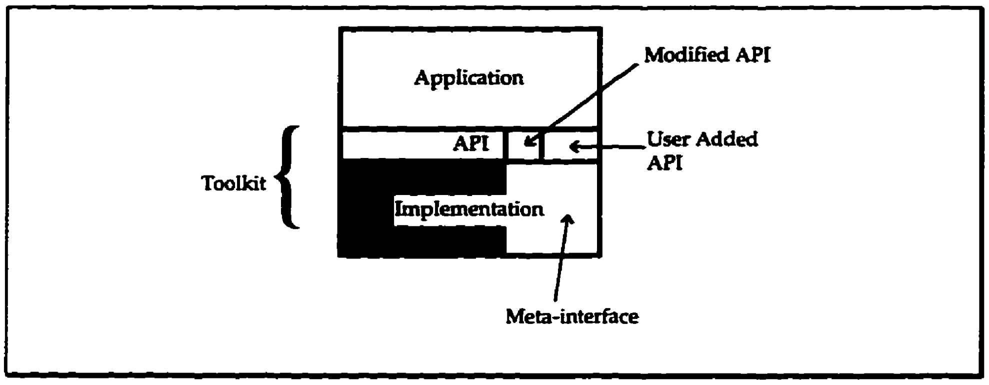

Figure 3-2. The addition of a meta-interface allows developers to refine and extend the API of the toolkit.

3.3.1 The Origins of Open Implementation

The work on open implementation has its roots in computational reflection (Smith, 1982) and metaobject protocols such as CLOS (Paepcke, 1993; Kiczales et al., 1995). These approaches to programming languages allow the developer to inspect the internal workings of a language, and to extend it without modifying existing applications. The ability to modify a language in the language itself, is called *reflection*. The structures that control the behaviours of a language are referred to as *metalevels* in a functional language, and *metaobjects* in an object oriented language.

For example, a metaobject strategy that allows a developer to control how objects are allocated in memory helped solve a particular problem in knowledge representation (Kiczales et al. 1993). Knowledge representation often declares classes with hundreds of slots. However, most of these slots are never used and end up taking large amounts of memory. The CLOS metaobject protocol (Bobrow et al., 1993; Attardi, 1993) allowed the developer to define new ways in which the instance variables of a class are allocated. One implementation strategy used a small hash table strategy for storing the slots of an object, which provided a good representation for the sparse nature of the data.

The effective application of metaobjects and reflection to programming languages prompted Kiczales to consider the technique's applicability to toolkits, using the mechanisms of a meta-interface.

3.3.2 Building a Meta-Interface

Toolkits with an explicit meta-interface expose implementation issues to the developer. However, not all the implementation should be exposed, as insignificant details and areas where the developer does not need control would add unnecessary complexity to the interface (Kiczales et al., 1991). The major problem in building meta-interfaces is determining what the developer should be able to control (Paepcke, 1993; Kiczales et al., 1993).

Potential areas to add meta-interfaces in a toolkit can be identified by examining work-arounds (hematomas and coding between the lines) in existing applications and by examining complaints that application developers have made about the toolkit. The toolkit designer can use these problems to determine which features need to be opened up and made flexible through a meta-interface. For example, in Chapter 2 we identified differences in the runtime architecture of groupware toolkits that are contentious because of the inherent trade-offs; in particular, how data is distributed and the concurrency mechanisms used. These differences indicate the need for a meta-interface in groupware toolkits that gives application developers control over the implementation choices.

From experiences in developing open implementations and meta-object protocols, Kiczales (1995) has developed principles that a toolkit designer must consider when designing a meta-interface. These principles include: conceptual separation, scope control, and incrementality. We describe each and discuss how it can be applied to groupware toolkits.

Conceptual separation means that the application developer should be able to customize particular aspects of the toolkit's implementation, without having to understand the entire meta-interface. For example, if an application developer wishes to modify the concurrency behaviour in a groupware toolkit, they should not have to modify the way data is distributed.

Although it is important to conceptually separate the concerns of distribution and concurrency, this can be difficult in groupware toolkits. Concurrency policies depend on the type of data distribution used. A centralized piece of data requires a different form of concurrency than a replicated piece of data. However, the actual implementations of concurrency schemes must be separated from the implementation of distribution schemes. When defining new concurrency schemes, developers will have to understand the *characteristics of* the distribution scheme, but they should not have to modify the *implementation* of the distribution scheme.

Scope control determines the extent of a change the meta-interface will have in a toolkit. The scope of that change can affect the entire application or be limited to a few small components. By restricting the scope of the change, a toolkit developer reduces the likelihood that a change meant for one component of an application affects another. Secondly, and more importantly, by limiting the scope of a change multiple different behaviours can coexist within the same application.

Naming particular scopes allows developers to distinguish between different behaviours of the toolkit. They can reuse the named components in new contexts, and differentiate between the behaviours. Application developers can then tailor their components by naming the particular behaviours they want for a component.

In a groupware toolkit, scope control can be used to allow different data distribution and concurrency behaviours to coexist within the same application. For example, an application may contain various types of shared data, e.g., centralized, replicated or migrating, based on the way the particular piece of data is used. Scope gives application developers fine grained control over the ways their objects are shared.

Incrementality means that an application developer should be able to modify a toolkit, without having to rewrite their new components from scratch. Application developers thus need good default implementations that can be built upon incrementally. These defaults can also be used by the application developer to understand the implementation of the toolkit.

The default data distribution implementations that should be provided in a groupware toolkit are relatively obvious. Replicated and centralized data are the most popular approaches in use today and already address a wide range of applications. What must be done is to expose their implementations for distributing this data and the concurrency control techniques, allowing the application developer to understand and modify the implementation.

3.4 Summary

In this chapter we have seen a call for flexibility in runtime architectures by developers of groupware toolkits. We argued that open implementation is a feasible way to solve the inflexibility endemic in the current generation of groupware toolkits. We identified distribution of data and concurrency control as two components which require a meta-interface. We introduced the principles of open implementation: conceptual separation, scope control and incrementality and discussed how they could be applied to a meta-interface to control data distribution and concurrency.

In the next chapter, we introduce a prototype groupware toolkit called GEN which provides a meta-interface for data distribution and concurrency.

4. GroupEnvironment: The Programmer's Interface

Chapter 2 showed that current groupware toolkits contain design decisions that ultimately affect the application developer. Toolkits can be too rigid when they do not allow the application developer to select the method of data distribution and concurrency control.

Chapter 3 revealed that the problems of rigidity in groupware toolkits are not specific to this domain, but are more general, stemming from implementing toolkits as black boxes. Recent research in *open implementation* presents a strategy for building toolkits that allows the application developers to control particular design decisions made in the toolkit.

In the next two chapters we will apply the principles of open implementation to a prototype groupware toolkit called the *GroupEnvironment* (or GEN). In particular we will show how a toolkit can provide developers with both a programming interface for standard groupware features and a meta-interface that gives control over the method of data distribution and concurrency control.

This chapter first describes how the requirements of the toolkit impact on the design of the application. It then examines how the runtime architecture is separated into the programmer's interface, the meta-interface and the black box. It continues by detailing the programmer's interface. Finally, we demonstrate how a simple brainstorming application can be built using the programmer's interface. The meta-interface will be described in Chapter 5. Data distribution strategies in the programmer's interface are built on top of this meta-interface. For simplicity, we defer the description of the implementation of those components until Chapter 6, after we have discussed the workings of the meta-interface.

4.1 Overview of Requirements and their Implications for the Design

In this section we look at the three major requirements for the GEN system and give a brief overview on how this impacts on the design. The two major requirements are:

- GEN must be a functional groupware toolkit; and
- GEN must provide a flexible data sharing mechanism.

The way that the flexibility requirement is met in this thesis introduces a third requirement: GEN must provide an open implementation that meets the criteria of scope control, conceptual separation and incrementality. In the following sections we examine these requirements and discuss briefly how they impact on the design of GEN.

A functional groupware toolkit. We are building a groupware toolkit and as such we must provide the typical building blocks that developers require to construct applications. Application developers should be able to use the toolkit without resorting to use the meta-interface when building an application. To meet this requirement GEN provides a *programmer's interface* which contains the building blocks commonly found in groupware toolkits including: session management, notification, organization of data and mechanisms for sharing data. We discuss the programmer's interface in Section 4.4.

Flexible data sharing mechanisms. This requirement has two implications. First, developers must be able to choose from among data sharing strategies or create new ones. Second, the implementation of the sharing strategy must be separated from the implementation of the object being shared so that application developers can use a particular sharing strategy with a variety of different types of data. For example, a replicated sharing strategy should work with an OrderedCollection, Rectangle, or any other object in the system.

Creating and/or choosing data sharing strategies. There were several ways considered for letting developers define the particular sharing strategy they would use. First, we could let application developers choose from a library of pre-existing sharing techniques.

This is too simplistic, as there are a large number of possible combinations of concurrency control and distribution schemes available. If a developer required a specific sharing strategy not supported by the toolkit they would have to work around the toolkit, or discard the toolkit.

As another option, we could provide the application developer with full access to the source code. They could create new strategies by modifying the source. However, this is likely too complex because developers would have to understand the entire toolkit even if their change was relatively minor.

In GEN we chose open implementations as a reasonable way to support flexible data sharing. Open implementations allow both selection of data (through the default implementations) and access to source in a structured way (through the meta-interface). The meta-interface presents a secondary interface to the toolkit that presents a simplified model for how data sharing can be modified in the toolkit. These models are discussed in Section 5.2.

Separating the implementation of the sharing strategy and the object being shared. If the implementation were to mix the sharing strategies with the actual implementation of the objects, the developer would have to modify the implementation of the object's class to change how an object was shared. It would be better to separate the code for sharing the object from the actual implementation of the object. We considered two possible designs.

The first possibility was to put the sharing strategy in a root class, such as a class called ReplicatedObject or CentralizedObject. A particular class could be replicated or centralized by inheriting the chosen behaviour from the appropriate root class. The problem with this strategy is that developers could not have two different instances of the same class with different sharing strategies. Additionally, currently existing classes, such as OrderedCollections, would have to be reimplemented as sharable versions.

The second possibility, and the one chosen, was to use wrappers (Gamma et al., 1995). Wrappers allow the addition of new behaviours transparently and dynamically, while still preserving the normal interface to the object. In GEN this allows us to add the sharing behaviour to the object at runtime, without changing how the developer interacts with the object. Secondly, wrappers do not require changing the class implementation, and the sharing behaviour is specified individually for each instance. For example, developers can have an OrderedCollection instance with no sharing behaviour, others that are centralized, and still others that are replicated - all within the same application. Furthermore, each of the different instances would use the same implementation of OrderedCollection. The properties and use of wrappers is discussed further in Section 5.3.

Open implementation. An open implementation requires two components: a programmer's interface and a meta-interface. The programmer's interface contains the typical components of a groupware toolkit. The meta-interface allows the developer to modify the toolkit. A meta-interface needs to adhere to the principles of scope control, conceptual separation and incrementality. These features are best supported in an object oriented language that allows both inheritance and polymorphism (Rao, 1993) (as we shall discuss in Section 4.3). The implementation of wrappers, mentioned previously, also requires the use of a dynamically typed language. To meet these requirements we use Smalltalk, an object oriented language that supports polymorphism, inheritance and dynamic typing. Finally, because groupware is distributed, we use a distributed object implementation in Smalltalk, as we shall discuss in Section 4.3.

Summary of design motivation. In Figure 4-1 we summarize how the requirements of the system led us to the final design choices. To begin, the toolkit requires both the basic functionality of a groupware toolkit, as well as a flexible data sharing mechanism. The requirements for a functional toolkit are met by the programmer's interface, which we discuss later in this chapter.

The flexible data sharing requirement is handled using an open implementation, and exposes the details of how data is shared in the meta-interface. The meta-interface must address the issues of scope control, incrementality and conceptual separation, leading to the use of an object oriented language.

We also need to allow flexible sharing strategies to be applied to many different types of objects, without changing their implementation. This led to the use of wrappers. Finally, because groupware is based on distributed languages, we need a distributed object implementation to share information between sites.

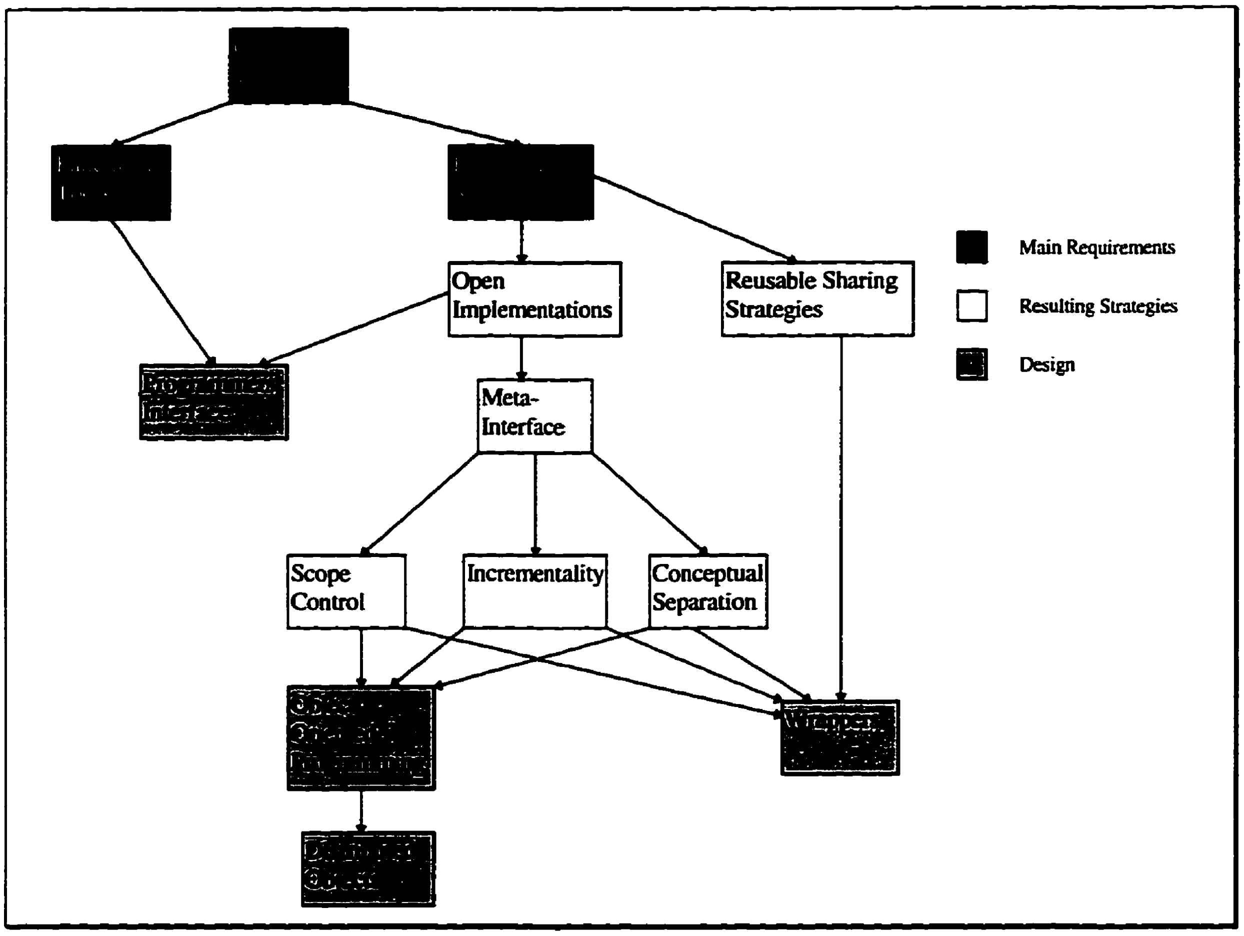

Figure 4-1. How the requirements are met by the design.

4.2 The Structure of GEN

GEN provides all the core features of a groupware runtime architecture, including a process structure, interprocess communication, distribution of data, concurrency control and notification. The runtime architecture is divided into three separate categories: the black box, the programmer interface, and the meta-interface. The relationship between these categories is shown in Figure 4-2.

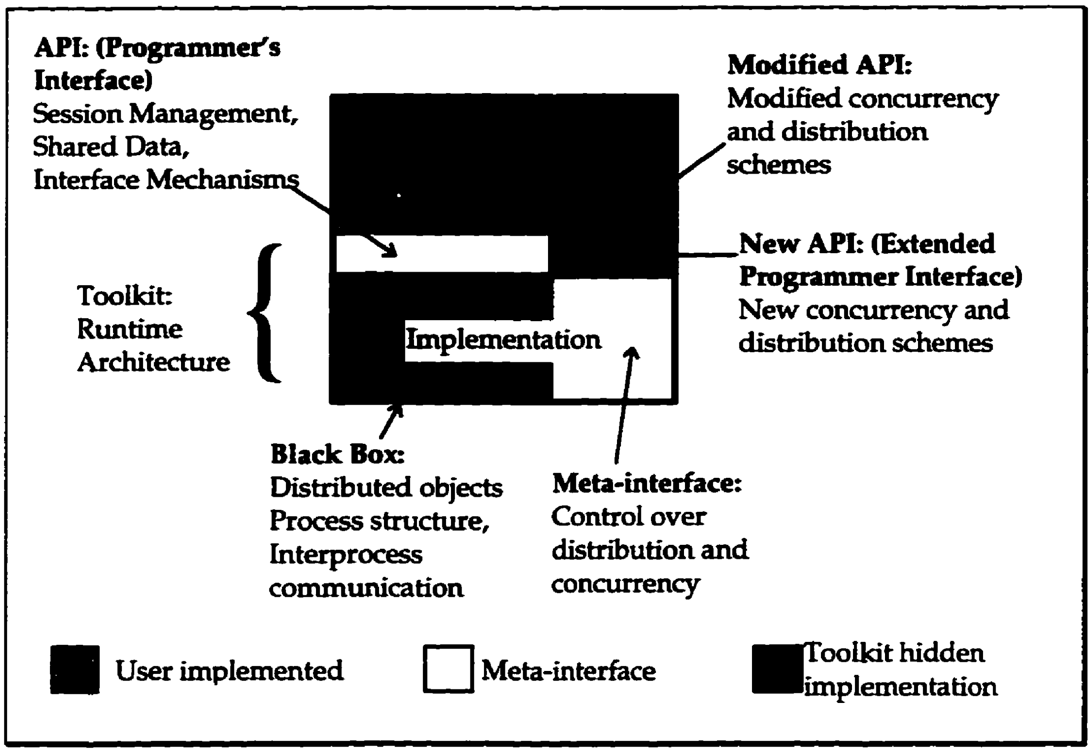
API: (Programmer's Interface)
Session Management,
Shared Data,
Interface Mechanisms
Modified API:
Modified concurrency and distribution schemes
New API: (Extended Programmer Interface)
New concurrency and distribution schemes
Toolkit: Runtime Architecture
Implementation
Black Box:
Distributed objects
Process structure,
Interprocess communication
Meta-interface:
Control over distribution and concurrency
User implemented
Meta-interface
Toolkit hidden implementation

Figure 4-2. Application of Open Implementation to the runtime architectures of a groupware toolkits.

The *black box* contains the part of the implementation that is both fixed and completely hidden from application developers. As seen in Figure 4-2, application developers only access this layer indirectly through the API of the programmer interface. In GEN, the black box implements a distributed objects layer, which in turn handles interprocess communication and process structure. This layer provides the basic mechanisms that allow data sharing between sites. It is used internally to make the higher level building blocks for groupware applications that are supplied by the programmer's interface.

The *programmer's interface* contains the high level building blocks described in Chapter 1: session management and mechanisms for sharing data. The applications built by programmers sit on top of this layer, using the API GEN provides (Figure 4-2). Even if toolkits are constructed using the open implementation strategy, the programmer interface should still be adequate for building most applications. In GEN, the programmer's interface supplies session management and default data sharing strategies. Still, there are many ways that data can be shared, and the toolkit can provide only a few of them in the programmer's interface.

The *meta-interface* gives the programmer a second and more complex API that lets them define new ways of sharing data to fit the particular needs of the application. This is done by reprogramming the way data is distributed and how concurrency is managed. The meta-interface uses the distributed object layer to construct these new sharing strategies, as shown in Figure 4-2. Additionally, the default implementations for the sharing strategies contained in the programmer's interface are supplied, so that developer can modify these to meet their specific needs.

These three components and their interactions will be elaborated in the remainder of the thesis.

