# Linguistic Support For Developing Groupware Systems

by

ROY NEJABI

B.Sc., York University (1993)

A Thesis submitted to the

Faculty of Graduate Studies, York University

in partial fulfillment of the requirements

for the degree of

Master of Science

Thesis Supervisor: T.C.N. Graham

Graduate Programme in Computer Science

Department of Computer Science, York University

4700 Keele Street, North York, Ontario

Canada, M3J 1P3



(July 1995)

September 15, 1995

# Abstract

Groupware refers to computer based systems which are explicitly designed to support groups of people working together. Synchronous groupware systems, the form of groupware which this thesis aims to support, allow several users to work simultaneously on the same information. This class of applications is difficult to develop. The difficulty is due to the involvement of distribution, networking and interaction technologies in the development of this class of applications. This thesis argues that groupware development is sufficiently difficult to justify a special purpose language. This thesis presents a set of requirements which facilitate the development of groupware. These requirements are classified into two inter-related groups: requirements for expressiveness, such as the need to support the development of a rich set of collaboration styles, and requirements for ease of use, such as the need for a transparent communication infrastructure. These requirements are used as the criteria for the development of a prototype toolkit: Multi-User Clock. This toolkit provides a special purpose programming language for the construction of groupware and multi-user user interfaces. This thesis shows how Multi-User Clock can be used to develop a wide range of groupware systems, and how this toolkit satisfies many of the identified requirements.

# Acknowledgments

I would like to thank all the members of the supervisory committee: Nick Graham, Tim Brecht, Tom Papadakis, and Richard Irving, for their invaluable comments and inputs. Their efforts have certainly contributed to the quality of this thesis, and for that I am thankful. I would like to express my particular gratitude to my supervisor, Nick Graham, without whose efforts this thesis would not be possible. I am very grateful for his numerous discussions and readiness to provide the assistance when I needed.

I am grateful to Katherine Figuracion for all her assistance and support during this thesis and her efforts for proof-reading numerous thesis versions. I also would like to thank my many friends and colleagues at York, for their moral support and friendship. Special thanks to Tore Urnes (for letting me borrowing books from his private collection and keeping them for months), Ragab Omran (for many intellectual discussions and many basketball games), Kaushik Guha (for sharing many laughs and squash games), and Jason Tsui (for his readiness to help at all times).

Lastly, but not least, I feel indebted to all the people from whom I have learned a lesson throughout my life, and dedicate this work to them.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Overview

*Groupware* refers to software systems which are designed to assist groups of people working together. This is in contrast with most traditional software, which only supports single-user interaction. The research discipline which studies groupware, as well as the wider context in which the groupware is used, is referred to as *Computer Supported Cooperative Work*, or *CSCW* [Ellis 91]. This field has been the focus of much research in recent years, resulting in the development of a number of groupware toolkits. Groupware toolkits provide a set of tools to facilitate the development of groupware systems. Although these tools have led to great progress in the development of groupware systems, there remain unsolved problems. These problems include providing transparent distribution and communication mechanisms.

This thesis describes the design and implementation of a groupware toolkit: *Multi-User Clock*. This toolkit is an extension of single-user Clock which was developed by Graham [Graham 94]. Multi-User Clock addresses several problems associated with groupware development, including transparent distribution of application parts, communication between these parts, and controlling concurrent activities in various sites.

This chapter begins by presenting an overview of *synchronous groupware systems*, the form of groupware which Multi-User Clock is intended to support. It then provides a brief description of the motivation for this work and the contributions of this thesis. This chapter concludes with an outline of the rest of the thesis.

## 1.2 An Introduction to Groupware Systems

Groupware falls into a relatively new research field called Computer Supported Cooperative Work (CSCW). This field is an interdisciplinary research area focused on the role of computer and communication technology to support group work. Groupware refers to computer based systems which are explicitly designed to support groups of people working together. Although the roots of groupware can be traced to the late 1960s, the systems have only recently begun to proliferate [Roseman 93a]. Examples of groupware systems include desk top conferencing systems, shared editing and drawing applications and electronic mail applications.

Groupware systems are often divided into four categories, as shown in Figure 1.1 [Ellis 91]. This taxonomy is based on two dimensions: time and location. Groupware systems can support users working at the same time (*Synchronous*) or at different times (*Asynchronous*), as well as users working in the same physical location (*Co-located*) or at a distance (*Distributed*).