4.3 Foundations: Objects, Distributed Objects and Smalltalk

The GEN implementation was built using the Smalltalk object oriented environment and relies heavily on a distributed objects scheme. In this section, we explain why this language and these particular techniques were chosen, and why they are appropriate to the GEN implementation.

The object oriented paradigm. A meta-interface is a backdoor into the implementation of a toolkit. Developers use this backdoor to specialize and change the implementation by modifying the toolkit. Object oriented programming systems are particularly useful for this purpose because the properties of inheritance and polymorphism support the way the meta-interface components (objects) are changed and specialized (Rao, 1993).

Inheritance provides a powerful mechanism for incrementally specifying new behaviours. By inheriting from existing meta-interface classes, application developers can reuse behaviours they are not changing, and override those behaviours they wish to modify. Polymorphism in object oriented languages ensures that meta-interfaces with compatible APIs but different behaviours are interchangeable. In GEN, this allows us to interchange sharing strategies such as replicated and centralized behaviours without changing the programmer's interface.

Smalltalk. Smalltalk was chosen as the implementation language because of its dynamic binding of methods and values (Goldberg & Robson, 1983). Dynamic binding can be used to allow objects to intercept messages on behalf of other objects. As we shall see in Chapter 5, the meta-interface uses message interception to control the distribution of objects, the way messages are routed between sites, and concurrency.

Distributed Objects. Groupware systems require some form of distribution. The distributed objects implementation hides the interprocess communication layer, and process structure needed to build groupware applications. In the same way that remote procedure calls (RPC) allow procedures to be executed at remote sites, distributed objects allow messages to be sent to objects at remote sites. One difference between these strategies concerns where the message/procedure is sent to. The RPC layer sets up a communication path between processes, and the procedure is executed in the process that receives the RPC. Distributed object implementations must go one step further and route messages between individual objects, because each message is executed by a particular object within the process.

Besides routing messages to the appropriate object, distributed systems must also determine how a particular object is distributed. There are two ways an object can be shared between sites, either through a remote reference or a copy.

A remote object is an object that resides on a single site, but can receive and execute messages from other sites *transparently*. A *transparent message send* allows an object to send a message to any other object, without being aware of whether the receiver is local or remote. To achieve transparency, each remote object is represented locally by a*proxy* object. When an object sends a message to a remote object, it is actually sent to a local proxy. This proxy knows the location of its real object counterpart on the remote machine and automatically forwards the message to it. The remote object computes the result and replies to the original sender, through the proxy, without knowing that the sender was on a different machine. For example, Figure 4-3 shows object X sending a message to a remote object A through a proxy contained at the local site. Neither A nor X are aware that the sender or receiver of the message are on different machines.

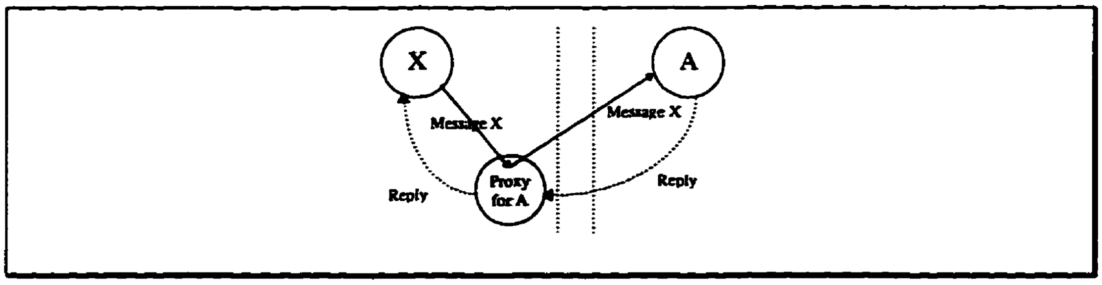

Figure 4-3. When X sends a message to the remote object A it is sent through a proxy object, which forwards the message on to the real object.

Remote referencing of objects is not always the most efficient way to distribute an object and *object copying* can significantly increase the speed of an application when a remote object is not required (Dollimore et al., 1991). In this case, a copy of the object is created on the site it is distributed to. For example, a remote string that is to be read a character at a time can require many remote messages be sent to the object to get each of the characters. By copying this object once, most of these remote message sends are eliminated.

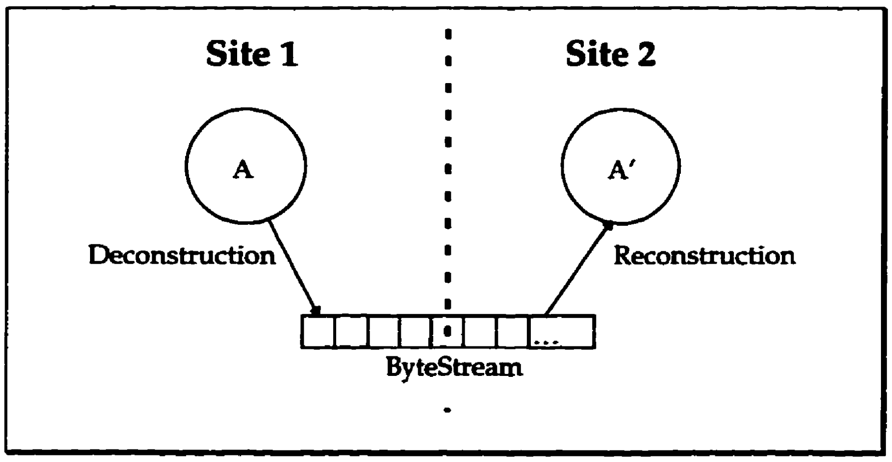

Figure 4-4. The marshalling of an object.

To copy an object between sites it must first be converted to a bytestream and sent over the network. The mechanisms used to convert an object to a byte stream and then back to an object is referred to as *object marshalling*. Figure 4-4 shows how an object is marshalled between two sites. An object is deconstructed into a byte stream, transmitted to the new site and then reconstructed as an exact copy.

At the time of GEN's implementation no distributed object layer with flexibility required was readily available, requiring us to build a rudimentary one. In particular we require fine-grained control over how objects are marshalled between sites which was not present in the publicly available distributed object toolkit Emerald (Hutchinson et al., 1987). As such, GEN shows one way that conventional distributed object layers must be extended when giving developers finer grained control over how objects are distributed.

While substantial, GEN's distributed objects layer lacks strategies for handling fault tolerance, distributed garbage collection, and persistence. These are research topics in their own right, and are beyond the scope of this thesis. The reader is referred to the Emerald system (Hutchinson et al., 1987), Arjuna (Parrington et al., 1995), and Smalltalk distributed objects implementations (Bennett, 1990; Dollimore et al., 1991) for additional information.

Object oriented programming, the Smalltalk environment, and our distributed objects implementation give us a foundation for building higher level components appropriate for constructing a groupware programmer's interface.

4.4 Programmer's Interface

In Chapter 1 we identified a common set of session management and shared data components that groupware toolkits include for groupware programmers. In this section we explore how these are implemented in GEN. We will show that session management is accomplished through the use of a global namespace, allowing developers to connect to particular pieces of shared data. We will also see that the shared data implementation allows the developer to specify how the data is distributed and kept consistent. The shared data layer also provides notification, so that sites can react to remote changes in the data. Finally, we will introduce the concept of environments which allow application developers to organize their shared data.

4.4.1 Session Management

Session management in toolkits involves setting up the low level details of managing connections between machines. This includes the communication infrastructure and the ability to locate either processes in procedural programming paradigms or objects in object oriented programming paradigms.

Communication Paradigm. In procedural programming, developers are given a form of RPC, which hides the details of communicating between machines. In object oriented programming this paradigm is changed to one in which objects communicate. In the previous section we saw how this was accomplished through distributed objects.

Location of Processes or Objects. Distributed functional programming focuses on being able to send remote procedure calls to processes. In toolkits such as GroupKit (Roseman, 1993) the session management layer maintains a list of the processes. In our object

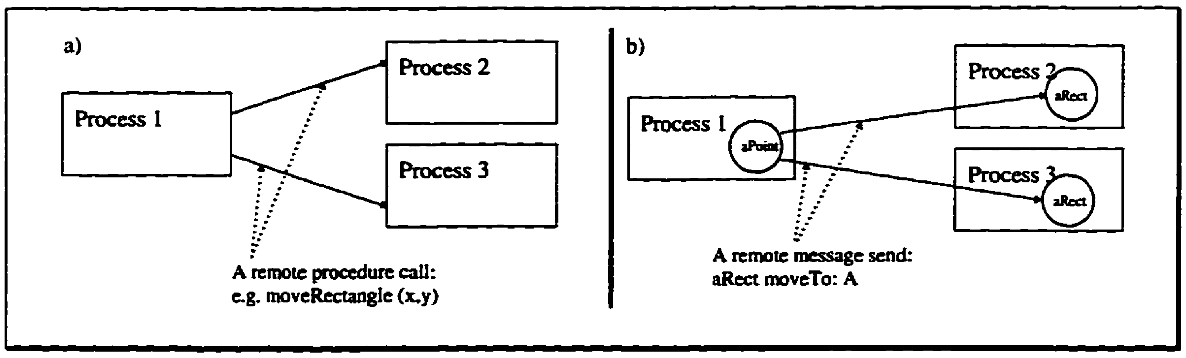

Figure 4-5a shows the functional model of distributed programming, where procedure calls are sent to other procedures. In Figure 4-5b, the object oriented programming model requires that messages be sent to objects contained within the process.

oriented toolkit, the focus is on locating distributed objects. Figure 4-5 shows how these distributed programming paradigms differ. In Figure 4-5a the remote procedure calls to move a rectangle are sent to the process, while in Figure 4-5b, the message to move is sent directly to the rectangle. In the case of an object oriented groupware toolkit, the application developer needs to locate particular objects.

The session management layer in GEN allows developers to locate particular objects by defining a global namespace. A global namespace allows machines to publish objects to an area that all sites can see and give them a unique name. Other processes may obtain references to those objects by looking them up in the global namespace. For example in Figure 4-6, Process 1 obtains a reference to the rectangle object in Process2 by looking it up in the global namespace under the name 'RectangleA'.

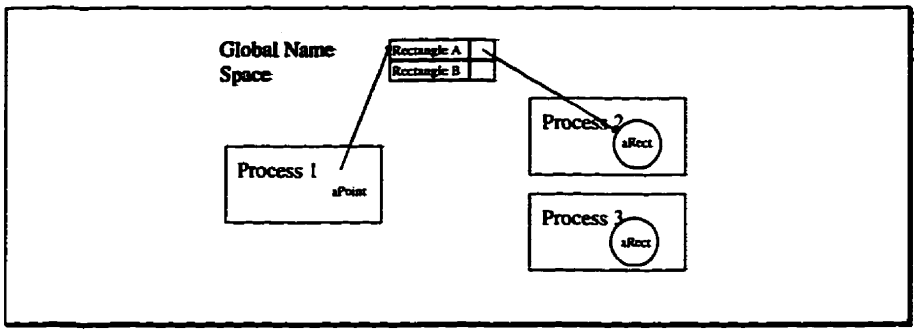

Figure 4-6. The global name space is an address space that all processes can see and access objects by a unique name.

The Global Name Space Implementation and API. The global name space is implemented in GEN as a dictionary that is replicated across all participating sites. When an object is added to this global dictionary at one site, the object is broadcast as a remote reference or copy (see Section 4.3) to all the other global dictionaries in the system. In Figure 4-7 we see that Process 2 has published its *Rectangle* under the name 'RectangleA' as a remote reference, and the other sites can look up the name 'RectangleA' in their local copy of the dictionary to get a reference to the centralized rectangle.

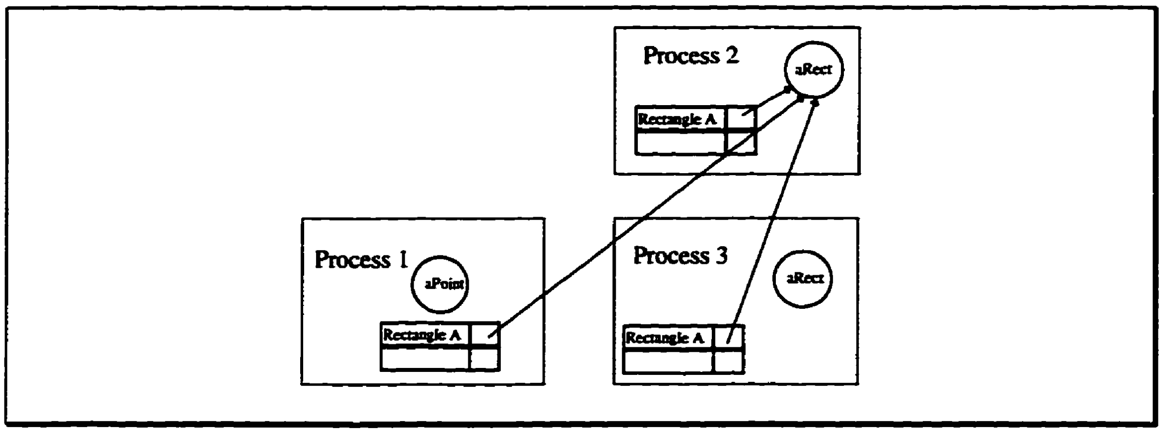

Figure 4-7. The implementation of the global name space is through a dictionary which exists on each site.

The operations for manipulating the *ObjectDirectory* is indicated by their method protocols as given in Table 4-1. For readers not familiar with Smalltalk a method is an operation on the object that owns it. The first operation of the API connects the *ObjectDirectory* to the object directories at other sites. Specifying a site and a port in the message *#addToMainDirectory:port:* connects the object directory to a remote site (by convention a work group will maintain an object directory on a well known host and port). The next two operations are used to publish and retrieve objects from the global namespace. The *#addObject:named:* protocol specifies that an object be added to this global namespace, with its name given as an argument (which is a string). The *#objectNamed:* protocol lets the developer get a reference to a global object by specifying the name in the argument. Finally, the *#removeObjectNamed:* protocol removes an object from the global namespace.

The session management layer of GEN provides the infrastructure for distributing and locating objects between sites. The following section looks at abstractions in GEN for sharing data.

4.4.2 Support for Shared Data

Although distributed objects provide the primitives necessary for the development of groupware applications, developers require higher level building blocks. The following section shows how the GEN API provides for data distribution abstractions in the form of replicated or centralized objects, concurrency control through *atomic* objects, data

Protocol	**Effect**
ObjectDirectory addToMainDirectory: anInternetAddress port: aPortNumber	Connects the current machine to the object directory on the remote machine.
ObjectDirectory addObject: anObject named: aString	Stores an object in the global name space
ObjectDirectory objectNamed: aString	Answers a reference to the object named a name
ObjectDirectory removeObjectNamed: aString	Removes an object in the global name space

Table 4-1. Protocols for publishing objects into the global space

organization strategies using environments, and notification using call-backs.

An object oriented groupware toolkit differs from procedural toolkits, because of its focus on data rather than procedures. First, we show how concurrency control and distribution of objects differ from procedural programming because they can be encapsulated in the object itself. Later in this section we will see how notification is integrated with the object, and show how a separate entity known as an *environment* is used to help organize objects.

4.4.2.1 Data Distribution and Concurrency through Replicated and Centralized Objects

In procedural programming, the application developer typically manages communication through remote procedure calls. Procedural toolkits abstract this in architecture dependent ways: replicated architectures use multi-casting while centralized architectures send the RPC to a unique site. In contrast, object-oriented toolkits send messages directly to the object as if it resides on the local machine. The object itself determines how the messages are broadcast to their distributed counterparts, depending on how the object is distributed across the network.

GEN provides two basic configurations of distributed objects. First, a *replicated object* maintains multiple copies of itself on each of the machines. When a message is received by a copy of the object, it will automatically broadcast the message to all the other copies. Second, a *centralized object* maintains an actual object on one machine and object proxies at the other sites. When a message is sent to a proxy, it automatically forwards the message to the site containing the actual object. In either implementation, the application developer does not have to consider how the message is handled; the object itself either broadcasts or forwards the message as needed.

Consistency becomes problematic with distributed objects that have copies on different machines. They can become inconsistent when messages arrive at different machines in different orders. One solution is to make objects *atomic* (Stroud & Wu, 1995). An atomic object in GEN is guaranteed to keep all the copies of the object consistent. In the same way that the developer does not have to consider how a message is distributed around a network, the developer does not have to worry about how the object will be kept consistent; the object itself will keep itself consistent. There are many different ways that atomic objects can be implemented (Kittlitz, 1994; Stroud & Wu, 1995). Later in this section we will present one implementation that uses a locking element to maintain consistency by acquiring a lock before each message is sent to the various copies of the object. The lock is released only after the message has finished executing at all the sites.

Using GEN's programmer interface, developers can choose either replicated, centralized, or replicated-locking data distribution on a per object basis. Objects using these different data distribution and concurrency strategies can coexist in the same application. In the following paragraphs, we shall briefly describe how these are implemented in GEN and

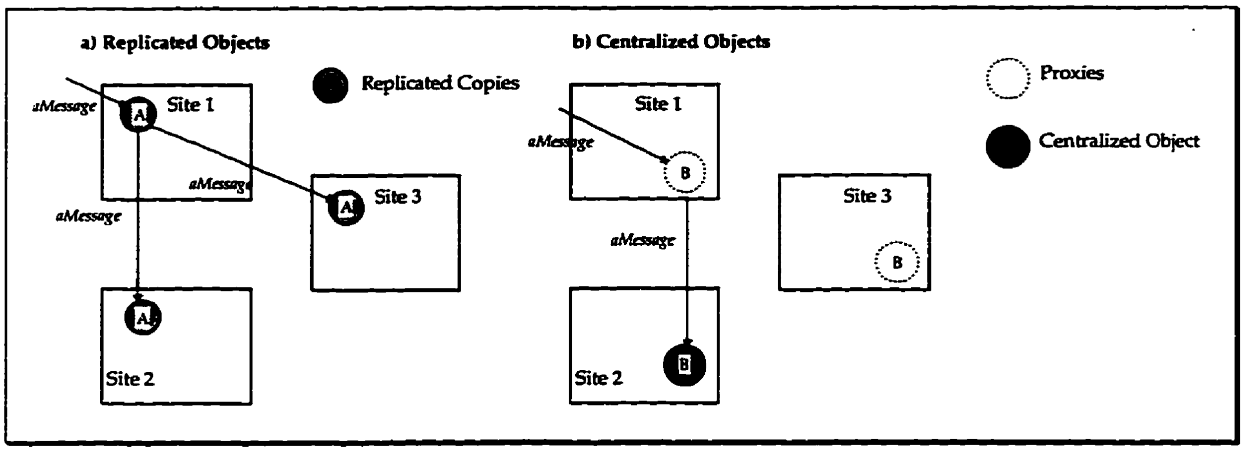

Figure 4-8. Example default object distribution schemes, replicated and centralized.

accessed through the API.

Replicated objects automatically create a copy of themselves when they are distributed to a new site. Object replicas are synchronized by having messages sent to one replica broadcast to all the replicas on the other sites. Figure 4-8a illustrates this. A message is sent to object A at Site 1 which is then broadcast to its replicas at Sites 2 and 3. GEN's default implementation of replicated objects does not include any form of concurrency control, and messages may arrive at different sites in different orders. As mentioned previously, this is a reasonable default for many environments, such as whiteboards which do not require a high level of consistency (Greenberg and Marwood, 1994).

Replicated-Locking objects. The replicated locking object is a modification of the replicated object. These objects automatically maintain a single centralized lock which can only be held by a single replica at a time. This lock is automatically acquired when a message is received by an object. The message is then broadcast to all sites, and executed at each of those sites. Once execution has completed at all sites, the lock is automatically released. If a message is received when the lock is held by another replica, it will wait until the message with the lock has completely finished executing, before either executing locally or being broadcast. This guarantees that a message sent to any replica of an object is executed at all sites before the next message is processed.

Centralized objects. A centralized object has a single copy of the object located at a single site. All messages sent from remote sites are forwarded to it through a proxy object (Decouchante, 1986; Steele, 1991). Figure 4-8b shows an example where a message sent to the proxy for B on site 1 is automatically forwarded to the actual object B on site 2. In GEN, a centralized object resides at the site which created it. Therefore, although a number of centralized objects may exist, they may not all be located at the same site.

Object Distribution and Latecomers. The actual distribution of shared objects is handled *transparently* by GEN. When a process references a shared object that does not exist locally, the GEN runtime system will automatically replicate the object (if replicated), or make the proxy (if centralized) at the local site. The shared object can be referenced through the global namespace (discussed previously), or when it is passed as a parameter in a message to an object located on another machine.

This model automatically updates latecomers to an ongoing conference because the current state of the objects are automatically distributed when they are referenced by the latecomer. For example, when a latecomer looks up a replicated object in the global namespace (which is automatically included), that object and its current state will be copied to the local machine.

The API: Choosing a distribution strategy. An important part of GEN is that any object can be made sharable (e.g. Smalltalk's current implementations of a Dictionary, OrderedCollection or Rectangle). The protocols used to specify the distribution strategy (*#replicated, #replicatedLocking* or *#centralized* shown in Table 4-2) are applied to an instance of an object after it has been created.

For example, we can create a replicated Rectangle by sending an instance of a rectangle object the message *#replicated.* A new instance of the Rectangle is answered that will automatically create a replica of itself on other sites that reference it.

Protocol	Effect
sharedObject := anArbitraryObject replicated	Answer a replicating version of the object. When the replicating object is referenced remotely, a copy is sent that is automatically kept up to date.
SharedObject := anArbitraryObject replicatedLocking	As above except consistency is guaranteed through a mutually exclusive lock
sharedObject := anArbitraryObject centralized	Answer a centralized version of the object. When the object is referenced remotely the remote, site receives a proxy to it.

Table 4-2. Protocols for specifying the type of distribution and concurrency control for an object.

In this section we have seen how a shared object can be either centralized or replicated, or a replicated locking object. In the following section we will see how notification is also tied to the object.

4.4.2.2 Notification

Notification allows the developer to react to asynchronous events. For example, when another site changes a shared data value, the local site can be notified of the change and perform operations in response, such as updating the interface.

Notification in GEN is done at the message level, where the developer can request notification both before and after particular messages are executed by an object. The developer specifies that an event should be generated when a particular message is received by an object. The developer then attaches a call-back to the event, which will invoke a user specified method which can (say) update the display, or take whatever action is necessary. When the message is sent to the object, the event is generated and the call-back is automatically triggered. Because encapsulation guarantees that an object can change only when a message is sent to it, developers can use notification to capture all changes made to an object.

Figure 4-9 provides an example. In this case, a rectangle is added to a collection of objects to be drawn on a canvas. This is done by sending an add message to the set. The set object generates an *ObjectAdded* event, which in turn invokes the redraw call-back.

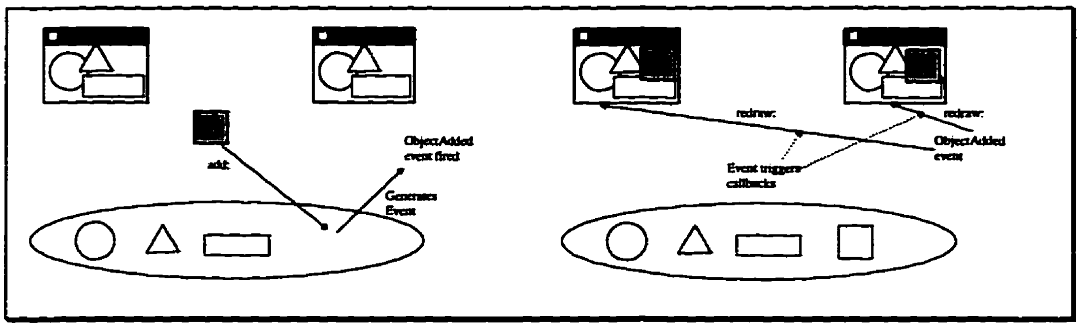

Figure 4-9. The message *#add:* causes the event AddObject to be generated which triggers the call-back *#redraw*.

API: Specifying Events and Call-backs. The developer needs to be able to specify the message, the type of event that it creates, whether the event should be generated before or after the message is executed, and what call-backs are attached to the event. The protocol *#addPreGroupwareEvent:onMessage:*, shown in Table 4-3, causes the receiving object to generate the specified event (aSymbol) before the message with the specified name (aMessageName) is executed. Similarly, *#addPostGroupwareEvent:onMessage:* causes the event to be generated *after* the message has executed.

Call-backs are attached to events through the *#addGroupwareCallback:* protocol and detached with the *#removeGroupwareCallback:* protocol. The parameters to these message include the *receiver*, *selector* and *clientData*. The *receiver* is the object the call-back is sent to (such as a canvas). The *selector* is the message name sent to the receiver, and the *clientData* holds any additional information that the developer wishes to include when the call-back is fired.

Protocol	Effect
aSharedObject addPreGroupwareEvent: aSymbol onMessage: aMessageName	Specifies that an event named aSymbol is triggered before any message name aMessageName is executed
aSharedObject addPostGroupwareEvent: aSymbol onMessage: aMessageName	Specifies that an event named aSymbol is triggered after any message name aMessageName is executed
aSharedObject addGroupwareCallback: aSymbol receiver: anObject selector: aSelector clientData: anObject2	Specifies that *anObject* be sent the selector *aSelector* whenever the event is triggered.
aSharedObject removeGroupwareCallback: aSymbol receiver: anObject selector: aSelector	Specifies that the call-back associated with *anObject* and selector *aSelector* be removed from the event list.

Table 4-3. Protocols for adding events to shared objects.

Note: using notification to separate model and view. Notification is often used to support a separation of the data model from the view (Roseman & Greenberg, 1997). The model holds the underlying shared data while the view is the user interface that the user sees and interacts with on the display. The application developer ties events to the data model and uses call-backs to update the interface. Figure 4-10 shows an example of a histogram view of data. Here, one site makes a change to the underlying data model by sending it a message. This generates an event which triggers a call-back message that is broadcast to both views.

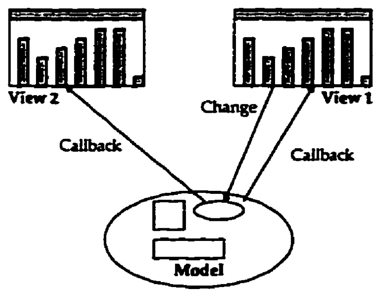

Figure 4-10. Notification is used to keep an interface aware of changes in the models shared data.

4.4.2.3 Organization of Data Through Environments

As in any programming language, large amounts of data are often managed through some form of scoping rule. In GEN, placing all the data into a global namespace can result in two applications inadvertently using the same name for the same piece of data (called a namespace conflict). In order to alleviate this problem, GEN provides a model for organizing data into manageable subunits called *environments*.

An environment (Abelson, Sussman & Sussman, 1986) is an object which associates particular pieces of data with particular names, i.e. it is a structure for defining variables. Scoping rules can be defined to relate environments together by enclosing environments inside one another. As such, if a particular variable is not found in the current environment, the scoping rule can be used to specify how enclosing environments are searched for the variable.

Consider the example in Figure 4-11. The environments BB and CC are enclosed by the environment AA. If the program references a variable in BB's environment, such as A, its value (O) is returned. If the program references a variable that only the enclosing environment (AA) contains, such as Y, then the value in AA (15) is returned. If the variable is not found in the environment, or any of its enclosing environments, an undefined value is returned. Only the enclosing environment is searched. For example, variables in CC are never searched when a request is made in BB.

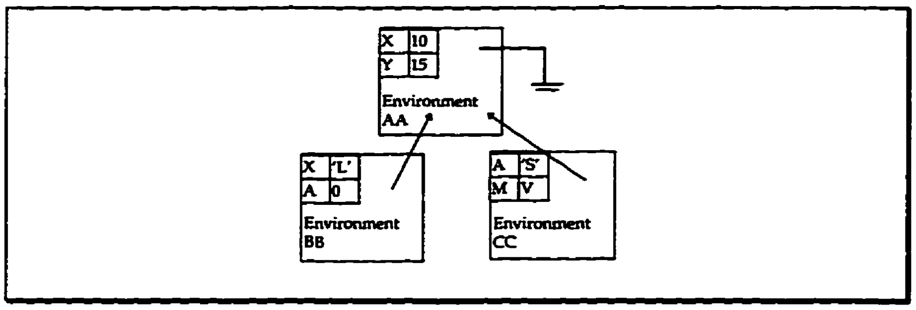

Figure 4-11. A simple environment structure, where AA encloses BB and CC.

Environments are convenient ways to store the shared data model of the program, simplifying the separation of the model from the view (Roseman & Greenberg, 1996). The interface can attach call-backs to particular shared variables contained in that environment. In large applications, the environment serves as a convenient unit for grouping related data, and to avoid name collisions that are possible in a global namespace.

In GEN, environments are first class objects - they are instantiated in the same way as any other object in the system. This means they can be replicated or centralized. They can also use events and call-backs to associate actions with changes in a particular environment.

The API. Table 4-4 shows the protocols for environments which we call *GroupEnvironments*. The *#new* message lets the developer create an environment. The enclosing environment is specified through the *#superEnvironment:* message. New elements are added to an environment through the *#globalAt:put:* protocol and removed

Protocol	**Effect**
GroupwareEnvironment new	Answers a new groupware environment.
aGroupwareEnvironment superEnvironment: anEnvironment	Set the enclosing environment to anEnvironment
aGroupwareEnvironment globalAt: aString put: anObject	Add the object anObject into the environment under the name aString.
aGroupwareEnvironment removeGlobalAt: aString	Remove the object with name aString from the environment.
aGroupwareEnvironment globalAt: aString	Answer the object in the environment named aString. If the environment does not contain aString, search the super environment.

Table 4-4. Protocols for using environments in groupware applications.

through *#removeGlobalAt:*. Finally, a variables value is returned through *#globalAt:*

which searches the environment hierarchy for the variable and returns its associated value.

4.5 A Simple Example: The Brainstormer

This section presents a simple example of a brainstorming application that demonstrates how the components in GEN are used to construct a groupware application. The "brainstormer" is presented in three parts: its user interface; the general model used to construct it; and then a detailed look at key pieces of the code.

The interface. Brainstorming tools are a typical groupware application used by multiple participants to generate ideas about a particular subject. In this example, the brainstormer works by letting users add individual ideas to a visually shared list (Figure 4-12). There are three interface components in our brainstormer. An idea list collects ideas entered by all users and is visible to everyone. A text box lets users type in their own ideas, and an okay button lets a participant add an idea to the shared list.

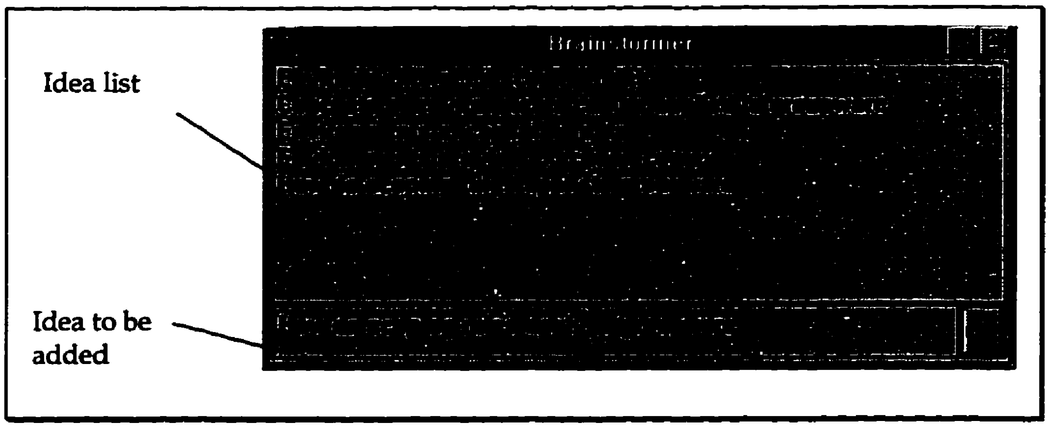

Figure 4-12. Example Brainstormer application.

The general programming model is composed of two components. The first component is the organization of the data, including determining what data is shared. The second component is how the model is linked to the view, so that when the shared data is modified, all of the displays are updated.

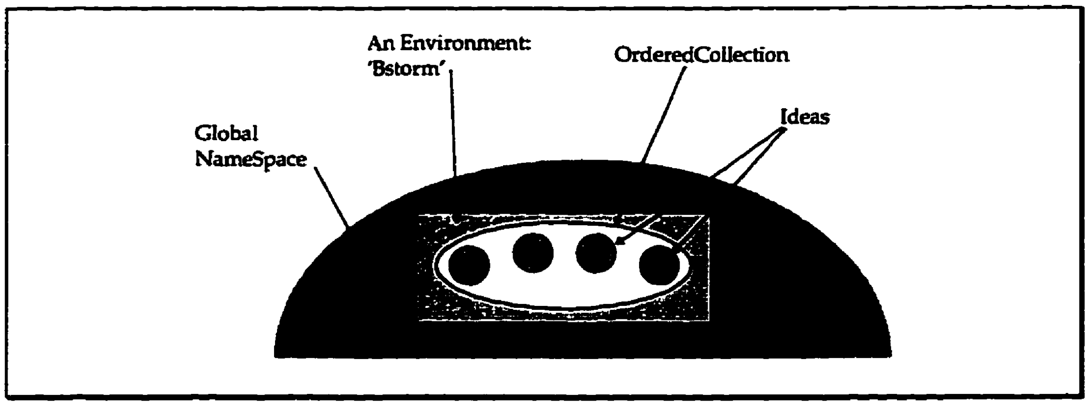

Figure 4-13. The global name space contains the 'Bstorm' environment which contains an OrderedCollection of Ideas.

Ideas in the brainstormer are strings entered by the user. The ideas are stored in an OrderedCollection (or linked list), which will be distributed between sites as a replicated object. The brainstormer uses an environment (called 'Bstorm') which is published to the global namespace. Figure 4-13 shows the relationship between these objects. The global namespace contains the environment which contains the OrderedCollection of ideas.

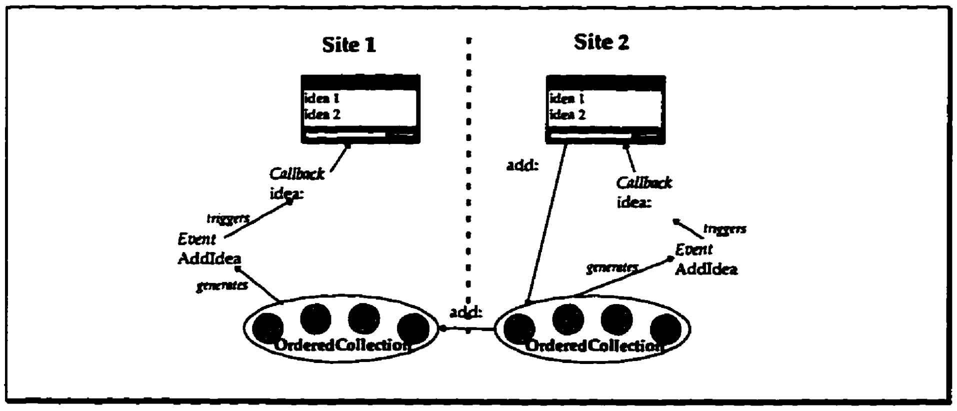

Figure 4-14. When an idea is added to an ordered collection, the AddIdea event is generated causing the call-backs associated with it to be executed.

Linking the model to the view is relatively easy. Whenever an idea is added to the OrderedCollection (the model) we will generate an event, which triggers a call-back in the interface (the view). The call-back will cause the interface to update its list of ideas. In Figure 4-14, we can see that when the *#add:* method is invoked at Site 2 to add an idea to the collection, it generates an event at both sites (called AddIdea) which triggers the call-back named *#idea:* which updates the local displays.

Starting up a simple Brainstormer. When groupware applications are started there are often initialization procedures that must run at a single site before users can connect to the application. Once the application has been started, special code must then be run on each site to connect the user to the other participants in the conference. This code would normally be executed by a special session management interface (Roseman & Greenberg, 1994), that allows clients to create and connect to conferences. However, we include it here for completeness.

The initialization of the brainstormer requires setting up an environment to hold the shared data. The *GroupEnvironment* is created in line 1 of Figure 4-15, with a distribution strategy of centralized. The environment is published in the *ObjectDirectory* (our global namespace) under the name, 'Bstorm' as shown in line 2 of Figure 4-15.

```
1  theEnvironment := GroupEnvironment new centralized.
2  ObjectDirectory addObject: theEnvironment named: 'Bstorm'.
```

Figure 4-15. Code to initialize the data structures for the brainstormer.

When a user wishes to join the brainstorming conference, they must start up the view (the brainstormer class) and connect it to the environment 'Bstorm', which contains the shared data. The brainstormer gets a reference to the environment by looking up its name in the *ObjectDirectory* as shown in line 1 of Figure 4-16. The user then creates a new Brainstormer window by sending the message *#openInEnvironment:* to the Brainstormer class, where the parameter specifies the environment.

```
1  bstormEnv := (ObjectDirectory objectNamed: 'Bstorm').
2  Brainstormer openInEnvironment: bstormEnv.
```

Figure 4-16. Code to start the brainstormer at a particular site.

The implementation. The previous code shows how the brainstormer application is started up by a user. We now look at the internal workings of the Brainstormer class to see how the start-up code (*#openInEnvironment:*) actually executes. Figure 4-17 only

```
openInEnvironment: anEnvironment
        |ideas|
        "STEP 1 Check to see if there are existing conferences"
        (anEnvironment globalAt: 'Ideas') isNil "Check for existence"
            ifTrue: [
                    "STEP 2 Set up the OrderedCollection of Ideas"
                    ideas := (OrderedCollection new replicated).
                    anEnvironment
                            globalAt: 'Ideas'
                            put: ideas.
                    "STEP 3. Create the event"
                    (anEnvironment globalAt: 'Ideas') "Create the event"
                            createPostGroupwareCallback: #AddIdea
                            onMessage: #add:].
        "STEP 4. Add the callback"
        ideas := (anEnvironment globalAt: 'Ideas').
        ideas "register a call-back"
            addGroupwareCallback: #AddIdea
            receiver: self
            selector: #idea:clientData:callData:
            clientData: ''.
        ... set up window ...

idea: anIdea clientData: ignore callData: callData
        list addItem: (callData at: 1) position: 0.

okayButton: w clientData: ignore1 callData: ignore2
        ideas add: (text getString)

close: widget clientData: ignore callData: data
        ideas
            removeGroupwareCallback: #AddIdea
            receiver: self
            selector: #idea:clientData:callData:
```

Figure 4-17. Code to set-up a shared brainstormer with replicated data.

shows those lines of the application that are related to the sharing of data, and for the sake of brevity we have excluded the code used to set up the window and display it.

In the *#openInEnvironment:* method (the first one called) shown in Figure 4-17, there are four distinct steps to setting up the application: 1) check to see if there is an existing conference; 2) if necessary set up the OrderedCollection of ideas; 3) create the appropriate event for when ideas are added and; 4) register the appropriate call-backs for keeping the list of ideas up to date. Step 1 checks to see if there is an existing conference. The variable 'Ideas' is looked up in the environment (line 4). If 'Ideas' is not found then nil will be returned and the idea collection will have to be created. Step 2 shows how it is created. An OrderedCollection of ideas is created using by sending the class the message *#new*. The returned instance is then made into a replicated object by sending it the message *#replicated* (line 7). The collection is then added to the environment (lines 8 - 10). In step 3, an event called *AddIdea* is associated with the *#add:* method of the OrderedCollection (lines 12-14). This event will be fired whenever a new idea is added. Finally, step 4 links the view (the window, which is self) to the event *AddIdea* (line 17-21). When the event is triggered the window will receive the call-back *#idea:clientData:callData:*.

The rest of the brainstormer is relatively straight forward. The call-back *#idea:clientData:callData:* is called whenever a new idea is added to the collection of ideas and it updates the idea list pane by adding the new idea to the bottom of the list, as shown in lines 24-25. The pressing of the OKAY button has been connected to the call-back *#okayButton:clientData:callData:* (lines 27-28) which adds the string contained in the text pane to the underlying list of ideas. Finally the close method (lines 30-34) removes the call-back that was registered, allowing garbage collection of the replicated object when it is no longer required.

One of our claims about GEN is that application developers can change the data distribution strategy. This is simple in GEN, as the only change required is to specify the type of sharing when the OrderedCollection of ideas is created. That is, line 7 in Figure 4-17 is changed to *ideas := (OrderedCollection new centralized)* and the data architecture is automatically changed.

4.6 Summary

We have presented the programmer's interface to GEN that provides the similar functionality as runtime architectures of other groupware toolkits. In addition, we show

how the developer can specify the way in which the data is distributed. An example demonstrated how the various components of the architecture are combined and coded in a groupware application.

5. GEN's Meta-Interface

The previous chapter described the programmer interface for GEN. This chapter presents GEN's meta-interface and how it gives developers control over data distribution and concurrency control mechanisms.