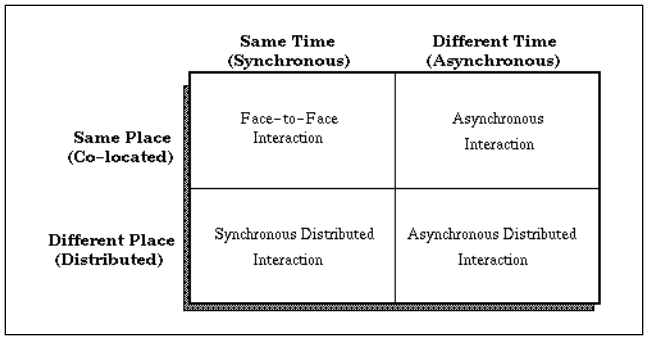



**FIGURE 1.1 A time space taxonomy of groupware applications from [Ellis 91]**

Asynchronous groupware applications support collaboration that is spread in the time axis. Examples of asynchronous groupware applications include electronic mail and systems implementing organizational memory (e.g., Lotus Notes [Lotus 93]). Synchronous groupware systems, on the other hand, support collaboration which is performed in real-

time. Synchronous groupware, which is often referred to as desktop conferencing, includes examples such as collaborative writing/drawing/design tools and group decision support systems. Much of the recent research in groupware has been focused on real-time synchronous systems. While most groupware toolkits should be developed to support both synchronous and asynchronous work, the problems associated with building the two classes of systems are sufficiently diverse that providing the necessary underlying support for both would be a large undertaking. Hence, we have chosen to limit our research to synchronous groupware systems. The term groupware is hereafter limited to synchronous groupware applications.

## 1.3 Thesis Motivation and Contributions

Groupware applications are difficult to develop. There are significant challenges in the design and the development of groupware systems that typically do not arise in other applications [Urnes 94]. These difficulties are due to the basic nature of this class of applications, which is to support concurrent collaborative activities amongst a number of distributed users. As a result, developers of groupware systems must consider technical issues such as communication between the distributed parts of the application, controlling concurrent activities, and synchronization of multiple input and output events. They also must address issues which concern group interactions and provide a medium through which the collaboration is conveyed. Finally, all difficulties associated with single-user systems remain present in the development of groupware systems. These issues place many obstacles in the progress of groupware development. These obstacles deflect the developers from their main goal, which is to design and develop a successful groupware system.

Applications with special needs, such as groupware, can benefit from toolkit support. Toolkits are designed to relieve the developers from many low-level details which can impede the development process. Software toolkits provide a set of components with gen-

eral functionality, leaving only application specific functionality to be coded by the developer. This not only saves programing time, but can also increase the quality of the resulting software.

Several toolkits have been developed in an attempt to address some of the difficulties involved in developing groupware. The support of these tools is often in the form of library of components and functions, added to an existing single-user toolkit. As a result, the extent to which these tools can help is limited to the programming language underlying the toolkit. The diversity and complexity of requirements which groupware toolkits need to address justifies the need for a special purpose programming language.

This thesis describes the requirements for groupware toolkits and how they are addressed in Multi-User Clock. Multi-User Clock is a toolkit, with a special purpose language, for developing multi-user interfaces and groupware systems. This toolkit addresses several unsolved problems in groupware development, including issues concerning application distribution, networking, communication, and conflicts between concurrent processes. In Multi-User Clock these issues are addressed by providing a mechanism for the transparent distribution of the application architecture and data, a built-in communication infrastructure (including transparent event handling between the distributed applications parts) and a transparent concurrency control mechanism. These facilities are intended to abstract the inherent problems in distributed systems, giving the developers the illusion of developing single-user applications. Multi-User Clock also provides support for development of a customized user-interface for each participant of a multi-user application. Furthermore, Multi-User Clock provides a transparent constraint maintenance mechanism which maintains consistency amongst user-interfaces of distributed users.

In short, the goal of Multi-User Clock is to ease groupware development using declarative techniques. These methods allow the developers to specify ‘what’ they want to achieve

rather than ‘how’ to achieve it. This will allow the developers to focus their efforts on the main task: developing a successful groupware system.

Many factors contribute to the success of a toolkit. First, the toolkit must provide the necessary components and functionality to facilitate the construction of applications in the target domain. Second, it must provide specialized support for the generic problems inherent to the application domain. Third, it should be flexible to allow simple tasks to be done quickly, and provide sufficient support so developers can design more complex systems.

The Multi-User Clock programming environment meets these objectives by providing sufficient expressive power while maintaining ease of use. This not only facilitates the development process, which leads to desired application, but can also expedite it.