The goal of a meta-interface is to define a family of strategies that developers can use to create specialized implementations of the toolkit. The choice of which components to expose and how to expose them determines the family of behaviours that can be supported by the meta-interface. Although the meta-interface will support a broader range of behaviours than a conventional toolkit, it will not support all possible behaviours (Rao, 1993). The current implementation of GEN's meta-interface only supports modification of data distribution and concurrency control.

This chapter describes the meta-interface. It examines how the requirements of scope control, conceptual separation and incrementality are applied to the meta-interface. It then develops a design for the meta-interface that lets developers manage both data distribution and concurrency control. It continues by discussing how an implementation strategy called *wrappers* provides the flexibility required for building the meta-interface. Finally, the chapter delves into how a special class of wrappers are used to specify the API for the meta-interface.

5.1 Requirement for GEN's Meta-Interface

In this section we revisit the general principles of scope control, conceptual separation, and incrementality. We will see that these principles differentiate the meta-interface from a simple inheritance scheme by clearly separating how the changes the developer makes in the meta-interface layer will affect the toolkit. We examine how these principles are applied in the GEN meta-interface.

Scope Control limits the effect of a change in the meta-interface to a particular set of objects. There are three possibilities for the effect of a scope change on objects within an application: the entire application, a class of objects, or individual instances of objects. First, the entire application may be affected by the change to the meta-interface. For example, if the programmer changed the toolkit to use replicated objects, all objects in the application would be replicated, and there would be no possibility of having centralized objects within it. Although this solution may be acceptable for some applications, there are situations where the programmer needs to control the distribution of data based on the particular needs of an individual piece of data.

Second, the scope of a change can affect only a class. For example, the developer could declare a class of objects as having a particular distribution and concurrency scheme. If the programmer changed the *OrderedCollection* class to be replicated, then they can only create a centralized OrderedCollection by creating a new subclass. Each additional type of sharing would require the application developer to declare a new sub-class. If many different types of sharing were desired for a single class, there may be an explosion in the number of sub-classes.

The final possibility is that the scope of a change applies to a particular instance of an object. Here, two instances of OrderedCollection can have two different sharing mechanisms in the same application, so that one can be centralized and another replicated. This requires no additional coding and is the approach taken by GEN.

Conceptual Separation requires that the components of data distribution and concurrency control be separated so that the developer can deal with their implementation individually. In practice this may be difficult because there are interdependencies between components. In GEN we would like the methods that change the concurrency model and the distribution model to be distinct. However, a programmer modifying either the concurrency control or distribution mechanism must be aware of how they interact, since they are not entirely independent. For example, in replicated architectures the application developer must also maintain the concurrency mechanism when the object is distributed to a new site. If the concurrency mechanism is distributed differently than the object, the developer must also handle the distribution of the lock component separately.

In GEN, the difficulties in separating the concurrency control mechanism and the distribution mechanism mean that we can only present a limited form of conceptual separation. In this case when a developer defines a new concurrency mechanism they must also define how the concurrency mechanism is distributed.

Incrementality means that developers can build on top of existing implementations. GEN provides incrementality through the use of a separate class hierarchy, which specifies the sharing mechanism separately from the implementation of the objects to be shared. Application developers can then subclass existing sharing mechanisms to implement new ones, reusing existing strategies to help create new sharing strategies. For example, to create a migrating strategy, we will show in Chapter 7 how the replicated sharing strategy is modified to move objects, rather than copy objects around the network.

5.2 Design of the Meta-Interface

The meta-interface has to present a programming model to developers that allows a range of behaviours to be specified, but that does not inundate the developer with low level details (Rao, 1993). For example, a poor meta-interface could provide the developer with a sockets implementation and a way to convert objects into streams. While developers could use this to build a range of new behaviours, it would be a difficult and tedious task. Conversely, the meta-interface could provide a limited set of default implementations that the application developer could select and use in their application. However, if the range of behaviours supplied did not meet their needs, developers would revert to the strategies of coding hematomas and coding between the lines (as discussed in Chapter 3). The model presented by GEN needs to be both relatively simple to use and flexible enough to define a broad range of behaviour.

In this section we look at the model used in GEN's meta-interface to let developers modify data distribution mechanisms (Section 5.2.1) and concurrency control mechanisms (Section 5.2.2).

5.2.1 Control Model for Data Distribution

There are different approaches to give developers control over data distribution. In distributed computation systems, such as Emerald (Hutchinson et al., 1987), the focus is on using a set of distributed resources efficiently to solve a complex problem that requires significant computing power. These systems give the developer control over where computation is done, in an attempt to minimize resource use and computation time.

In groupware systems, the focus is different. These systems must keep the information shared by participants consistent, while at the same time giving rapid feedback to the local user (Greenberg and Marwood, 1994). The usual bottleneck in groupware applications is the time it takes to transmit changes between sites. Chapter 2 showed how particular runtime architectures change how the data is represented at different sites, and how messages are sent between the sites (e.g. centralized and replicated objects) to overcome these bottlenecks under specific circumstances. By modifying how the data is represented and how messages are sent between various copies, developers can minimize transmission time.

Rather than giving developers explicit control over where an object is located (and hence where computation is done on it), our implementation gives developers the ability to control the representation of the object at the local site, and the ability to control how messages are routed between the different sites.

Object Representation and Content Control. The developer needs to be able to control the representation of an object. Representing an object as a proxy, for example, allows them to create proxies on new sites so that the object can be remotely referenced (a technique used in centralized architectures). Figure 5-1 illustrates an example. We see that an object copied from Site 1 to Site 2 is mutated to become a proxy, completely changing the representation of the original object.

Conversely, this control over an object's representation can be used to create a copy of the object on the new site, allowing the object to be locally referenced (a technique used in replicated objects). The way the contents of an object are transferred is equally important. For example, the contents of an object may reference a global variable in a local process. In this case there are several ways this reference can be transferred between sites. First, it may reference the global variable remotely, which will require several message sends. Second, it may copy the global variable to the new site, which, if it were something like the local sites name (i.e. fully qualified domain name), may be incorrect. Third, the global reference may be made to point to the global variable as it is defined by the new site. Figure 5-1 shows how the contents of an object are changed when it is copied between Site 1 and Site 3. In this case the original object on Site 1 contains a reference to the global variable X. When the object is copied to Site 3, it is changed so that the reference points to the global variable X on the new site.

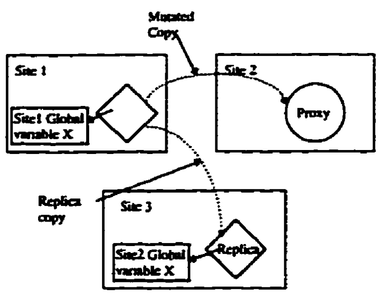

Figure 5-1. Control over the contents/ representation of the object.

Once an object has multiple representations of itself in the network, the second problem then becomes how messages sent to one object is forwarded (or routed) to other copies of the object that exist around the network.

Message Routing Control. GEN lets developers control how messages are forwarded around the network (or routed). Some messages may be executed locally (for example, if they simply read the state of an object), others may have to be broadcast (for example, if they change the state of an object), while still others may simply be forwarded to a single site (for example, centralized objects).

In implementing replicated objects, messages have to be broadcast to multiple sites. Figure 5-2a shows how a message sent to an object at Site 1 is then broadcast to its replicated counter parts at Sites 2 & 3. In a centralized object however, the proxy sends the message only to the original object and the other proxies are not involved in the process. Figure 5-2b illustrates how the proxy in Site 1 forwards its message to the original object at Site 2 and not to the other proxy at Site 3.

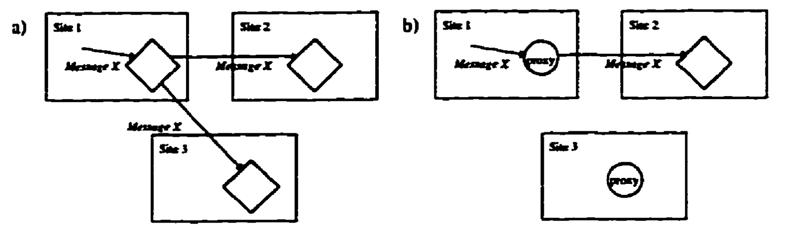

Figure 5-2. Control over message routing, examples of centralized and replicated schemes.

5.2.2 Mechanisms for Concurrency Control

The second kind of control that GEN's meta-interface provides is the ability to modify the concurrency mechanism. For example, concurrency control is required in replicated objects because there is a potential conflict when two messages are sent simultaneously to the same object. This is illustrated by Figure 5-3a. Here we can see that both Site 1 and Site 3 are sending potentially conflicting messages to Site 2. These messages may perform incompatible operations, such as one site may delete an object while the other site resizes it.

Control over these mechanisms is required because there are many different forms of concurrency control (e.g. optimistic locking, pessimistic locking). The type used depends on the requirements of the application (Greenberg & Marwood, 1994). Concurrency control mechanisms in GEN is implemented at the message level. Before a message is executed by an object, and then again after a message has completed executing at all sites, the developer can specify arbitrary actions to control concurrency. For example, consider the replicated locking object in Figure 5-3b that implements a centralized locking strategy. Both sites receive a message and try to obtain a lock before forwarding it. Only Site 3 receives the lock, and its message Y is sent to Site 1 and 2. Once the message has finished executing at all sites, the lock will be released. Site 1 would then receive the lock allowing Message X to be broadcast (not shown).

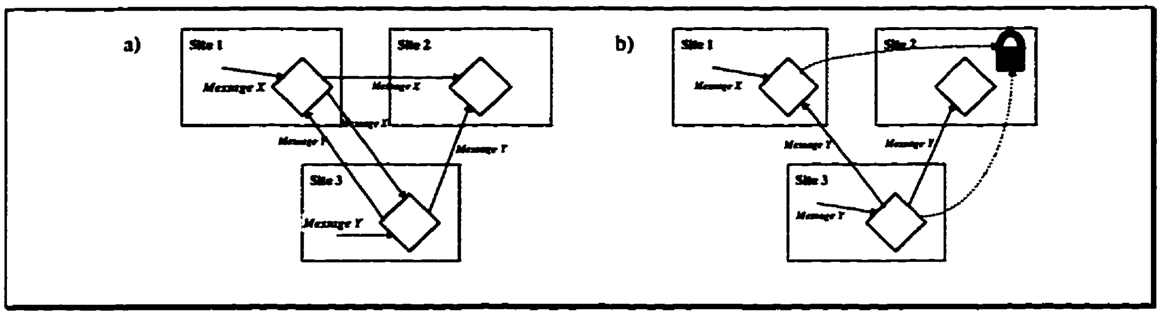

Figure 5-3 . The need for concurrency control is obvious when two sites send potentially conflicting messages to the same site.

5.2.3 Summary

The models in the meta-interface are designed to allow the developer to control concurrency and distribution. The black box handles the details of inter-process communication, marshalling of objects and forwarding of messages. By abstracting these components away, GEN simplifies the problems of building new sharing strategies, while providing flexibility.

5.3 An Implementation Strategy: The Wrapper Model

In this section, we consider how the properties of wrappers, also known as decorators, allow a programmer to modify the behaviour of an object (Gamma et al, 1995). A wrapper is an object that allows the developer to attach additional behaviours to another object dynamically in order to extend its functionality. A wrapper encapsulates another object inside itself by selectively passing messages through to the original object, and implementing new messages that the object will understand. As we shall see, wrappers are a good mechanism for constructing the meta-interface in Smalltalk and GEN.

5.3.1 The Choice of Wrappers

We believe wrappers are well suited to implementing meta-interfaces because of two properties: the ability to add new behaviours to objects transparently and dynamically;

and the ability to create these new behaviours without changing the implementation of the object.

The ability to add new behaviours transparently and dynamically implies that the original object's interface does not change. That is, the set of messages that an object will understand and respond to does not change once the wrapper is added. Thus, shared objects will have the same interface as non-shared objects of the same class. This means that the distribution and concurrency mechanisms implemented by GEN will not affect the original behaviour of the object.

The dynamic nature of wrappers lets the developer delay decisions about the addition of new behaviours until runtime. For example, the application developer may make the decision about distribution behaviour (e.g. replicated or centralized) and the concurrency control mechanism (e.g. optimistic, pessimistic) based on runtime factors such as network speed or available memory.

The ability to create new behaviours for an instance without changing the class implementation reduces the number of subclasses that developers must manage. Without wrappers, one could provide the additional functionality on a per class basis. However, each combination of class, data distribution scheme, and concurrency control scheme would require the creation of a new class. This is tedious, complex, and difficult to maintain. With wrappers, the change is made in the wrapper hierarchy. This new form of wrapper can then be applied to an instance of any class.

Adding new behaviours to an instance of an object allows different instances of the same class to have different behaviours. For example, a developer can create two rectangles, and can choose to place a centralized wrapper around one, and a replicated wrapper around the other. As well, because the GEN wrapper is independent of the object contained, a single class of wrapper can be applied both to, say, Rectangle objects as well as Dictionaries, or even entire expert systems.

The implementation of the wrapper classes are completely separated from the implementation of the object classes that they contain. Each shared object has its own unique wrapper which allows the developer to have several objects of the same class with different wrappers and hence different distribution and concurrency schemes. The wrapper itself may even be in a different inheritance hierarchy than the objects that it wraps. This feature allows us to apply a particular wrapper implementation (such as replicated, or replicated locking) to unrelated objects such as Rectangles and OrderedCollections.

The GEN wrapper classes are composed of two separate components. First, a message handling component intercepts messages bound for the contained object. GEN uses that interception to perform notification, concurrency control and message routing. A second completely separate implementation handles messages sent by the distributed object layer to distribute the wrapper and the object. By modifying these methods, the developer can control the contents and representation of the object when it is copied to new sites. We begin by looking at the model for the message interception layer.

5.3.2 A Wrapper Model for Message Interception

A wrapper allows developers to dynamically add a thin layer of functionality to an object, by intercepting messages destined for the original object. The wrapper intercepts messages in three stages:

1. Pre-message management. The wrapper can execute arbitrary code immediately upon intercepting the message.
2. Message handling. The wrapper can decide how to pass the original message and its arguments to the contained object.
3. Post-message management. The wrapper can execute arbitrary code immediately after receiving the answer from the contained object, and can decide how to return the result.

In GEN, the wrapper uses these three steps to implement the meta-interface for notification, concurrency control, and message routing. As shown in Figure 5-4, messages destined for a particular object are intercepted by the wrapper. The first stage generates initial notifications and events (if required), and initializes the specified concurrency control method. Stage two decides how to distribute and route the message to the object, and how to manage the reply. Finally, stage three manages any post-operative concurrency control functions, and generates final notification and events (if required).

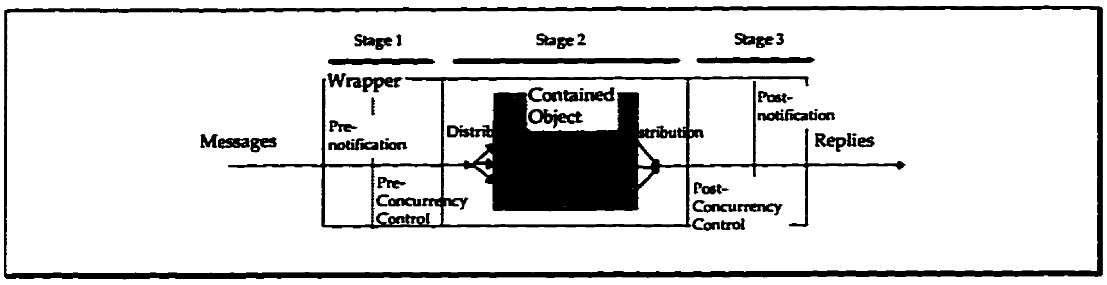

Figure 5-4. A wrapper around an original object intercepts messages destined for it. The wrapper contains the controls for distributing the object and maintaining consistency.

A note about notification. As mentioned in Chapter 4, notification is an important part of any groupware toolkit. In this discussion we will see the hooks for it when we discuss message interception in detail. However, because we are only interested in distribution and concurrency control, we have not exposed the model in the meta-interface. Notification is an important area to expose to the application developer, however because of the variety of ways (Hill, 1992) and potential complexity of implementing them it is a research are beyond the scope of this thesis.

The message interception layer allows us to control messages destined for the contained object. The second function of the wrapper is to respond to messages used to distribute the object which allows control over the representation and contents of that object.

5.3.3 A Wrapper Model for Representation and Contents Control

The wrapper has a second, completely separate, mechanism which is used to control the representation and contents of the object when it is distributed across the network. When an object is about to be distributed to a new site, the system sends messages to that object to convert it to a bytestream (called marshalling). The wrapper responds to these messages and provides hooks so that the developer can substitute a modified object which will be copied to the new site, in place of the original object. In Chapter 4, we described how the marshalling of a distributed object directly converted it into a bytestream. For replicated and centralized objects this was actually handled by the wrapper, which deconstructs and reconstructs the object. Figure 5-5 shows how an original object A is transformed into A' before it is again transformed into a bytestream. The bytestream is transmitted to the other site and converted back into the object A'. It can then again be changed during this reconstruction phase into a new object called A''.

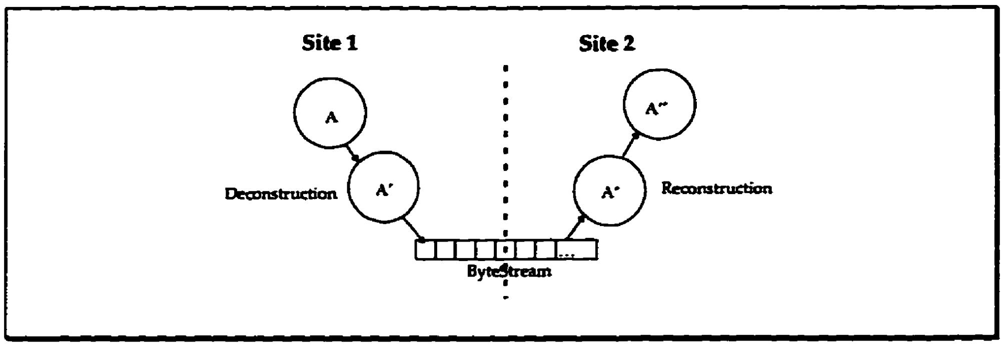

Figure 5-5. The wrapper intercepts the deconstruction and reconstruction of the object, when it is being marshalled to allow customizable behaviours.

In summary, the use of a wrapper provides a way to meet the goals of scope control and incrementality mentioned in Section 5.1. The fact that a wrapper is applied to a particular instance of an object allows different concurrency control and data distribution behaviours to be applied to different instances of the same class, giving GEN the desired level of scope control. Furthermore, the wrappers form a class hierarchy, which the developer can specialize and use to incrementally modify the meta-interface.

5.4 Putting Wrappers Around Objects

The first step to specifying how an object is shared, is to place a wrapper around the object. As we saw in the previous chapter, the methods *#replicated* and *#centralized* were called when specifying the sharing strategy for an object. These were helper functions for the wrapper object. The wrapper object is created by sending the message *#object:* (specified in Table 5-5) to the wrapper class, where the parameter is the object to be contained in the wrapper. For example, the replicated wrapper class is called

ReplicatedElement. To instantiate a replicated wrapper around an original object, the developer would execute the following piece of code:

wrapperedObject := ReplicatedElement object: originalObject.

The object returned by this call, *wrapperedObject*, is a new wrapper containing the *originalObject.*

Protocol	Effect
wrapperedObject := aWrapperClass object: anObject	Returns a new object that can be shared according to the aWrapperClass scheme

Table 5-5. Protocol for wrapping objects

Additionally, new wrapper classes are going to have additional instance variables to manage and initialize. The protocol *#initializeGWElement* (Table 5-6) provides the application developer the ability to specify how particular instance variables are initialized.

Protocol	Effect
aWrapperClass initializeGWElement	Initializes the instance of the wrapper class

Table 5-6. Initialization protocols.

5.5 A Wrapper API for Message Interception

The root of the GEN wrapper class hierarchy is the GroupwareElement class. This class allows single user objects to be transformed into groupware objects by performing message interception. It also adds the hooks to handle pre- and post-notification, pre- and post-concurrency control functions and message routing. The replicated, replicated locking and centralized schemes, described in Chapter 4, inherit from this abstract class to provide their specific behaviours (their implementations are detailed in Chapter 6).