Multi-User Clock is designed and developed as part of this thesis. It is currently a prototyping tool and has poor performance. This toolkit is intended to be used only by computer science professionals for research purposes.

## 1.4 Thesis Outline

This thesis describes and evaluates the design and implementation of Multi-User Clock, a tool for the development of groupware and multi-user user interfaces.

Chapter 2, “Related Work”, presents the characteristics and features of groupware systems as well as the requirements which groupware toolkits need to address in order to support these features. This chapter also surveys a number of existing groupware toolkits and examines them in terms of the listed criteria.

Chapter 3, “An Introduction to Clock”, provides an overview of the Multi-User Clock-Multi-User Clock language and its features.

Chapter 4, “Language Design” describes the selected requirements which are addressed in Multi-User Clock.

Chapter 5, “Language Implementation”, describes the implementation techniques which have been adapted to satisfy the design criteria outlined in Chapter 3.

Chapter 6, “Summary and Conclusion”, summarizes the contributions of this research and presents several areas for future work.

# Chapter 2

# Related Work

## 2.1 Overview

Chapter 1 presented a description of groupware systems and described the need for groupware toolkits. This chapter begins with a discussion of characteristics specific to groupware and the underlying technologies which support these characteristics. We then identify the groupware requirements which have guided the design of Multi-User Clock. Finally, a survey of existing groupware toolkits is presented in order to familiarize the reader with other studies in groupware systems and toolkits.

## 2.2 Features of Groupware Applications

This section identifies and presents the characteristics of groupware systems. The purpose of this presentation is to provide an in-depth understanding of groupware systems by studying their behaviour. This also provides a basis for determining some of the features of groupware that are important to its success. These features will later be viewed as design requirements for a groupware toolkit.

Before opening our discussion, it would be appropriate to introduce some of the common terminology in the groupware domain. To clarify this terminology, we present a simple groupware system, a shared drawing program which was designed and implemented using the Multi-User Clock toolkit. Figure 2.1 illustrates two windows which are generated by this drawing program.


**FIGURE 2.1 An example of a Groupware application: A shared drawing program. The figure illustrates two windows, each of which belongs to one user of the program. The objects drawn by each user are depicted on all users' displays.**

This simple drawing program demonstrates many aspects of groupware systems. Each user can invoke the program from a different workstation connected to a different network. An invocation of a groupware system is termed, informally, a *session.* A session consists of a group of users called *participants* and provides each participant with an interface to a shared context. For instance, participants may see synchronized views of evolving objects on the drawing canvas. The drawing program provides a WYSIWIS (What-You-See-Is-What-I-See) interface to the shared drawing. This means that, at any given time, interfaces of all users portray the same images. In our drawing program, a colour scheme is used to differentiate objects drawn by different users. Other forms of interfaces are also possible, including a WYSINWIS (What-You-See-Is-NOT-What-I-See) interface, in which each participant's interface portrays different images.

The system's *response time* (or *feedback time*) is the time necessary for the actions of one user to be reflected by their own interface. Consider an example in which two users cooperatively draw an object, in this case drawing the word "Hi!". Each user draws a line by holding the mouse button and dragging it towards the final point. In this example, the response time is calculated from the time the user drags the mouse until the time the line is drawn on the screen. The *notification time* (or *feed-through time)* is the time necessary for one user's actions to be propagated to the remaining users' interfaces. In the above exam-

ple this is the time necessary for the line to be drawn on the other participant's screen. The notification *granularity* may vary. The notification granularity is the frequency of (i.e., how often) reflecting one user's action on other participants' interfaces. In our case, the interfaces of other users are notified after an object (line/box) is completely drawn on the original screen. We could also allow all participants to view all the intermediate steps (i.e., drawing rubber band objects) involved in drawing an object. This would result in a finer granularity in notification.

The remainder of this section presents the main features of groupware systems. The first two features of groupware are shared by all multi-user systems. The last feature, however, is exclusive to groupware systems. The main characteristics of groupware are:

- *Supports Distributed Users:* Groupware applications are inherently multi-user systems. The users of these systems can be either temporally or geographically spread. In our work we are mainly concerned with synchronous groupware systems. These systems are designed to support geographically spread users. Although this class of applications supports both co-located and distributed users, much of the research in synchronous groupware has focused on systems which support geographically spread users. This is a reasonable approach since most groupware systems designed for distributed users can also be used by co-located users.

  The distributed nature of synchronous groupware suggests that in general, one cannot assume that the participants of a groupware application are all connected to the same machine, or even to the same local area network. During an active session, the participants are free to join or leave the session at any time, thus making the sessions of the synchronous groupware systems volatile.

- *Supports Information Sharing:* The sharing of information has a central role to play within cooperative applications. In order to support collaboration, groupware applications must permit users to share data. In groupware applications, access to shared data follows an ad-hoc method. This means that generally the participants do not follow a

pre-planned script, making it impossible to predict what information will be accessed in the future. Due to the high level of concurrent activities during a session, there is a high degree of access conflict as participants work on and modify the shared data.

- *Supports Collaboration/Coordination:* Typical multi-user applications (e.g., distributed databases) strive to provide a *collaboration-transparent* environment. That is, they provide the illusion to each user that she/he is the sole user of the system. Conversely, groupware applications provide a *collaboration-aware* working environment where users' actions are reflected on other participants' work space, providing an awareness of the other users' activities.

  To achieve collaboration, groupware systems must be highly interactive, with short response to user actions. Since participants' actions are based on other participants' actions, the real-time notification times must also be short.

After identifying the major characteristics of groupware systems, we are now in a position to look at the technologies which support these features. The following section discusses the technologies which enable groupware systems to offer the features introduced in this section.

## 2.3 Enabling Technologies in Groupware Development

This section identifies the technologies which are involved in the development of groupware systems. A successful synchronous groupware application involves expertise from various technical domains. Groupware relies on the approaches and contributions of the following disciplines [Baecker 93, Urnes 94]:

- *Interaction Technologies:* Groupware facilitates collaboration by employing computing facilities to support interaction between groups of users. Interaction technologies include those which are involved in the support of human-human and human-computer interaction. Typical examples are direct manipulation, window systems, I/O devices, discrete (e.g., text) and continuous (e.g., audio/video) media, and work space management.

- *Distribution Technologies:* Since groupware applications are intended to provide collaboration between distributed users, they are inherently distributed systems. Distribution technologies provide variation in architectures and approaches to consistency management. The architecture of a groupware application is determined by how the application is distributed among a number of geographically spread workstations. Consistency management techniques provide control mechanisms for accessing the shared data as well as ensuring concurrency control.

- *Communication & Networking Technologies:* These technologies provide support for the transportation of data between different hardware components. In particular these technologies resolve such concerns as communication channels, network capacity, connectivity and communication protocols.

Other researchers [Ellis 91, Baecker 93] suggest expanding the above set by including other disciplines such as artificial intelligence and multi-media technologies. These disciplines are important in contributing to the development of many groupware systems (e.g., video conferencing). This contribution will become more apparent with the emergence of future classes of groupware systems. However, for the purposes of this research, we will limit our investigation to the above three domains. Incorporating other technologies may be an interesting avenue for future research.

## 2.4 Requirements for Groupware Toolkits

Section 2.2 identified a set of characteristics typical of groupware applications. Based on these characteristics, this section identifies a set of requirements which must be addressed by a successful groupware toolkit. These requirements result from our experience developing and using various groupware toolkits. Although some of these requirements may not be exclusive to groupware development toolkits, nonetheless they represent fundamental criteria which are crucial in the success of any user interface toolkit.

We organize these requirements into two inter-related groups: requirements for *expressiveness* and requirements for *ease of use*. The requirements for expressiveness are those which must be addressed in order to allow the developers to construct the desired application. The requirements for ease of use, however, are those which groupware developers can benefit from during the development of a groupware application. These requirements, if addressed, will help the developers to design better groupware systems.

### 2.4.1 Requirements for Expressiveness

In this section we review the requirements for expressiveness. These requirements, if addressed, facilitate the development of any desired groupware system and contribute to the quality of the end product. The quality of a groupware system can be loosely defined as how accurately the application addresses the needs of the participants. This can determine a groupware's success. Based on our research and experience, we believe that a groupware toolkit should offer:

- *Support for Multiple Group Activities:* Groups exhibit a variety of behavioural characteristics as part of their interaction. These include cohesion, commitment, relationship with the organization, and use of time [Mandavida 94]. To support these characteristics, groupware applications must provide support for a variety of collaborative activities.
- *Support for Interchangeable Collaboration Methods:* Group studies suggest that collaboration is the result of a mixture of interaction methods such as: face-to-face meetings, verbal, written, and non-verbal cues [Ishii 94]. The interchangeable collaboration requirements stem from the need to accommodate group members as they switch back and forth between the different interaction methods. Groupware systems should facilitate "seamless" switching between different interaction functions [Ishii 94].
- *Support for Customization*: A particular group has detailed knowledge about itself and its needs. This knowledge can be used to customize the groupware application which is used by the group. Hence, groupware systems should be adaptable to individual and overall group needs.