A message to a GroupwareElement wrapper goes through the steps illustrated in Figure 5-4. As already mentioned, the wrapper intercepts generic messages bound for the original object, passes it through a pre-notification and pre-consistency layer. It then goes through a routing layer which distributes or forwards the message to other objects (e.g. replicas and proxies) which may include sending the message to the original object. Finally, after the message has been routed, the consistency layer is called again allowing the release of any resources that it required, and any final notification is performed.

5.5.1 Intercepting Messages

Our discussion about wrapper implementation begins by describing how GEN can apply a wrapper to any object of any class through Smalltalk's *#doesNotUnderstand:* protocol (Gamma et al., 1995). In Smalltalk, whenever an object receives a message that its class does not recognize, the object will automatically dispatch a *#doesNotUnderstand: aMessage* to the object. This normally brings up an error notification or debugging dialog.

This protocol can be overridden to implement a wrapper layer to objects. First, wrapper classes are constructed so they do not implement any messages, nor do they have any super classes. The consequence is that they do not understand any messages. When a wrapper receives a message it does not understand (which will be all of them), Smalltalk calls the *#doesNotUnderstand: aMessage*, where the original message is passed as an argument. By modifying the *#doesNotUnderstand:* method in the wrapper object, the developer can intercept and manipulate the message sent to the contained object. For example, Figure 5-6 shows a simple example of how the method *#doesNotUnderstand:* is altered so that it just forwards the message on to the contained object (i.e., so the wrapper has no effect).

```
doesNotUnderstand: aMessage
    ^containedObject
        perform: aMessage selector
        withArguments: aMessage arguments
```

Figure 5-6. Message interception through the #doesNotUnderstand: implementation. In this case the object simply forwards the message to the contained object.

Of course, GEN requires a more complex implementation of the *#doesNotUnderstand:* method to perform notification, concurrency control and distribution. Figure 5-7 shows that there are actually five stages that the message goes through in a wrapper. Before the message is distributed, both pre-notification and pre-consistency phases occur (lines 3 & 4). The *#handleMessage:* method is then executed (line 5), which is responsible for the distribution of the message to all the replicas or possibly a centralized copy. After the message has completed, the post-consistency and post-notification phases occur (lines 6 & 7).

```
1 doesNotUnderstand: aMessage
2       |result|
3           self handlePreNotification: aMessage.
4           self handlePreConsistency: aMessage.
5           result := self handleMessage: aMessage.
6           self handlePostConsistency: aMessage.
7           self handlePostNotification: aMessage.
8       ^result
```

Figure 5-7. Open implementation of message interception.

The root GroupwareElement class just provides these stages as hooks: there is actually no implementation behind them. It is up to other classes that inherit from GroupwareElement to implement particular approaches to notification, consistency, and routing. Additionally, by overriding the *#doesNotUnderstand:* method in new subclasses of the wrapper, the developer can add new behaviours at message interception time, and even remove the default components of notification and/or consistency if they are not required.

5.5.2 Controlling Concurrency Mechanisms

Wrappers place two hooks for concurrency checks immediately before the message is distributed to the object, and immediately after the distribution has completed as shown in Figure 5-8. As mentioned before, distribution in GEN is accomplished by broadcasting or forwarding messages between objects at different sites. By providing a wrapper layer to control the concurrency of these messages, concurrency within the objects can be managed.

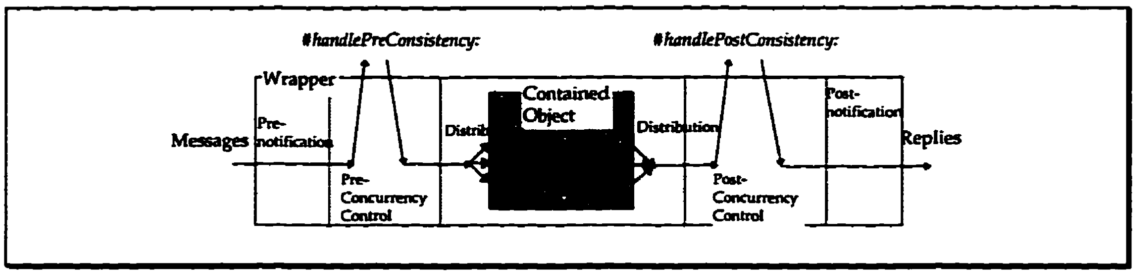

Figure 5-8. The pre and post notification stages.

Concurrency is controlled implicitly (Stroud & Wu, 1995). This means the wrapper is responsible for maintaining the consistency of the object, without requiring changes to the object itself. For example, we can consider a wrapper which ensures messages are executed in mutually exclusive fashion. An *OrderedCollection* may be wrapped by one of these. If two objects are added to the *OrderedCollection* at the same time from different sites, the first add received must be executed at all sites before the second one begins. The algorithm for ensuring mutually exclusive execution of the methods in this case will be implemented in the wrapper, and will not require any changes to the implementation of the *OrderedCollection*. Also, because the wrapper handles the mutual exclusion of the messages, application developers will not have to worry about acquiring a separate lock before sending the message *#add:* to the *OrderedCollection*. The object itself maintains its own concurrency, and the application developer does not deal with it other than to select a wrapper which gives them the desired level of concurrency control.

Application developers specify the concurrency control in the wrapper using the protocols of *#handlePreConsistency:* and *#handlePostConsistency:*, shown in Table 5-7. By default, the pre-consistency and post-consistency methods do nothing. Consistency, as mentioned before, is often not required for some style of groupware applications (Greenberg & Marwood, 1993). However, by modifying these methods the developer can create new consistency schemes. For example, in pessimistic locking schemes the application developer will use the *#handlePreConsistency:* method to acquire the lock. The *#handlePostConsistency:* method will be used to release the lock and is called only after the message has been delivered to and executed by each remote site. In Chapter 6, we shall show how a modified scheme can be used to implement a form of optimistic locking.

Protocol	Effect
aSharedObject handlePreConsistency: aMessage	Override this method to define new ways for handling concurrency control, before a message is sent.
aSharedObject handlePostConsistency: aMessage	Override this method to define new ways for handling concurrency control, after a message is sent.

Table 5-7. Protocols for overriding consistency methods.

5.5.3 Controlling the Routing of Messages

Once the pre-consistency stage is completed, the interception of messages is used to implement controllable routing of messages. This allows developers to broadcast messages to other replicas, forward them to centralized objects, or just send them to the contained object. Figure 5-9 shows the steps a message will pass through when being routed by a wrapper. After completing the first consistency stage, the message being sent to the object is intercepted by the *#handleMessage:* method. This method first calls *#distributeMessage:* to forward the message to remote copies and then calls *#performMessage:* to send the message to the local copy. The *#distributeMessage*: forwards messages to the remote copies of the objects, by calling them with *#handleRemoteMessage:*. The wrapper uses proxy objects (from the distributed object implementation) to communicate with remote replicas, as will be discussed further in Chapter 6, where the default implementation of replicated objects in Chapter 6 is presented.

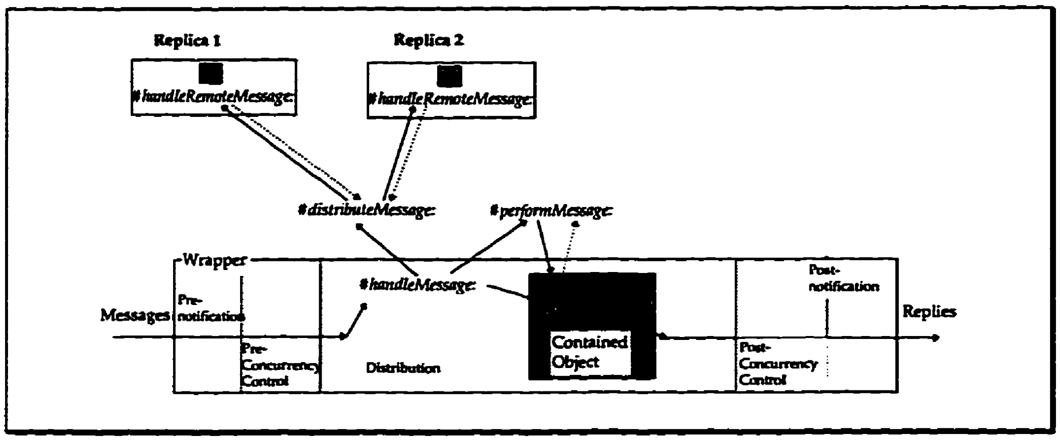

Figure 5-9. The components of message routing in the wrapper.

Table 5-8 shows the three methods that can be modified to allow different distribution techniques. The default implementation's first one, ***#distributeMessage:***, for example, will broadcast messages in the case of a replicated object, or will forward the message on to the actual site (via a proxy) in the case of a centralized object. Similarly,

Protocol	**Effect**
aSharedObject distributeMessage: aMessage	A modifiable method that allows developers to change the way messages are distributed to local an remote objects.
aSharedObject handleRemoteMessage: aMessage	A modifiable method for changing the way in which messages sent from other sites are handled.
aSharedObject performMessage: aMessage	A modifiable method for changing the way in which messages from the local site are changed.

Table 5-8. Protocols for handling message distribution

#performMessage: controls whether or not the message is sent to the local object. For example, in the replicated object implementation, ***#performMessage:*** will always send the message to the local object. For centralized objects, ***#performMessage:*** will only send the message to the object if it is located on the local machine. The final method, ***#handleRemoteMessage:***, allows the developer to distinguish between messages forwarded by another replica from those sent by another object. For example, the *#handleRemoteMessage:* is used in replicated objects to ensure that the message forwarded to them is not then again subjected to concurrency control.

5.6 The Wrapper API for Controlling Object Representation and Contents

The next component of the meta-interface looks at the representation and contents of the object. There are situations where objects that are copied between sites should not be identical. For example, a global value such as the name of the local site will have to be changed when the object is copied between sites. Also, proxy objects are not copies of the object, just remote references. As such, they use a completely different representation for the object. The meta-interface provides the capability of changing the representation and contents of an object while it is being transported to a new site.

This is done by allowing the developer to substitute a new object both when the object is deconstructed on the sending site, and then again when the object is reconstructed on the receiving site. This is accomplished by the methods *#deconstructObject* and *#reconstructObject*, which are sent by the distributed object system to the wrapper just before and just after the object is sent to the remote site. These two methods return an object that should be substituted for the current one.

The *#deconstructObject* method is called before the object is copied to the new site. As such, it may be used to break links to local references. For example, in Figure 5-10, we can see that the deconstruction of the object A creates a copy of the object called A' which has removed the reference to the site name 'Site 1'. Once the deconstruction phase has been completed the object is then copied to the new site. An instance of that object is created on the new site, which is then sent the *#reconstructObject* message. In Figure 5-10 we can see that a new object is substituted for A' called A'', and the link to the new site's name 'Site 2' has been created.

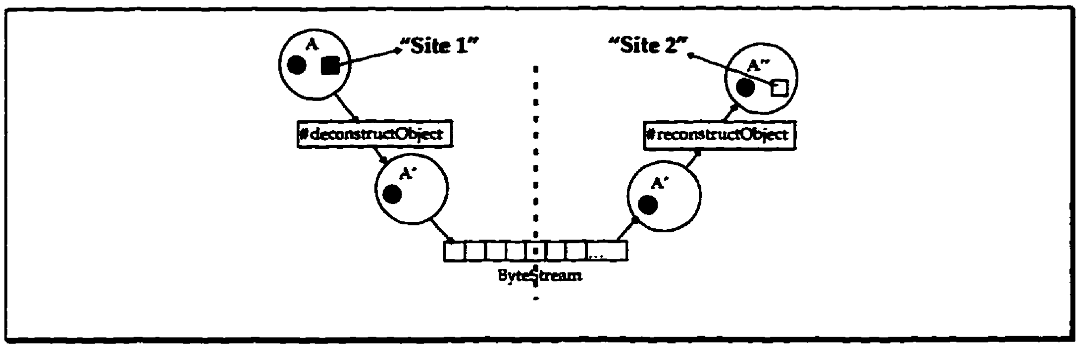

Figure 5-10. Deconstruction is used to remove a reference to a global variable while reconstruction is used to rebuild that link.

The deconstruction and reconstruction model modify both the wrapper and the contained object in the same piece of code. The wrapper may need to be deconstructed in a special way to handle changes in the locking mechanism or other internal data structure (we shall see this in Chapter 6, when we discuss the replicated wrapper). The object, we have seen, may need special deconstruction to handle changes to its contents, such as a reference to a global object.

The deconstruction and reconstruction phases in GEN are further refined to supply default implementations that separate the deconstruction of the wrapper from the deconstruction of the contained object, allowing the developer to specify the modifications of these components at an instance level. In Table 5-9, we detail the protocols that separate the deconstruction and reconstruction of the contained object from the deconstruction and reconstruction of the wrapper. These protocols take a *Block* (an executable piece of code) which is specified at runtime, allowing the developer to provide a custom piece of code. This technique removes the need to define a new wrapper class each time a developer wishes to modify the way a particular class of object is distributed. Different instances of a wrapper class can deconstruct and reconstruct their objects in different ways. The protocols *#objectDeconstructionBlock:* and *#objectReconstructionBlock:* control the deconstruction and reconstruction of the contained object. The protocols *#wrapperDeconstructionBlock:* and *#wrapperReconstructionBlock:* take *Blocks* which control the deconstruction and reconstruction of the wrapper.

Protocol	Effect
aSharedObject objectDeconstructionBlock: aBlock	aBlock takes 1 parameter (the contained object) and answers a mutation or copy of the contained object that is to be passed on.
aSharedObject objectReconstructionBlock: aBlock	aBlock takes 1 parameter (the mutated contained object) and answers a mutation of that object that is the new contained object.
aSharedObject wrapperDeconstructionBlock: aBlock	aBlock takes 1 parameter (the *wrapper* object) and answers a mutation or copy of the contained object that is to be passed on.
aSharedObject wrapperReconstructionBlock: aBlock	aBlock takes 1 parameter (the mutated *wrapper* object) and answers a mutation of that object that is the new wrapper object.

Table 5-9. Protocols for specifying instance specific behaviours for deconstructing and reconstructing the wrappers.

The meta-interface simplifies this process in one more step, by allowing the meta-interface developer to specify default implementations for each of these modification steps. The protocols *#defaultObjectDeconstructionBlock*, *#defaultObjectReconstruction-Block*, *#defaultWrapperDeconstructionBlock* and *#defaultWrapperReconstructionBlock* are defined in each class and define the default protocol for deconstructing and reconstructing the wrapper and the object.

5.6.1 Using Substitution to Control Distribution: Copies and Proxies

The ability to deconstruct and reconstruct an object can also be used to give the developer control over the way an object and its contained components are shared. For example, if a replicated wrapper contains a lock, it may not be desirable to replicate the lock when it is transmitted to the new site. Rather, it should be maintained as a remote reference to provide mutual exclusion between the different sites. By substituting a proxy, the developer can create a remote reference to the object.

In this case, GEN allows the developer to substitute a special object that dictates how the contained object is shared. These two objects are *RemoteProxys* and *RemoteCopys*, as shown in Table 5-10. The substitution of a *RemoteProxy* for an object forces a remote reference for the object when it is distributed, whereas substitution of a *RemoteCopy* forces the object to be copied.

Protocol	**Effect**
aProxyObject := RemoteProxy for: anObject	Creates a special object for the object anObject which will force a proxy parameter to be created for anObject.
aCopyableObject := RemoteCopy for: anObject	Creates a special object for the object anObject which will force a remote copy to be created for anObject.

Table 5-10. Protocols for remote object distribution.

For example, Figure 5-11a) shows how object A references object B. When object B is being copied across, a *RemoteProxy* is substituted, then when A is reconstructed on the other site, it contains a proxy to the original object B. Conversely, in Figure 5-11b) a *RemoteCopy* is substituted for object B, so that when A is reconstructed on the new site, a new copy of B is also created.

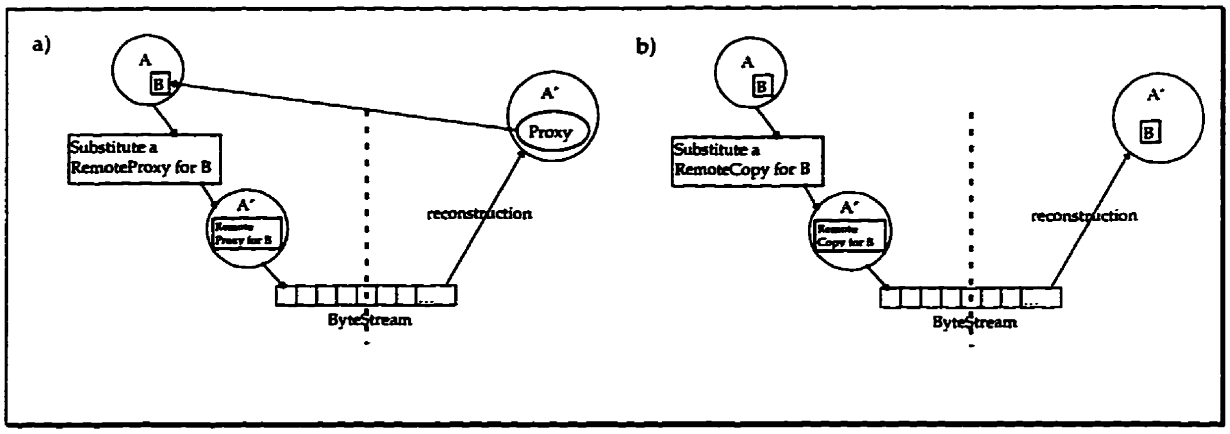

Figure 5-11a) Shows how a remote proxy substitution creates a remote reference. Figure 5-11b) Shows how a remote copy substitution creates a new copy.

To simplify matters GEN provides a method which sets, by default, whether an object is copied or remotely referenced. The object is sent the message *#isCopiedRemotely* which will answer true if the object is to be copied, false if it is to be remotely referenced. This method is implemented in the class, so the answer will apply to all instances of the class. Note that the *RemoteCopy* and *RemoteProxy* implementations override this default.

This is the last of the meta-interface that deals directly with giving the developer control over how objects are distributed and their concurrency mechanisms.

5.7 Summary

This chapter has presented four separate sections. The first examined how the principles of open implementation, namely scope control, conceptual separation and incrementality were applied to GEN's meta-interface. It was explained that conceptual separation was not entirely possible in GEN because of dependencies between distribution and concurrency control mechanisms.

The second section examined the model we use to let developers modify distribution and concurrency control mechanisms. To control how objects are distributed between sites, developers can modify both the contents and representation of the object. Additionally, developers are given control over how messages are routed between sites. Concurrency control, on the other hand, was implemented at the message level by giving developers the ability to implement concurrency mechanisms both before and after the object executes a message sent to it.

The third section introduced the notion of wrappers and how they could be used to modify the behaviour of objects. The fourth and final section discussed how wrappers were used to implement the meta-interface API that lets developers modify the mechanisms used to distribute the objects and control concurrency.

The following chapter will evaluate the meta-interface by illustrating how the replicated, replicated locking and centralized data sharing strategies are implemented. Chapter 7 will continue by showing how the meta-interface can be used to build new constructs, such as migrating objects, selective message broadcasts, and a form of optimistic locking.

6. Case Studies: The Default Implementations

The last component supplied with the GEN toolkit is the default implementation for the data sharing strategies. The default implementations are important for several reasons. First, they are necessary because they give developers, who do not want to concern themselves with the meta-interface, a usable implementation. Second, the default implementations are built on top of the meta-interface and serve as examples of how the meta-interface works. Third, and as we will see in the subsequent chapter, they can be incrementally modified, allowing developers to build on existing strategies to construct new ones. Finally, the default implementations serve as one way to evaluate the meta-interface, by showing how sharing strategies can be constructed using the meta-interface.

In the current version of GEN, the default implementation provides the developer with an API to replicated, replicated locking and centralized data sharing strategies (introduced in Chapter 4). These particular default implementations were chosen because they are typical of data distribution and concurrency strategies found in other groupware toolkits and likely form a reasonable set of data sharing strategies for groupware developers. Additionally, as we shall see in Chapter 7, these particular implementations can be built upon incrementally to provide new strategies, such as migration and a form of optimistic locking.

This chapter shows how the meta-interface is used to construct the default implementations. It begins with an implementation for replicated objects. It then shows how this implementation is modified to create replicated locking objects. The chapter concludes with the final default implementation of centralized objects.

6.1 Case Study #1: Replicated Objects

As we saw in Chapter 4, a replicated object has a copy of itself located at each site. A replicated object is useful because a local copy provides quick feedback for any local changes made to the object.