- *Support for Persistent Sessions:* Groups need a record of past activities for future planning and growth [Mandavida 94]. A persistent session can behave as a group memory. Information kept in the group memory will enable the current, as well as later, groups to understand their development, learn from the past mistakes, and avoid session discontinuation.

- *Support for Error Independence:* A local application error, that is the execution of an erroneous command whose results are not shared with other users (and which may result in the loss of the connection) should not cause failures in connections between other users and the program.

- *Support for Developing Good User-Interfaces:* The user-interface of a groupware system not only provides a communication medium between the users and the system, but also provides a channel through which the collaboration is conveyed to the participants. This dual functionality places a burden on the design of groupware user-interfaces. As a result, all the requirements necessary for a good user-interface apply to groupware applications. The requirements for a good user-interface include:

  *Direct Manipulation:* This refers to the manipulation of objects (e.g., buttons and menus) in a user-interface in order to cause an action in the underlying application. *Event driven* programming is a common technique for implementing direct manipulation. In this class of programs, unlike the conventional programs in which the user must respond to prompts from the program, a user can initiate one or more dialogues with the application at any given time. Furthermore, the user can switch between different dialogues at will. This implies that the application must be capable of handling user inputs in a non-deterministic order.

  *Semantic Feedback:* In order to alleviate the cost incurred by iterative refinement, it is usually considered beneficial to separate the user-interface from the underlying application. In order to allow the user-interface to correctly reflect the state of the application's data, it needs to constantly poll the underlying application in order to

acquire up-to-date information. This implies that user-interface systems must provide fast access to application data. This often conflicts with the clean separation of the user-interface and the application.

*User-Interface Consistency:* The components of a user-interface should appear and behave in the same way throughout the user-interface. Also, similar problems (e.g., selection) should be solved in similar ways. The user-interface tool must provide support for maintaining consistency in user-interfaces.

- *Good Performance:* Performance is one of the major factors in the success of a graphical user-interface (GUI) with respect to user acceptance. A GUI application with poor performance will lead to user frustration and eventual lack of use or re-use. In a multi-user groupware setting, performance plays a more critical role. Since groupware applications are cooperative and support collaboration, the input of one user is greatly dependent on the input of other users. Hence, users of such systems constantly need to be informed of other participants' actions within a reasonable feed-through time. Reasonable timing, in this case, can be defined as the acceptable latency which does not degrade or interfere with collaborative activities among participants. A groupware application with good performance not only provides rapid feedback but also offers rapid feed-through.

The list of requirements presented in this section is not complete. This incompleteness is partly due to the fact that many aspects of group requirements still remain unknown. Many researchers, from various interest groups, are currently studying groupware and its development. The outcome of these studies will undoubtedly place additional requirements into the design of groupware systems.

### 2.4.2 Requirements for Ease of Use

This section presents the requirements for ease of use. These requirements facilitate the development of groupware systems. Although it may be argued that many groupware requirements can be satisfied even in the absence of these tools, having these facilities

available will enable the developers to provide better groupware systems faster. We believe that in order for a groupware toolkit to meet the level of support which developers require, it must provide:

- *A High-level Declarative Programming Language:* Many aspects of a groupware application are directly or indirectly related to the programming language which is used to define it. The programming language that a groupware toolkit offers must be simple and easy to learn. The shorter the learning curve, the more likely it will be adopted by programmers. It is generally desirable to have as high a level of specification as possible. To achieve this, we need a declarative specification language. Declarative languages allow programmers to think at a much higher level than traditional imperative programming languages. This feature enables programmers to concentrate on "what" they want, rather than, on "how" to make it happen.

- *Support for Iterative Refinement:* Groupware design, like user-interface design, requires user feedback. User involvement in the design process validates the predefined groupware requirements and may establish new design elements. Iterative refinement refers to the process of iterating between design, implementation, user testing, and redesign. The difficulties in building and modifying groupware makes iterative refinement an expensive technique. Hence, a groupware toolkit must provide support for the easy creation and modification of user-interfaces.

  Groupware developers need to consider a number of technical, psychological and sociological issues. This makes groupware development an experimental task which requires user feedback. To aid this experimental process, support for *rapid prototyping* is needed. Rapid prototyping allows the groupware developer to perform usability evaluations by deploying working prototypes at early stages, thus providing a basis for making early critical decisions about the application being developed. Rapid prototyping facilitates the process of iterative refinement and can be partially supported by providing a high-level programming language.

Support for iterative refinement also facilitates *incremental development*. Using this method, each part of an application can be designed and implemented individually. Thus, the developer can focus on the functionality of one part of the application at a time. The well-designed and implemented application parts can later be put together to construct the main application.

- *Support for the Customization of User-Interfaces:* A groupware toolkit should provide facilities for the development of applications in which different users of the program have different user-interfaces. In particular, it should support the development of applications in which the users: (i) have different display images; (ii) have different access rights and supporting roles; (iii) interact using different hardware, thereby allowing users with different workstations and file systems to collaborate with each other.

- *Support for Flexible Placement:* In a distributed multi-user environment, programmers should have the flexibility of choosing where the data and modules should reside. In particular, they should have the flexibility of placing the data either on the local workstation, for faster local response, or on the server machine [Dewan 92b].

- *Support for Compatibility:* A multi-user toolkit should allow single-user programs developed by the tool to be transformed to the multi-user ones. This will promote reusability within the application domain in which the toolkit is being used.

- *Support for Session Managemen*t: An invocation of a groupware system is termed a session. A groupware toolkit must offer support for creating a session, allowing participants to enter and leave a session, and finally terminating a session. Performing each of these tasks involves a number of technical issues which must be handled in a transparent fashion, without imposing an additional burden on the programmer.

- *Support for Transparent Distribution of Application and Data:* The components of a multi-user groupware application are distributed across a number of remote machines. A distributed architcture may be designed in one of three ways: centralized, replicated, or semi-replicated. In a centralized architecture, all messages are forwarded to the central machine which in turn, forwards messages to other participants. In a replicated

architcture, no central machine is used and replicated processes communicate directly with each other. A semi-replicated architecture is a hybrid between centralized and replicated architectures. There are a number of trade-offs between the three architectures. Chapter 5 presents a brief study of each of the three architectures. Groupware toolkits must be able to distribute the application and the data among the distributed workstations in a transparent fashion.

- *Support for a Transparent Communication Infrastructure:* It is essential for the distributed application parts to communicate. These distributed parts must be capable of sending and receiving messages to and from one another without concern for low-level communication issues. Hence, a groupware toolkit must not only provide support for establishment and maintenance of communication channels between the remote workstations, but it also must be capable of handling communication between the distributed application parts in a transparent fashion.

- *Support for Concurrency Control:* Groupware tools must provide support for handling concurrent activities. Concurrency control is needed within groupware systems to help resolve conflicts between participants, and to allow them to perform tightly coupled group activities. Coupling, in the context of groupware, refers to the degree which one user's interface state is dependant on other users' actions [Dewan 93]. The overhead which is associated with most concurrency control mechanisms typically has an impact on the system's performance. Groupware tools must provide support for handling concurrent activities without sacrificing the ease of programming or performance.

- *Support for Collaboration Awareness:* The toolkits should also support the development of collaboration-aware programs, thereby allowing programs to exercise control over the way their collaborative tasks are performed.

We shall use the above criteria as a guideline for our survey of existing toolkits, and as a blueprint for our tool development.

## 2.5 An Overview of Some Existing Groupware Toolkits

In this section we provide a brief survey of the related work which is being conducted in the area of groupware toolkits. The goal is to provide the reader with a broader understanding of different groupware toolkits, and familiarize her/him with various research foci in this area. We also intend to evaluate these tools based on the criteria which are detailed in Section 2.4. This evaluation may not present a fair comparison as each of the surveyed toolkits was designed and developed to meet different goals. Rather, the intent is to highlight the fact that current toolkits give limited attention to the programming aspect of the groupware systems.

All of the examined systems (except for the Rendezvous and the Notes systems) are non-commercial systems resulting from research efforts in academia. The order of presentation is alphabetical.

## 2.6 The GroupKit System

GroupKit [Roseman 92, 93a, 93b] is built on top of Tcl/Tk [Ousterhout 94] by researchers at the University of Calgary, Canada. It extends an existing toolkit, Tcl/Tk, for developing single-user graphical user-interfaces. The developers of the GroupKit system argue that much research effort in recent years concentrates on addressing problems which arise when porting a single-user application to a multi-user one. Hence, in their effort to address some of the CSCW related problems, the developers of GroupKit have placed much of their effort into providing support for session management and multi-user widgets. While these issues are important, the GroupKit developers did not address some other technical issues such as concurrency control and communications between the application parts. Figure 2.3 shows an schematic view of the GroupKit system. As is shown in the figure, each user (client) interacts with its own application replica. Also, users also interact with the local *Registrar Client* process, in order to establish or terminate a session. The *Central Registrar* maintains an up-to-date list of all active sessions.