Replicated objects need special mechanisms for routing messages and managing replica creation. First, when there are multiple replicas of the object on different sites, a replica receiving a message must broadcast it to the other replicas in the system. This keeps all copies up to date. Second, when the object is distributed to a new site, there must be a mechanism for creating a replica at that site.

In this section we examine how the meta-interface modifies the routing of messages to perform the message broadcast, and how the wrapper's contents are modified when a new replica is created. The name of the replicated object wrapper class is *ReplicatedElement,* and its implementation specializes the *GroupwareElement* class (i.e., the basic object wrapper).

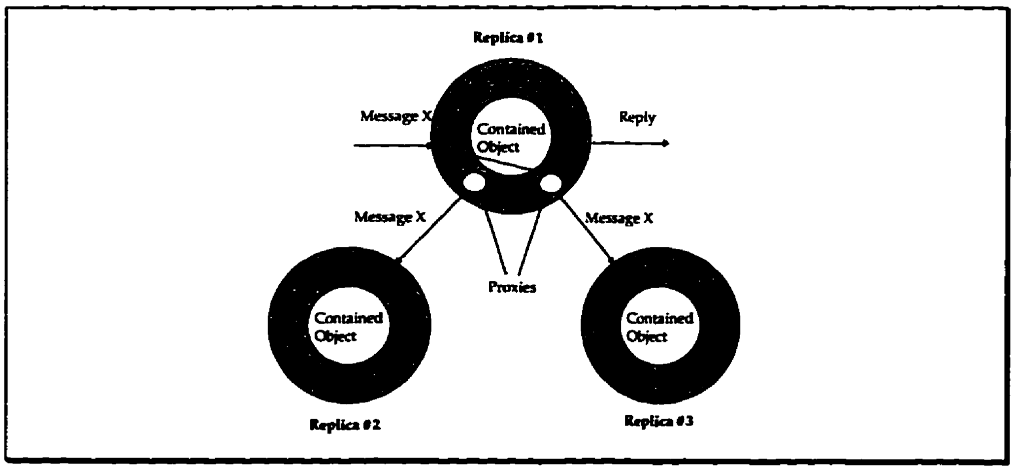

Figure 6-1. Synchronization in replicated objects through message forwarding.

6.1.1 Message Routing

In this section, we will discuss how the replicated object routes messages between the various replicas in the system. Since the replicated element must broadcast each message it receives to all the other replicas, it maintains a list of proxies which point to the wrappers for the other replicas. A replica will use this list (called *replicas*) to forward messages it receives to the other replicas. One replica sends a message to another remote replica, by forwarding the message through a local proxy for the remote replica. The local proxy will then forward the message. For example, in Figure 6-1 Replica #1 has received a message (Message X), which it sends to the proxies for Replica #1 and Replica #2. These proxies then forward the messages to the actual object on the remote sites using a special protocol (*#handleRemoteMessage:)* that ensures the message is not rebroadcast to the new sites (as discussed in 5.5.3).

In this case the *ReplicatedElement* must modify two components of the *GroupwareElement* class: the message routing scheme, for distributing the messages to the replicas; and the object representation, to keep the list of replicas up to date when a new replica is created.

Changes to the Message Routing Scheme. There are two changes to the way messages are distributed using the *GroupwareElement*. The first change modifies the ways messages are sent, through the *#distributeMessage:* protocol. The second modifies the way messages are received by the *#handleRemoteMessage:* protocol.

ReplicatedElement defines an implementation of *#distributeMessage:* which sends a message to each of the replicas by iterating over the list of *replicas* as shown in Figure 6-2 (lines 2 & 3). The iteration is performed in a separate thread (using *#fork*), to allow the current thread to continue processing while the message is distributed to the other sites.

```
1 distributeMessage: aMessage
2        [replicas do: [ : aReplica |
3            aReplica handleRemoteMessage: aMessage]] fork.
4
5 handleRemoteMessage: aMessage
6        self handlePreNotification: aMessage.
7        containedObject performMessage: aMessage.
8        self handlePostNotification: aMessage.
9        ^nil
```

Figure 6-2. Code for the distribution of messages to replicas.

The modifications to *#handleRemoteMessage:* speed up the reply process and handle local notification. To ensure that the message is not rebroadcast to all the other sites, *#handleRemoteMessage:* sends the message to the local object only, and does not call *#distributeMessage:* again (as discussed in Section 5.5.1). To speed up the reply process, the implementation returns the 'nil' object which is quick to transfer between sites (line 9 of Figure 6-2). The implementation does not need to return the original object because all replicas should answer the same value when sent a message, thus the implementation can use the result of the local *#performMessage:* and ignore the replies of the other replicas. The reply is an acknowledgement indicating that the message was received and executed at the local site successfully.

The second change to *#handleRemoteMessage:* involves the use of the pre- and post-notification methods in lines 6 and 8. In a replicated object, we make the local wrapper perform notification, to reduce the number of messages that must be broadcast. As mentioned in Chapter 5, we did not provide an open implementation for notification, which has side-effects when trying to implement the replicated object. The problem is that the underlying implementation does not automatically broadcast the notification events to all the sites. By adding the pre- and post-notification methods to *#handleRemoteMessage:*, the local site generates the message. However, this violates the principle of conceptual separation (see Chapter 3), since the developer must now modify the notification system when changing the message routing strategy.

6.1.2 Changing the Object's Representation.

A replicated object has a copy of the object at each site. When a new replica is created (for example, this occurs when a new site references the object), a copy of both the

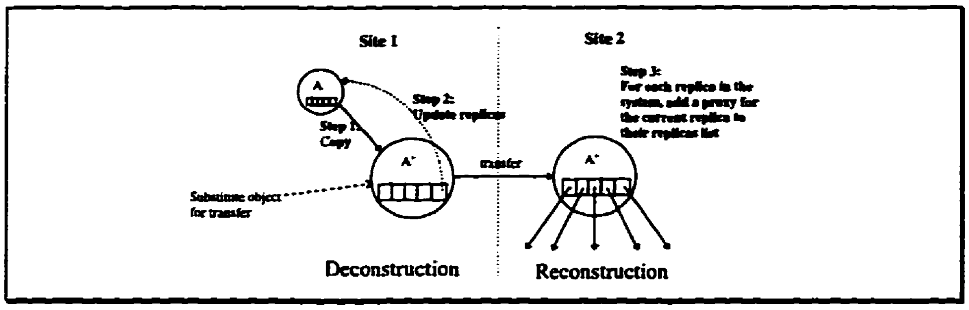
Site 1
Site 2
A
Step 1: Copy
Step 2: Update replicas
Step 3: For each replica in the system, add a proxy for the current replica to their replicas list
A'
transfer
A'
Substitute object for transfer
Deconstruction
Reconstruction

Figure 6-3. The steps in maintaining the list of replicas.

wrapper and the contained object must be passed to the remote site. To create this copy of the wrapper and the contained object, the method *#isCopiedRemotely* answers true, forcing the distributed objects layer to copy the object when it is distributed. By default, this makes an exact copy of the object. However, we need to modify the *replicas* list, as described below.

Changing the Object's Contents. The implementation of the *ReplicatedElement* uses the deconstruction and reconstruction phases to maintain the *replicas* list as the object is transferred around the network. The deconstruction phase is used to create the list for the new replica, while the reconstruction phase is used to inform the existing replicas of the addition of a new replica. Figure 6-3 illustrates the various steps that are covered by the deconstruction and reconstruction of the object, which are listed below.

Step 1. During deconstruction at site 1, an original replica A is copied to make a substitute object A'.

Step 2. The list of replicas in A' is updated to contain a remote proxy which points to the original replica A.

Step 3. For each of the other replicas, the list of replicas must be updated to include a proxy which points to the new replica. To accomplish this the replica that was just created broadcasts a proxy for itself to all the other replicas in the system when it is being reconstructed on the new site.

We will now detail this implementation. The list of replicas is constructed for the new object when it is being copied from an existing replica. Step 1 occurs in the deconstruction phase and creates the substitute object by copying the current wrapper and all its instance variables using a deep copy (line 3 of Figure 6-4).

```
1 defaultWrapperDeconstructionBlock
2       ^[: currentWrapper | | newVersion |
3           newVersion := currentWrapper deepCopy. "Step 1"
4           newVersion replicas add: (RemoteProxy for: currentWrapper). "Step2"
5           newVersion replicas: (RemoteCopy for: newVersion replicas).
6           newVersion]
```

Figure 6-4. During deconstruction, the list for the new object is formed.

The list of replicas in the wrapper that was copied is almost, but not quite, complete. While the *replicas* list of the wrapper being copied had pointers to all the other wrappers, it is missing a proxy to itself. For example, in Figure 6-1, Replica #1 could be copied to create a new replica, say Replica #4. If the system were to copy Replica #1 identically, then Replica #4 would point only to Replica #2 and Replica #3. A proxy must be added to the list that points to Replica #1. Step 2 (line 4), creates a proxy for the wrapper being copied and adds it to the list of replicas. Finally to ensure that the list of replicas is passed as a copy (rather than a remote reference), the clone list itself is made into a remote copy (line 5).

When the replica is rebuilt on the other side, its wrapper contains the list of all other replicas in the system. However, the other replicas do not have a proxy to this new replica. When the replica is reconstructed, it will broadcast a remote proxy for itself to all the other replicas (Step 3), bringing them up to date. Figure 6-5 shows how the reconstruction of an object is used to broadcast a proxy of the replica that was just created to all the other replicas. In line 3 the list of replicas is iterated over, and a *RemoteProxy* of the new replica is sent to each of them.

```
defaultWrapperReconstructionBlock
        ^[: remoteObject |
                remoteObject replicas do: [: aReplica | "Step 3"
                    aReplica addClone:
                        (RemoteProxy for: remoteObject)].
                remoteObject]
```

Figure 6-5. The replicas are reconstructed by sending a proxy to all the replicas.

In this section, we have shown the default implementation for a replicated element. It demonstrated how the deconstruction and reconstruction blocks are used to maintain the list of replicas at other sites, and how the message interception layer is used to broadcast messages to these replicas. We now show how this scheme can be modified to support locking.

6.2 Case Study #2: Replicated-Locking Objects

With replicated objects, two replicas may be sent messages at the same time. These messages may arrive in different orders at different sites, leading to different execution orders and possible loss of consistency (see Chapter 2 for a detailed explanation of the need for consistency). To provide control over consistency, we created replicated locking objects that add strict concurrency control to replicated objects.

The *ReplicatedLockingElement* inherits its functionality from the *ReplicatedElement* described in Section 6.1. However, it adds guarantees of the contained object's consistency by ensuring that any message sent to the object will not execute until it acquires a system wide lock. Only after the message has executed at each site is that lock released. The *ReplicatedLockingElement* uses the pre and post-consistency hooks to implement this form of locking. Figure 6-6 illustrates how site 3 acquires the lock when it receives a message 'Y'. Before broadcasting the message to all the other sites, site 3 must first obtain a centralized lock. To do this, site 3 sends a request for the lock to the lock object located on site 2. Once the lock is obtained, site 3 will broadcast message 'Y', wait for it to execute at each site, and then release the lock.

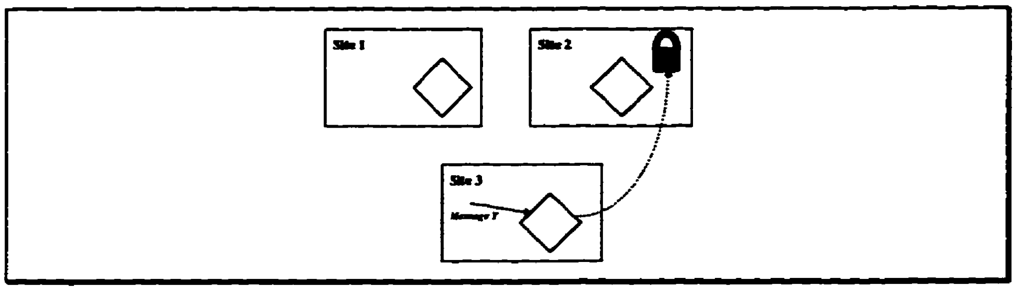

Figure 6-6. The acquisition of a centralized lock before message Y can be sent.

Modification of the Concurrency Mechanisms. To perform the locking, the *ReplicatedLockingElement* uses a standard Smalltalk *Semaphore*, which is created when the object is initialized in *#initializeGWElement* (shown in line 2 of Figure 6-8). This object is declared as centralized so that when the wrapper is distributed, there will be a single instance of the lock that all sites use. Whenever a new replica is created, it will have a proxy to the centralized lock.

```
1  defaultWrapperDeconstructionBlock
2       ^[: currentWrapper | | newVersion |
3           lockObject wait.
4           newVersion := currentWrapper deepCopy.
5           newVersion replicas add: (RemoteProxy for: self).
6           newVersion replicas: (RemoteCopy for: newVersion replicas).
7           newVersion]
8
9  defaultWrapperReconstructionBlock
10      ^[: remoteObject |
11          remoteObject replicas do: [: aReplica |
12             aReplica addClone:
13                  (RemoteProxy for: remoteObject)].
14          lockObject signal.
15          remoteObject]
```

Figure 6-7. New wrapper deconstruction and reconstruction blocks.

The *#handlePreConsistency:* and *#handlePostConsistency:* methods respectively wait and signal the semaphore (lines 6 & 7). The message is broadcast and then executed at all sites before the lock is released by having the *#distributeMessage:* wait until all the replies are received, Only then does it release the lock (line 13). In the *ReplicatedElement,* we saw that this method forks a separate thread, which allowed the current thread to continue running. In the *ReplicatedLockingElement* implementation, no new thread is forked — the current thread (which is doing the message sends) waits until all the replies are received from the replicas, which indicates they have processed the message. Once these replies have been received, the *#handlePostConsistency:* method will free the lock.

```
initializeGWElement
        lockObject := Semaphore forMutualExclusion centralized.
        ^super initializeGWElement

handlePreConsistency: aMessage
        lockObject wait

handlePostConsistency: aMessage
        lockObject signal

distributeMessage: aMessage
        replicas do: [ :aReplica |
                aReplica handleRemoteMessage: aMessage]
```

Figure 6-8. Locking in the pre-consistency and post-consistency methods.

The replicated element must handle the possibility that a message will be sent while the object itself is being replicated. If a message were sent by a replica after the object was copied, but before the object broadcast its proxy to all the remote copies, then the remote proxy would not receive the message and subsequently would be in an inconsistent state. To exclude this possibility, the *#defaultWrapperDeconstructionBlock* and *#defaultWrapperReconstructionBlock* are modified to acquire and release the lock before and after the deconstruction and reconstruction are done, ensuring that the creation of a new replica is mutually exclusive to the broadcasting of messages (lines 3 and 11 of Figure 6-7).

6.3 Case Study #3: Centralized Objects

The final case study chosen from the default implementation explores another common data distribution scheme used by groupware toolkits: centralized objects. Centralized objects are useful for several reasons. They are easy ways to implement concurrency, because they serialize messages that are sent to them. Additionally, they are useful for implementing concurrency strategies (such as the lock used in the replicated locking implementation), because the developer can use the built in features of the OS or programming language to implement mechanisms such as semaphores or mutexes to guarantee mutual exclusion.

In GEN, a centralized object resides on the machine which creates it. Remote sites receive a proxy when the object is distributed to them, which forward all the messages they receive to the centralized object. The wrapper itself is not distributed between sites, so centralized elements are straightforward to implement: they can use the functionality of the distributed objects layer to create the remote references.

The centralized object exploits the distributed object layer by creating a proxy at each site, rather than a new wrapper. Figure 6-9 shows how one site (Site 3) will contain the object while the other sites (Sites 1 & 2) only contain proxies which point to the centralized wrapper. When either Site 1 or Site 2 send a message to the object, the proxy will automatically forward it on to the wrapper.

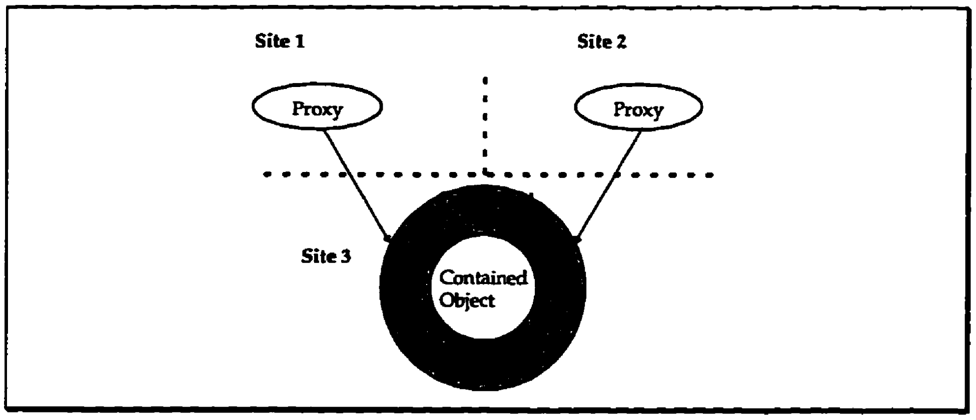

Figure 6-9. How a centralized object is represented on other machines.

The remote object specializes two methods. First, *#isCopiedRemotely* is changed to force a proxy to be made; this causes the proxy to be made automatically by the distributed objects layer. As shown in lines 1-2 of Figure 6-10, the method *#isCopiedRemotely* answers false. Second, *#performMessage:* now forwards the message to the local object by passing it on to the contained object (lines 4-5).

```
1 isCopiedRemotely
2         ^false
3
4 performMessage: aMessage
5         ^containedObject perform: aMessage
```

Figure 6-10. Code to implement a centralized object.

6.4 Summary

The centralized and replicated objects shown in these sections have demonstrated three implementations that allow developers to control data distribution and concurrency mechanisms. They serve as case studies to demonstrate the power of the meta-interface, as they demonstrate how the meta-interface can be used to construct these schemes.

These particular implementations are included in GEN, and their API is provided as part of the programmer's interface. What is important to realize is that they are really no different than anything else constructed using the meta-interface. Indeed, the default implementations are simply components built using the meta-interface by the toolkit developer for two reasons: to allow the application developer to use the toolkit without understanding the meta-interface; and to give the application developer examples of how the meta-interface can be used to construct sharing strategies.

The meta-interface is designed not only for power, but for flexibility. In the next chapter we will see how the meta-interface can be used, by application developers, to build new data sharing strategies, including migration and a form of optimistic locking.

7. Case Studies: Extending GEN by Adding New Data Sharing Strategies

In the previous chapter, we showed how the meta-interface could be used to construct the default implementations seen in the programmer's interface. In this chapter, we present three additional case studies that demonstrate the flexibility of the meta-interface by showing how it can be used to extend and construct new data sharing schemes. These schemes were not planned for when the meta-interface was designed.

The first case study provides a way to reduce the number of messages distributed by a replicated object by allowing the developer to specify which messages are broadcast, and which are not. The second case study implements a form of object migration that moves a centralized object around the network, based on which site uses the most frequently. The third case study demonstrates a new concurrency control mechanism by implementing a form of optimistic locking. It is not our intention to champion these sharing strategies, but rather to demonstrate how the meta-interface can be used by application developers to extend the range of strategies available to application developers.

7.1 Case Study #4: Selective Broadcast of Messages

Replicated objects currently broadcast all the messages they receive to all sites. Yet replicated objects do not really have to broadcast all messages sent to them. For example, because messages that read the state of a replicated object can be handled by the local object alone, they do not need to be broadcast over the network. Replicated objects, as implemented in the default implementation, are inherently slow. All messages are broadcast, and broadcasting is an expensive operation. Consequently, we would like to change the implementation of replicated objects to make their message broadcasts more selective.

In the *ReplicatedSelectiveElement*, we give the developer the ability to specify those messages that are broadcast to all sites, and those that are only sent to the local site. In particular, the developer must specify the name of each method that is not to be broadcast. When a message is received by the wrapper, the wrapper checks to see if the message is on the list, and routes it accordingly.

The *ReplicatedSelectiveElement* inherits from the *ReplicatedLockingElement*, with the addition of a *Set* of the method names (called message selectors) that should not be broadcast. The message *#addNonBroadcastMethod:* adds method names to this collection, and the *#distribute Message:* method is changed to filter the messages so it can selectively broadcast them.