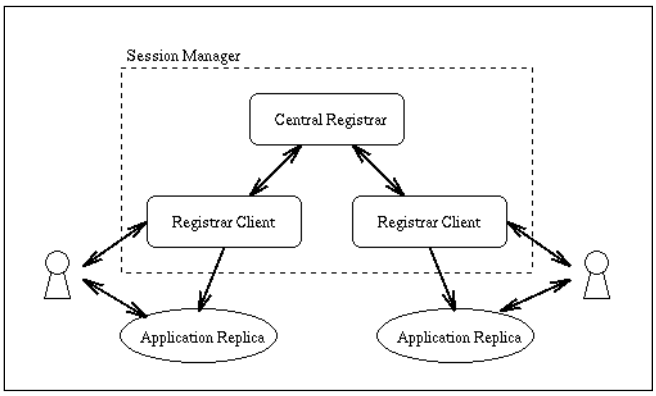



**FIGURE 2.2 The components and the data flow between these components in GroupKit applications.**

The main features or advantages of the GroupKit toolkit are:

- GroupKit provides a set of multi-user widgets, which are designed to address the special user-interface needs of the groupware applications. These widgets are an addition to the primary Tk [Ousterhout 94] widgets (i.e., buttons, scrollbar, menus and canvas). The two multi-user widgets provided by the GroupKit are a “vertical remote scrollbar” and a remote cursor [Roseman 92].
- The GroupKit session manager implementation is based on open protocols which facilitate flexibility in realizing different session management policies. The possible session management policies include: *Open Registration* (in which the session manager provides a permissive policy of creating and joining conferences between equal status participants), and *Centrally Facilitated* (which is designed for structured meetings controlled by a facilitator).

Some of the shortcomings of the GroupKit system are:

- Groupware applications, which are generated by GroupKit, have a replicated architecture. There is no built-in support for consistency control. Hence, in order to keep the data consistent amongst all replicas, the programmer needs to multicast all the update commands performed on the shared data.
- Lack of support for restricting user access to the shared data.

- Lack of support for providing WYSINWIS views.

### 2.6.1 Programming Environment of the GroupKit System

The GroupKit programming environment is based on Tcl/Tk and its Remote Procedure Call (RPC) extension. Tcl/Tk provides a simple interface to the X window system, allowing the programmer to write more concise programs than with the X/Motif toolkit. The built-in RPC in Tcl provides simple mechanisms for establishing connections, sending and receiving data, and closing connections. These calls are used for inter-process communication (IPC) among collaborative processes. Although Tcl is easy to learn and simple to use, it falls short of being a full-fledged programming language. For instance, it does not support any complex data structures, such as arrays or linked lists. Any complex program must be written in an auxiliary language (e.g., C) and linked into a Tcl script.

GroupKit does not provide any explicit support for concurrency control. The toolkit runtime support provides a broadcast mechanism which can be used by client applications to deliver any updates (on the shared data) to the peer client applications. This broadcast function, however, is unaware of any simultaneous update on the same data. Hence, it is the programmer's responsibility to create client programs that coordinate and synchronize peer applications. Except for the broadcasting functions that can be used in any application, there are no data flow features built into this toolkit. GroupKit does not provide new shared data types above Tcl/Tk data types. Similarly, debugging options of the toolkit are inherited from the Tcl interpreter.

### 2.6.2 Architecture of the GroupKit System

The architecture of the groupware applications, which are built with GroupKit, are composed of three different parts: a centralized name server (called the *registrar*), a decentralized conference manager (called the *registrar client*) and synchronized client applications. For a given shared environment, only one centralized name server is needed. However,

there should be one decentralized conference manager available for each collaborative user. The centralized name server and the decentralized conference managers are in constant communication with each other in order to maintain an up-to-date list of all active users as they join and leave a shared application. The name server maintains a list of all active users and ongoing sessions. It also serializes all updates to these lists. This control information (i.e., user and session information) is replicated to each conference manager. The conference managers spawn a new application process for each new participant. They also register each process with the central name server. A newly spawned local application process establishes a communication channel with every peer application. The information about the peers is retrieved from the local conference manager. Figure 2.4 illustrates the architecture of the GroupKit system. As can be seen in the figure, each client station has its own copy of the application. Clients are in constant, direct communication with each other in order to keep the application state and data consistent. Each client also has its own conference manager. The GroupKit session manager, which consists of conference managers and the central name server, is responsible for the initiation, maintenance and termination of sessions.