As we can see in Figure 7-1, the method *#initializeGWelement* (lines 1-3) creates the set called *protocols* to hold the names of the methods (selectors) when the object is initialized. The next method, *#addNonBroadcastMethod*: adds the specified message selector to the set of message selectors to be ignored by adding it to the *protocols* set in line 6. Line 9 of *#distributeMessage*: contains the test to see if the message is contained in the protocols set. If it is, the message will not be broadcast. Otherwise *#distributeMessage:* broadcasts the message to all the other sites by using its superclasses (*ReplicatedLockingElement*) *#distributeMessage:* method (line 11).

```
1  initializeGWElement
2        protocols := Set new.
3        ^super initializeGWElement
4
5  addNonBroadcastMethod: aMessageSelector
6        protocols add: aMessageSelector
7
8  distributeMessage: aMessage
9        ^(protocols includes: aMessage selector)
10           ifFalse: [
11                  super distributeMessage: aMessage]
```

Figure 7-1. Modifications for *ReplicatedSelectiveElement* wrapper.

network based on frequency of use. (While we have chosen frequency as the criteria to demonstrate how migration can be triggered, other measures such as recency could have been used).

In the migration scheme, each site maintains a wrapper. One of the sites maintains a wrapper with the current copy of the object and a frequency count of the messages received from each particular site, as illustrated by the top circle in Figure 7-3. The other sites each maintain a wrapper containing a proxy to the *wrapper* containing the original object, as shown by the circles at the bottom of the figure. It is important to note that these proxies point to the wrapper, so that the wrapper can intercept messages sent to the contained object, allowing it to maintain the frequency counts. If the number of messages from a particular site exceeds the number of messages received from the local site by a threshold value, then the contained object is moved to the site with the highest frequency count.

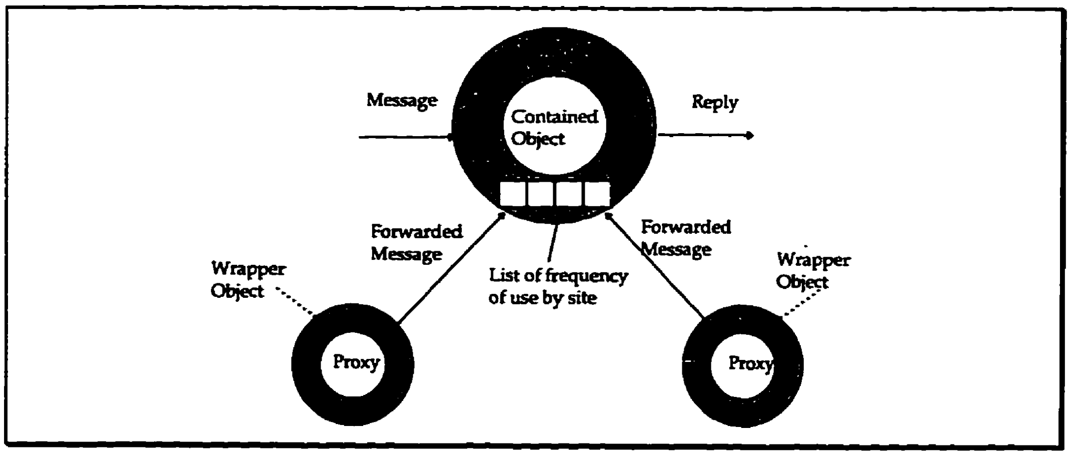

Figure 7-3. Structure of a migrating object. When the relative frequency count for a particular site exceeds a threshold the object is migrated.

The implementation of the *MigratingElement* requires a modification via the meta-interface to both the message routing scheme and the object's contents when the wrapper is distributed.

1. We modify the message routing scheme to add information about the source of each message send, which allows us to maintain site-specific frequency counts. The routing scheme must also move the contained object between sites, and keep the replicas informed about the actual object's location.
2. When the wrapper is distributed, it may not be copied identically to the new site. For example, if the wrapper being copied contains the actual object, the new wrapper will have to be modified to contain a proxy to the wrapper being copied.

The next two sections detail the implementation of the *MigratingElement*. This object inherits from the *ReplicatedLockingElement* so that messages sent to it are guaranteed to be mutually exclusive. It also contains an additional object: a frequency list called the *frequencyCollection* that holds a count of the number of messages received from each site. The first section deals with the changes to the message routing scheme, which adds the frequency information necessary to determine when and where the object will migrate, as well as performing the actual migration. The second section shows how the deconstruction and reconstruction phases are used to create new replicas that maintain the location of the actual object.

Changes to the message routing scheme. The first change to the message routing is the addition of information about where a message was sent from. This information will be used to maintain the frequency collection. In terms of message routing, the migrating object overrides the default behaviours of *#distributeMessage:*, *#performMessage:* and *#handleRemoteMessage:*. The message routing for migrating objects is more complex, so we have detailed it in several ways: Figure 7-4 presents a graphical representation of the path that the message takes; Figure 7-5 presents the code that handles the routing of the message; and below we describe the steps that a message travels through.

Once a message is received by an object it passes through the following steps (which are shown in Figure 7-4):

1. The wrapper's *#distributeMessage:* is passed the message. If the contained object is a remote object, it forwards the message on to the **wrapper** containing the remote object using the *#handleRemoteMessage:* method.
2. The wrappers *#performMessage:* is passed the message. If the contained object is a local object, the message is sent to itself through the *#handleRemoteMessage:* method.
3. The *#handleRemoteMessage:* executes the message locally, determines the frequency count for each of the sites that have sent the message.
4. If the frequency count for one site exceeds the frequency count for the local site by a threshold value, the object is moved to that new site.

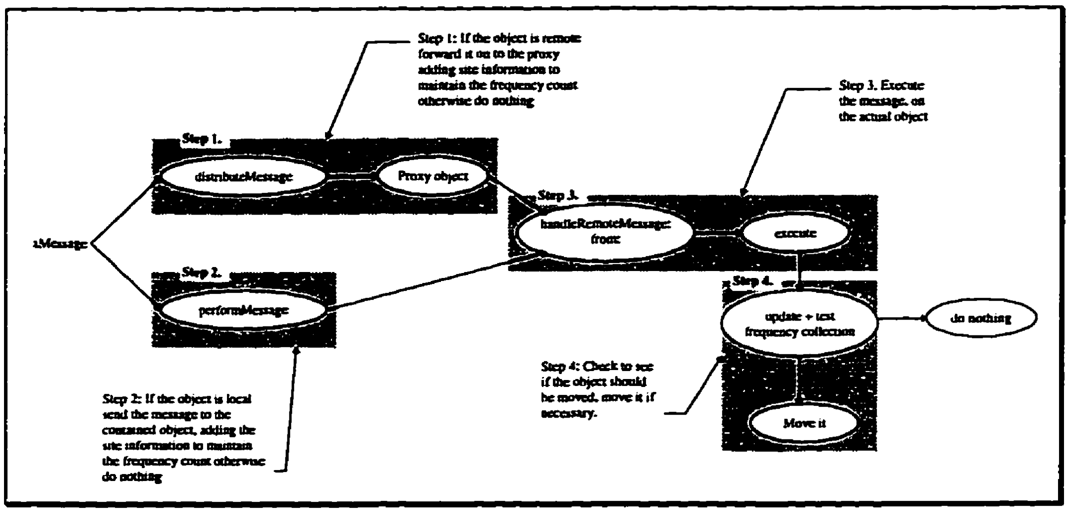
Step 1: If the object is remote forward it on to the proxy adding site information to maintain the frequency count otherwise do nothing
Step 3. Execute the message. on the actual object
Step 1.
distributeMessage
Proxy object
aMessage
Step 3.
handleRemoteMessage: from:
execute
Step 2.
performMessage
Step 4.
update + test frequency collection
do nothing
Step 4: Check to see if the object should be moved. move it if necessary.
Move it
Step 2: If the object is local send the message to the contained object, adding the site information to maintain the frequency count otherwise do nothing

Figure 7-4. The path of a message in a migrating object.

Step 1. The *#distributeMessage:* method first checks to see if the contained object is a remote proxy (line 2 in Figure 7-5). If it is not remote, no distribution is necessary and the message will be handled by *#performMessage:* in Step 2. Otherwise, the object is remote, and *#handleRemoteMessage:from:* is called with the additional parameter of a *RemoteProxy* for wrapper (line 4 & 5) (see also Step 1 in Figure 7-4). The proxy is used as an index into the frequency collection.

Step 2. The method *#performMessage:* checks to see if the contained object is local (line 8). If it is, the message *#handleRemoteMessage:from:* is sent to itself with a remote proxy for itself added on (see Step 2 in Figure 7-4). This is done because

#handleRemoteMessage:from: maintains the frequency collection and the remote proxy serves as the index into the frequency collection.

Step 3. The largest change is to *#handleRemoteMessage:*, to which an additional parameter was added to indicate the location of the machine sending the message. Its name is changed to *#handleRemoteMessage:from:*, as shown in line 13 of Figure 7-5. This method performs the message on the local object (line 15, and Step 3 of Figure 7-4).

Step 4. In lines 16 to 18 *#handleRemoteMessage:from:* executes a migration test (line 17; see also Step 4 of Figure 7-4). If the test evaluates to a new site, the wrapper will move the contained object to that new site (line 19). The migration test is run by executing a

```
1  distributeMessage: aMessage
2        containedObject isRemoteObject
3        ifTrue: [
4            ^containedObject handleRemoteMessage: aMessage from:
5                    (RemoteProxy for: self)]
6
7  performMessage: aMessage
8        containedObject isRemoteObject
9        ifFalse: [
10           ^self handleRemoteMessage: aMessage from:
11                   (RemoteProxy for: self)]
12
13 handleRemoteMessage: aMessage from: aClone
14       |newOwner result|
15       result := containedObject performMessage: aMessage.
16       (newOwner :=
17           migrationBlock value: aMessage value: aClone value: self) notNil
18           ifTrue: [
19                   self moveTo: newOwner].
20       ^result
```

Figure 7-5. Message Routing changes for migrating objects.

migration *Block* (an executable piece of code in Smalltalk). A default migration block (shown in Figure 7-6) is defined by this class, and takes three parameters: the message; the replica sending the message (a proxy); and the wrapper (self). When a message from a particular site is received, the frequency collection increments the count for the number of messages sent to the object for that site (lines 3 & 4). Once the count has been incremented, the number of hits from the local object (line 6) is compared to the number of hits from the highest ranking site (line 5). If the difference is greater than 10, the new owner is answered from this method (line 7). When this block returns a new site, the object will be migrated to it. Otherwise, the object will remain at the current site.

```
1 defaultMigrationBlock
2       ^[:aMessage :aSourceMachine :aWrapper |
3           (aWrapper frequencyCollection
4                   increment: aSourceMachine machineID).
5           aWrapper frequencyCollection highestValue >
6                   ((aWrapper frequencyCollection valueOf:
                            (RemoteProxy for: aWrapper) machineID) + 10)
7                   ifTrue: [aSourceMachine]
8                   ifFalse: [nil]]
```

Figure 7-6. Test block to determine if the object should be migrated

The message routing system also handles the moving of the contained object to the new site. *#HandleRemoteMessage:from:* is responsible for moving the object if the migration block returns a new site location (line 5 of Figure 7-5). There are two methods that move an object: *#moveTo*: and *#receiveObject:*, as shown in Figure 7-7. *#MoveTo:* first tells all the other sites where the object is going to be transferred to by setting their contained object to the new clone (line 2). The object is then sent to the new site using *#receiveObject:*. The *#receiveObject:* message places the clone in the copy of the actual object in the contained object instance variable (line 7).

```
1 moveTo: destination
2       replicas do: [: aReplica | aReplica object: destination].
3       destination receiveObject: (RemoteCopy for: containedObject).
4       self object: aClone.
5
6 receiveObject: anObject
7       containedObject := anObject
```

Figure 7-7. Moving the contained object.

The message routing has now provided the functionality necessary for directing the message to the site currently containing the object, and also for moving the object. The following section examines how the contents of the object are distributed.

Controlling the object's contents. When the object is distributed to a new site, the contained object must point to the wrapper containing the original object. However, this requires determining whether the wrapper being copied contains the actual object or a proxy. If it contains the actual object, a proxy must be constructed that points to the wrapper being copied. To determine if the contained object is a remote proxy, the message *#isRemoteObject* is sent to the contained object (line 7 of Figure 7-8). This message will return true if the contained object is a proxy, and false if it is local. If the contained object is not a proxy, then it must be the actual object. In this case, a proxy to the current wrapper is distributed. If the contained object is a proxy, then the proxy itself will be forwarded automatically.

```
1  defaultWrapperDeconstructionBlock
2        ^[: currentWrapper | | newVersion |
3            lockObject wait.
4            newVersion := currentWrapper deepCopy.
5            newVersion replicas add: (RemoteProxy for: self).
6            newVersion replicas: (RemoteCopy for: newVersion replicas).
7            containedObject isRemoteObject
8                    ifFalse: [
9                            newVersion object: (RemoteProxy for: self)].
10           newVersion]
```

Figure 7-8. Default deconstruction block for migrating object. The contained object is changed to a *RemoteProxy* for the contained object.

In this case study of migrating objects, we saw how an object's behaviour can be altered via the meta-interface to migrate itself to a specific site that uses it frequently, thus speeding up local response times.

7.3 Case Study #6: Optimistic Locking

Optimistic locks work by assuming that a lock request will be granted. After a request is made, it proceeds without waiting for the reply (assuming it will be a positive reply). If the lock is not granted, the system will somehow have to restore itself to a state similar to the one before the request was made. While more complex than simple conservative locks, this scheme is suitable for applications running on networks where interprocess communication is slow, where getting a lock is relatively expensive, and where the likelihood of being denied that lock is relatively small. Optimistic locking under these circumstances can provide a more responsive yet still consistent system than conservative locks, which is important in user interactions.

In this section we create a form of optimistic locking. After a lock request is made, messages will be allowed to execute in the local copy, even though the object is waiting for a global lock to be granted. However, messages are not broadcast to the other copies until the lock is acquired. This strategy differs from traditional optimistic locking mechanisms, where the changes are broadcast to all sites while the lock is being acquired. Still, it means that local responsiveness is high, which is especially important for managing a local user's interactions.

Optimistic locks are implemented in GEN as a wrapper that checkpoints the state of the object, requests the lock, but then makes the changes to the local object instead of blocking completely. While waiting for the lock request to be granted, additional changes may be made to the object, and processing continues at the local site as if the object had received the lock. This is implemented by storing the messages in a stack rather than performing a broadcast. If the lock is granted to the object, then the object broadcasts the stored messages to all the clones and releases the lock. If the lock is refused (e.g. it is already held by another clone), the object restores the checkpointed version that contains its original state before any messages were sent, and generates an event to indicate that the lock was denied.

Delving into the details a bit further, our optimistic lock objects, which inherits from the *ReplicatedLockingElement*, use the pre-consistency phase to checkpoint the object by copying it. The request for a lock starts an asynchronous task that performs the request in a background task. While the lock is being acquired, the object remains in a state of "waiting" for the lock and the program continues executing. Any messages sent to the object while it is waiting for the lock are queued. Finally, the object is informed of whether or not it received the lock. From that result, it will return to the checkpointed state (when denied), or broadcast the messages it received while waiting for the lock (when granted). The *ReplicatedLockingElement* waits when the lock is not acquired. The

OptimisticLock changes this to one that fails (rather than waits) when the optimistic lock[2] is not granted. Similarly, concurrency control mechanisms are overridden: we fork off an asynchronous thread to request the lock and then checkpoint the object.

Changes to the Concurrency Control Mechanism. The first part of the consistency mechanism handles the checkpointing of the object and the acquisition of the lock. The *#handlePreConsistency:* method checks to see if an asynchronous request for the lock is in progress. If it is, then another message was executed locally before this one, and a lock request is already in progress. In this case the system does not need to acquire another lock. Figure 7-9 shows how the system checks the status of the lock, which is nil if there is no current lock request (line 2). If the lock must be acquired, the message *#requestLock* is sent to the wrapper (line 3).

The *#requestLock* method (starting at line 5) performs several actions. First, it changes the lock status of the lock to 'waiting' (line 6), so that future messages sent to the object do not create further asynchronous lock requests while another request is already in progress. Second, it creates a checkpoint copy of the contained object by performing a *#deepCopy* (line 7). Third, a new stack for the messages is created by instantiating an *OrderedCollection* (line 8). Finally, an asynchronous process (the *ObjectLockTask*) is forked to request the lock and wait on the result.

---

[2] This is actually implemented as a method extension to the Semaphore class in Smalltalk.

```
1  handlePreConsistency: aMessage
2         lockStatus isNil ifTrue: [
3             self requestLock].
4
5  requestLock
6         lockStatus := #waiting.
7         oldObject := containedObject deepCopy.
8         messageStack := OrderedCollection new.
9         Processor activeProcess
10            sendSelector: #objectLock:for:
11            withArguments: (Array with: lockObject with: self)
12            to: ObjectLockTask create
13
14 lockStatus: aStatus
15        lockStatus == #waiting ifTrue: [
16            (lockStatus := aStatus)
17            ifTrue: [
18                    self broadcastMessageQueue]
19            ifFalse: [
20                    self restore]].
21
22 broadcastMessageQueue
23        [[messageStack size > 0] whileTrue: [
24            super distributeMessage: messageStack removeFirst].
25        self releaseLock] fork
26
27 releaseLock
28        lockStatus := nil.
29        lock signal.
30
31 restore
32        containedObject := oldObject.
33        self generateEvent: #OptimisticLockFailed callData: #().
```

Figure 7-9. Check-pointing the object at the pre-consistency stage.

At this point, the message will execute locally, as it would in a *ReplicatedElement*. However, the message will be stored on a stack (discussed later). The post-consistency method *#handlePostConsistency:* (not shown) performs no action, because the acquisition or rejection of the lock will happen asynchronously. When the asynchronous lock request completes, it will send the message *#lockStatus:* to the wrapper waiting on the lock (line 14). The parameter *aStatus* is a boolean, which indicates whether the process was successful at acquiring the lock. If the lock is successfully acquired, then all the messages that have been queued will be broadcast to all the other sites by executing the *#broadcastMessageQueue* (line 18). If the lock was not acquired, it must restore the state of the object by calling restore (line 20). The restore method rolls back the object by copying the checkpoint object over the contained object (line 32) and generates an event called *#OptimisticLockFailed* (line 33).

To perform the broadcast when the lock is acquired, *#broadcastMessageQueue* iterates through the stack of messages that have accumulated and uses the superclass' *#distributeMessage:* to send each message to all the other sites (line 24). Finally, the lock is released once all the messages have been sent and executed at all of the sites (line 25).

Modification of Message Routing. The message routing system is changed in two ways. First, messages are not broadcast by *#distributeMessage:*. Rather they are stored in the message stack until a lock is either acquired or not. When the lock is acquired, the messages in the queue will be executed. Second, if the current object is waiting for a lock and a message is received from another object, then the lock will be assumed to have failed and the wrapper will return the object to the checkpointed state.

```
1 distributeMessage: aMessage
2       ^messageStack add: (aMessage deepCopy)
3
4 handleRemoteMessage: aMessage
5       lockStatus = #waiting ifTrue: [
6           self lockStatus: false].
7       containedObject performMessage: aMessage.
```

Figure 7-10. The queuing up of messages to be distributed.

The stacking of messages is handled by the *#distributeMessage:* method, which adds the message to the message stack to wait for broadcast (line 2 of Figure 7-10). The *#handleRemoteMessage:from:* protocol may receive a message that was broadcast from another site, while the current object is waiting on a lock. In this case, we know that the lock will not be granted, since another site holds it. When the *#handleRemoteMessage:*

from: method receives a message from a remote object, the lock status is changed to false (line 6), forcing the old version of the object to be restored before the message executes.

7.4 Summary

In this chapter, we have shown the power of the meta-interface through a set of case studies that illustrate the flexibility of the implementation. We have implemented three new ways in which data distribution and concurrency control can be used within the GEN toolkit, which in turn demonstrates that the toolkit is expandable. None of these methods drove the original design of GEN. Combined with the default implementations, we have constructed a total of six different data sharing strategies, which vary both the way data is distributed and the method used to control concurrency. We have also implemented several other case studies, that have not been described here. These include a read/write lock element, a combined centralized/replicated element, and a broadcast priority element. While these are not described further, they are similar in complexity and style to the examples already seen.

This concludes our demonstration of how a meta-interface architecture within a groupware toolkit can allow developers to control both the distribution of data and the way concurrency is managed.

8. Discussion and Conclusion

We have now concluded the discussion of the implementation and evaluation through case studies of the GEN system. In this chapter, we discuss the contributions of this thesis. The discussion begins with a summary and a critique of the work. Following this, a section on future work looks at how this research can be developed further to increase our understanding of how open implementations are useful in developing groupware toolkits. Finally, we conclude by listing particular contributions that this thesis has made to groupware and open implementations.

8.1 Summary

I have argued that groupware application developers sometimes require control over the strategies used to shared data. The argument began with a review of groupware toolkits in Chapter 2, and showed how different toolkits have chosen different strategies for sharing data. Typically, the choice of strategy was based on what the toolkit developer believed would be the runtime requirements for factors such as consistency, speed, and ease of implementation.

The default strategy supplied by a particular toolkit is often sufficient to prototype and build groupware applications. However, there will be some cases where the toolkit will be used in circumstances different from those envisioned by the toolkit developer. In these cases, the toolkit will not match the needs of the application developer, and they will have to work around the toolkit with strategies such as hematomas and coding between the lines.

I then introduced a toolkit design strategy, called 'open implementations', that helps overcome these limitations. Chapter 3 described how open implementations separate toolkits into two parts: the programmer's interface and the meta-interface. The programmer's interface is normally used to create applications, just as in a standard toolkit. However, the meta-interface allows application developers to modify the strategies in the toolkit when they do not meet an application developer's needs.

In Chapters 4 and 5, I showed how an open implementation can be created for groupware toolkits. The chapter describes the implementation and API for GroupEnvironment (GEN), a groupware toolkit. GEN gives application developers control over how data is shared by providing an interface for modifying how data is distributed, and for modifying the concurrency control mechanisms.

I demonstrated the flexibility of this implementation through six case studies: three in the default implementations described in Chapter 5; and three extended implementations described in Chapter 6. All use the meta-interface to develop new strategies for sharing data. These examples illustrate in detail how the meta-interface is used to create new sharing strategies, such as different data distribution schemes (centralized, replicated and migrating), as well as different concurrency schemes (locking, optimistic locking).

8.2 A Critique of GEN

The development of a meta-interface is an iterative process (Kiczales et al., 1993). This was our first iteration of a meta-interface for a groupware toolkit, and is of course incomplete in several ways. This critique of the GEN prototype will direct the next iteration of a meta-interface in groupware by identifying weaknesses and open research areas.

One of the principles of meta-interface design is conceptual separation, as discussed in Chapter 3. Conceptual separation states that modifying one component in a meta-interface should not impact on other components of the system. This was not entirely achieved in GEN. Although the actual implementation of the components of notification, concurrency and distribution are separated, the developer has to be aware of how they relate to one another and how to compensate for interactions. For example, when the locking mechanism is changed (such as in the replicated locking implementation), the developer must also be concerned about how the objects that support the lock are distributed around the network. The distribution mechanism for the lock may not coincide with the way the wrapper or the contained object is distributed. It is not clear at this time that conceptual separation is possible in this area. However, because of the complexity it adds to modifying the meta-interface, it needs to be investigated further.

GEN's meta-interface to concurrency management is overly restrictive, as it only allows the developer to specify pre- and post-actions when a message is intercepted. This technique, although useful, requires the application developer to create the entire concurrency control strategy. They have to determine how to implement synchronization strategies to get the desired level of concurrency control. This may be difficult and time consuming. Recent parallel work by Dourish (1995a,b; 1996) has presented a meta-interface which simplifies the creation of concurrency mechanisms for groupware toolkits. Dourish's concurrency model uses negotiation, which gives the developer control over the *degree* of consistency of shared objects. For example, some portions of an object may be consistent while other portions of the same object are inconsistent. This may be useful in a shared drawing program: drawing areas that no one is using can be inconsistent, while areas that are being used by multiple people need to be very consistent. If Dourish's model were applied to GEN, it could make the construction of new concurrency mechanisms both richer and simpler.

The current implementation of GEN is sufficient to demonstrate our ideas. However it is slow and far from robust. Still, we have built simple applications (a shared white board, a brainstormer, and a meeting scheduler (O'Grady & Greenberg, 1994)) to test the different strategies of data sharing; again these are slow. This leads to the question: is the technique of open implementation with its additional layer of abstraction inherently slow and unusable? We must consider the reasons for this slowness both in terms of the particular implementation used in this thesis as well as the cost of using an additional layer of abstraction. First, in terms of the particular implementation used we must point out that the distributed object layer and object marshalling system were developed only to demonstrate our ideas, and are crude and relatively slow. For example, they copy the same object multiple times as it is being transmitted between sites. An optimization of these components would significantly increase the speed of the application. Second, although the additional layer of abstraction adds additional overhead to the processing time, it gives developers the ability to customize how data is distributed at a fine grain. This power to customize the distribution mechanism is likely to allow speed gains and other benefits that far outweigh the slowness of an additional layer of abstraction. Additionally, in groupware toolkits the bottlenecks are not usually processing speed but rather the time it takes to send messages between sites (Greenberg & Roseman, 1997), and the additional processing time associated with this additional layer of abstraction will have little impact. Finally, the ideas in this thesis have already influenced a limited implementation of meta-interfaces within GroupKit, another groupware toolkit being developed in our lab. The GroupKit meta-interface allows the developer to control how an environment and its contained data are distributed around the network, and it runs fairly quickly despite this additional layer of abstraction (Roseman, 1995).

8.3 Future Work

The work in this thesis has been directed towards demonstrating open implementations and the viability of giving the developer control over the way data is shared. The work can be built upon in several ways: 1) enhance the GEN toolkit itself; 2) iterate the meta-interface; 3) characterize different sharing strategies; and 4) consider how the ideas can be applied to other areas of distributed computing.

1. *The enhancement of GEN as a usable toolkit.* The model we have presented in GEN has demonstrated how a meta-interface can be constructed. However, there are two significant areas in which GEN could be enhanced. First, as mentioned previously, speed is an issue in GEN and needs to be addressed. The distributed object layer in GEN was implemented to explore the ideas for developing groupware applications and is inefficient. Recently, commercial distributed object implementations have become available, and replacing GEN's distributed object layer with an efficient commercial version could greatly increase its speed.

   Second, GEN explored only the runtime architecture side of building groupware toolkits. However, developers need interface components to build groupware applications (Greenberg & Roseman, 1997). These include items such as telepointers, transparent overlays, and multi-user scroll bars. These components should be added to GEN to give developers a more complete environment for constructing groupware applications.

2. *Iteration of GEN's meta-interface design.* The second area of study concerns how developers use the meta-interface. Our first iteration has given developers control over the routing of messages, the contents of objects, and pre- and post-concurrency message hooks. We have defined a particular family of sharing strategies that developers can manipulate. While I argued that this family of choices are useful to developers, there are probably better ways to implement the strategies, and there are probably other families of strategies to be considered. For example, pre- and post-concurrency mechanisms may not be sufficient to implement the consistency mechanisms that developers really require, and particular approaches may be outside the family of behaviours that we have provided for. The toolkit must be tested in the construction of real applications.

3. *Empirical evaluation of strategies.* Finally, developers need to be able to identify which strategies are useful under which circumstances. Graham & Urnes (1996a) have recently characterized a centralized and a cached data sharing strategy in terms of their performance using empirical methods. Using GEN, new strategies can be created. These should be empirically evaluated to determine under what conditions which data sharing strategies will be most effective. Thus GEN could become a tool to measure different architectural capabilities.

4. *Application of open implementations and wrappers to distributed computing in general.* Although the ideas presented have shown how open implementations and wrappers can be used to provide flexible data sharing in groupware toolkits, the techniques used may more widely applicable in distributed computing. In particular the idea of separating the data sharing mechanisms (e.g. using wrappers) from the implementation of the object being distributed (e.g. an OrderedCollection) may prove useful for writing other types of distributed computing, such as distributed agents.

8.4 Contributions

GEN has made contributions to both groupware development and open implementations.

8.4.1 Contributions to Groupware

1. *Multiple data sharing strategies in a single groupware toolkit.* Current groupware toolkits have presented application developers with a single strategy for sharing data in an application. GEN allows developers to select a data sharing strategy for individual objects in their application, letting them decide based on the particular needs of the data. GEN provides three default implementations: replicated; replicated locking; and centralized. Only one other toolkit provides more than a single strategy, however, it limits the choice to one of two possible strategies (Graham & Urnes, 1996b).

2. *An extensible groupware toolkit.* Current groupware toolkits provide developers with a single inflexible strategy for sharing data. The meta-interface of GEN provides developers with the ability to manipulate the implementation of the toolkit itself to create new strategies for sharing data. In this thesis, we have demonstrated three additional techniques that can be constructed to share data: selective broadcasts; migrating objects; and a form of optimistic locking. This idea has influenced GroupKit's design, which has now created an open implementation for its environments (Roseman, 1995). A mentioned in Chapter 2, Dourish (1995a) has also explored meta-interfaces for groupware toolkits, and has presented an model for providing extensible consistency mechanisms.

3. *Environments for organizing shared data.* GEN borrows the concept of environments found in traditional programming languages (such as Scheme) for organizing data to help groupware programmers organize shared data. GroupKit has also adopted this data structure, and it is used to implement some of their demonstration applications (Roseman & Greenberg, 1996).

8.4.2 Contributions to Open Implementations

1. *Demonstration of how open implementations can be applied to groupware toolkits.* Open implementations are a relatively new concept that suggests toolkits can become more powerful and useful to programmers by providing a second interface to control design decisions that are usually made by the toolkit developer. Both myself and Dourish (1995b) have demonstrated how open implementations can be applied to the area of groupware, giving application developers more control over how their data is shared. This broadens the range of toolkits using open implementations.

2. *Demonstration of the use of wrappers to construct open implementations based on message interception.* GEN has used wrappers to implement a meta-interface which allows the dynamic addition of new behaviours to objects. The details in this thesis show how wrappers can extend the functionality and change the way messages are sent between objects.

8.5 Conclusion

The current generation of groupware toolkits operates on the principle that only a single and unalterable sharing strategy implemented by the toolkit should be available to application developers. However, the choice of a particular sharing strategy influences the speed and consistency of the application. We have presented a prototype groupware toolkit called GEN which uses an open implementation to let application developers control the way in which their data is shared.

Groupware is pushing our concept of what computers can do into new areas. We now view computers as a sophisticated communication medium as well as a tool. As these new paradigms evolve, limitations of the models we use to construct software are exposed, forcing computer scientists to rethink the strategies they use. Open implementation is an exciting new strategy that makes developers reconsider the notion of abstracting away implementation details in black boxes. By exposing the internals of the black box in a clear and careful way, we can give developers the power they need to construct applications in these new paradigms.

Bibliography

Abelson, H., Sussman, G. & Sussman, J. (1986). *The Structure and Interpretation of Programs*. MIT Press, Cambridge Massachusetts.

Ahuja, S.R., Ensor, J.R. & Lucco, S.E. (1990). A Comparison of Applications Sharing Mechanisms in Real-time Desktop Conferencing Systems. In *Proceedings of the ACM COIS Conference on Office Information Systems*, 238-248.

Anupam, V. & Baja, C. (1993). Collaborative Multimedia Scientific Design in SHASTRA. In *Proceedings of Multimedia '93*, 447-456.

Arango, M., Bahler, L., Bates, L., Cochinwala, M., Cohors, D., Fish, R., Gopal, G., Griffeth, N., Herman, G., Hickey, T., Lee, K., Leland, W., Lowrey, C., Mak, V., Patterson, I., Ruston, L., Segal, M., Sekar, R., Vecci, A., Weinrib, A. & Wuu, S. (1993). The Touring Machine System. *Communications of the ACM*, 36, 68-77.

Attardi, G. (1993). Metaobject programming in CLOS. In Paepcke (ed.), *Object Oriented Programming: The CLOS Perspective*, MIT Press, Cambridge Mass, 119-132.

Barghouti, N. & Kaiser, G. (1991). Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3), 269-317.

Bennet, J (1990). Experience With Distributed Smalltalk. *Software Practice and Experience*, 20(2), 157-180.

Bentley, R., Rodden, T, Sawyer, P. & Somerville, I. (1994). Architectural Support for Cooperative Mulituser Interfaces. *IEEE Computer*, 27(5), 37-45.

Bonfiglio, A., Malatesta, G. & Tisato, F. (1989). Conference Toolkit: A Framework for Real-time Conferencing. In *Proceedings of the EC-CSCW '89 First European Conference on Computer Supported Cooperative Work*, 303-316.

Cortes, M. (1994). CSCW Survey: Concepts, Applications and Programming Tools. Department of Computer Science, State University of New York, Stony Brook.

Craighall, E., Lang, R., Fong, M. & Skinner, K. (1993). CECED: A System for Informal Collaboration. In *Proceedings of Multimedia '93*, 437-445

Crowley, T., Baker, E., Forsdick, H., Milazzo, P. & Tomlinson, R. (1990). MMConf: An Infrastructure for Building Shared Applications. In *Proceedings of the CSCW'90 Conference on Computer-Supported Cooperative Work*, 329-342.

Decouchante, D (1986). Design of a Distributed Object Manager for the Smalltalk-80 System. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems Languages and Applications*, 444-452.

Dollimore, J, Miranda, E. & Xu, W. (1991). *The design of a System for Distributing Shared Objects*. The Computer Journal, 34(6), 514-521.

Dourish, P. (1995a). Developing a Reflective Model of Collaborative Systems. *ACM Transactions on Computer-Human Interaction*, 2(1), 40-63.

Dourish, P. (1995b). The Parting of Ways: Divergence, Data Management and Collaborative Work. *In Proceedings of the Fourth European Conference on Computer-Support Cooperative Work*, 215-230.

Dourish, P. (1996). Open Implementation and Flexibility in CSCW Toolkits. *Ph.D. Dissertation*, Department of Computer Science, University of London.

Ellis, C.A. and Gibbs, S.J. (1989). Concurrency Control in Groupware Systems. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 399-407.

Gamma, E. Helm, R., Johnson, R. & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Reading Massachusetts.

Goldberg, A, & Robson, D (1983). *Smalltalk-80 the Language and its Implementation.* Addison Wesley, Reading Massachusetts.

Graham, T., Urnes, T., & Nejabi, R. (1996a). Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware. In *Proceedings of the ACM*

Symposium on User Interface Software and Technology (UIST'96), ACM Press, (in press).

Graham, T.C.N. & Urnes, T. (1996b). Linguistic Support for the Evolutionary Design of Software Architectures. In *Proceedings of the ICSE'18 Eighteenth International Conference on Software Engineering*, IEEE Press, 418-427.

Greenberg, S. & Marwood, D. (1994). Real-time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface. *In Proceedings of the ACM CSCW'94 Conference on Computer Supported Cooperative* Work, 207-217.

Greenberg, S. & Roseman, M. (1997). Groupware Toolkits for Synchronous. In Beaudouin-Lafon (Ed.), *Trends in CSCW*, John Wiley & Sons. Forthcoming.

Greenberg, S. Roseman, M, Webster, D., Bohnet, R. (1992). Human and Technical Factors of Distributed GroupDrawing Tools. *Interacting with Computers*, 4(3), 364-392.

Hill, R.D. (1992). The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications. In *Proceedings of the ACM SIGCHI'92 Conference on Human Factors in Computing Systems*, 335-342.

Hill, R.D., Brinck, T., Rohall, S.L., Patterson, J.F. & Wilner, W. (1994). The Rendezvous Architecture and Language for Constructing Multi-user Applications. *ACM Transactions on Computer-Human Interaction*, 1(2), 81-125.

Hutchinson, N., Raj, R., Black, A., Levy, H. & Jul, E. (1987). The Emerald Programming Language. *DIKU Report 87/22, Department of Computer Science*, University of Copenhagen, Denmark.

Jahn, P. (1995) Getting started with Share-Kit. *Tutorial manual distributed with Share-Kit version 2.0. Communications and Operating Systems Research Group, Department of Computer Science*, Technische Universitat, Berlin, Germany. Available via anonymous ftp from ftp.inf.fu-berlin.de:/pub/misc/share-kit.

Kiczales, G. (1992). Towards a New Model of Abstraction in Software Engineering. In *Proceedings of IMSA '92 Worksop on Reflection and Metalevel Architectures*, 1-11.

Kiczales, G. Ashley, J., Rodriguez, L., Vahdat, A. & Bobrow, D. (1993). Metaobject Protocols: Why We Want Them and What Else They Can Do. In Paepcke (ed.), *Object Oriented Programming: The CLOS Perspective*, MIT Press, Cambridge Mass, 101-118.

Kiczales, G., DeLine, R., Lea, A. & Maeda, C. (1995). Open Implementations Analysis and Design. (Tutorial Notes), *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications OOPSLA '95.*

Kiczales, G., des Rivieres, J. & Bobrow, D. (1991). *The Art of the Meta-object Protocol*, MIT Press, Cambridge Mass.

Kittlitz, K. (1994). Approaches to Object-Oriented Concurrency Control. *Masters Thesis.* Department of Computer Science, University of Calgary.

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

Moran, T., McCall, K., van Melle, B., Pedersen, E. & Halasz, F. (1995). Some Design Principles For Sharing in Tivoli, a Whiteboard Meeting Support Tool. In S Greenberg, S. Hayne & Roy Rada (eds.) *Groupware For Real-time Drawing: A Designers Guide*. McGraw-Hill, London, 24-36.

Nascimento, C. & Dollimore, J. (1992). Behaviour Maintenance of Migrating Objects in a Distributed Object-Oriented Environment. *Journal of Object Oriented Programming*, 5(5).

O'Grady, T. and Greenberg, S. (1994). A Groupware Environment For Complete Meetings. *In ACM SIGCHI Conference on Human Factors in Computing Systems Conference Companion Proceedings*, 307-308.

Paepcke, A. (1993). User-level Language Crafting: Introducing the CLOS Metaobject Protocol. In Paepcke (ed.), *Object Oriented Programming: The CLOS Perspective*, MIT Press, Cambridge Mass, 65-99.

Parrington, G., Shrivasta, S. Wheater, S. & Little, M. (1995). The Design and Implementation of Arjuna. *Department of Computing Science, The University of Newcastle upon Tyne*, Newcastle upon Tyne.

Patterson, J. F., Hill, R. D., Rohall, S. L., & Meeks, W. S. (1990). Rendezvous: An Architecture for Synchronous Multi-user Applications. In *Proceedings of the CSCW'90 Conference on Computer Supported Cooperative Work*, 317-328.

Patterson, J.F., Day, M. & Kucan, J. (1996). Notification Servers for Synchronous Groupware. *Lotus Development Corporation. In submission to the ACM CSCW Conference on Computer Supported Cooperative Work.*

Rao, R. (1993). The Silica Window System,: The Metalevel Approach Applied more Widely. In Paepcke (ed.), *Object Oriented Programming: The CLOS Perspective*, MIT Press, Cambridge Mass, 133-154.

Roseman, M. (1993). Design of a Realtime Groupware Toolkit. *Masters Thesis, Department of Computer Science*, University of Calgary.

Roseman, M. (1995). When is an object not an object? *Proceedings of the 1995 Tcl/Tk Workshop.*

Roseman, M. & Greenberg, S. (1992). GroupKit: A groupware toolkit for building real-time conferencing applications. In *Proceedings of the ACM CSCW'92 Conference on Computer Supported Cooperative Work*, p43-50.

Roseman, M. & Greenberg, S. (1993). Building Flexible Groupware Through Open Protocols. In *Proceedings of the ACM COOCS'93 Conference on Organizational Computing Systems*, p279-288.

Roseman, M. & Greenberg, S. (1994). Registration for Real-time Groupware. *Research Report 94/533/02, Department of Computer Science*, University of Calgary, Alberta, Canada.

Smith, B.C. (1982). Reflection and Semantics in a Procedural Language, *MIT Laboratory for Computer Science Report MIT-TR-272*, Cambridge, Mass.

Steele, D. (1991). Distributed Object Oriented Programming: Mechanism and Experience. *In Proceeding of Tools USA*.

Stefik, M. Bobrow, D. Foster, G. Lanning, S. & Tatar, D. (1987). WYSIWIS Revised: Early Experiences with Multi-user Interfaces. *ACM Transactions on Office Information Systems*, 5(2), 147-167.

Stroud, R. & Wu, Z (1995). Using Meta-object Protocols to Implement Atomic Data Types. In *Proceeding of the European Conference On Object-Oriented Programming*.

Tou, I., Berson, S., Estrin, G., Eterovic, Y. & Wu, E. (1994). Prototyping Synchronous Group Applications. *IEEE Computer*, 27(5), p48-56.

Wilson, B. (1995). WSCRAWL 2.0: A Shared Whiteboard Based on X-Windows. In S Greenberg, S. Hayne & R. Rada (eds.), *Groupware for Real Time Drawing, A Designer's Guide*, 130-142.