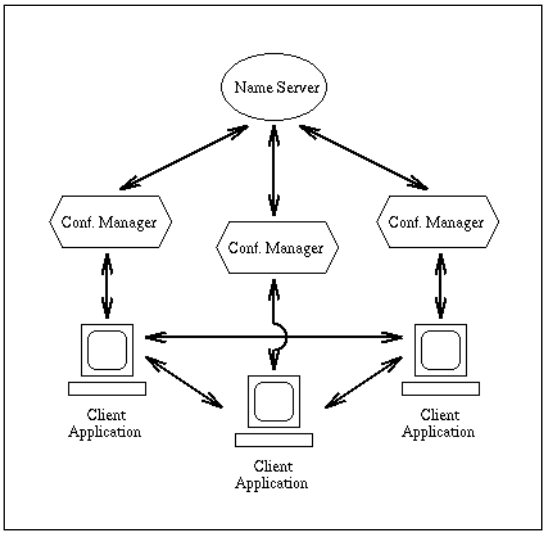



**FIGURE 2.3 Architecture of the GroupKit system.**

Client applications built with this toolkit use a replicated data architecture. The replication is done through a peer-to-peer communication channel between each application client. The application processes can multicast any update directly to peers and not through the centralized name server or conference managers. The advantage of using this replication method is that control is distributed among the conference managers. The disadvantage of this architecture mechanism is the high number of communicating ports which are used between each group of peer applications. These ports form a complete connected graph with redundant information. Another disadvantage of the GroupKit architecture is the lack of communication channels between the conference managers.

## 2.7 The Rendezvous System

The Rendezvous [Hill 90,92a,92b,93] programming toolkit was developed by Hill and Patterson as a tool for constructing synchronous groupware applications. Rendezvous provides an object-oriented language, derived from Common Lisp and the Common Lisp Object System (CLOS). The Rendezvous system is based on a programming model which separates the underlying data structure and its presentation. The system extends this model one step further for CSCW applications. The separation between the data and its presentation is achieved via a link layer which maps the underlying data to its view and vice versa. That is, a program which is generated by the Rendezvous system consists of three layers: Application (data), Link, and View layers. Each of these layers is implemented as a set of lightweight processes that interact with each other through two mechanisms: constraints and event handlers. Constraints are used at the link layer in order to express a relationship between data layer and view layer variables. Rendezvous provides support for multi-way constraints. This mechanism is used at the link layer to ensure consistency among all views. The event handling mechanism is employed to service any internal (inter-object signaling) and external (mouse button press) events. In this model, each function is associ-

ated with a number of events. Each of the events on the event list of a function will trigger the invocation of the procedure.

The Rendezvous system runs on SUN workstations, connected via Ethernet. These workstations run the Unix operating system with BSD (Berkeley Software Distribution) Unix sockets for inter-processes communication and the X Window System for user-interface programming. It incorporates the Common Lisp X interface (CLX) to handle the communications between view programs and the X server. The main feature of Rendezvous can be summarized as:

- Complete separation of underlying objects and their view, thereby allowing different users to have different views of the same objects.

The main shortcomings of the Rendezvous system are:

- Rendezvous applications are expected to run slowly, due to a strong reliance on Lisp and the elaborate constraint mechanisms.
- Rendezvous uses a centralized architecture. Although this communication architecture gives good performance for a small number of users, it is not scalable [Ahuja 90]. This means that the performance of an application will degrade with a large number of users.

### 2.7.1 Programming Environment of the Rendezvous System

The underlying design of the Rendezvous system is based on the common approach of UIMS (User-Interface Management System) developers of separating the underlying application and data structures from the application's user-interface. To achieve this goal, the Rendezvous introduces and employs the *ALV* (Abstraction Link View) mechanism. ALV is an architecture and programming method for building interactive multi-user applications, and is based on separating an interactive program into three main components: abstractions, views and links. A typical multi-user application, developed using ALV, consists of one abstraction and several views. The individual views do not have to be the

same. Each view can display a portion of the shared data, or two views can provide different presentations for the same set of data. The ALV model is depicted in Figure 2.5.

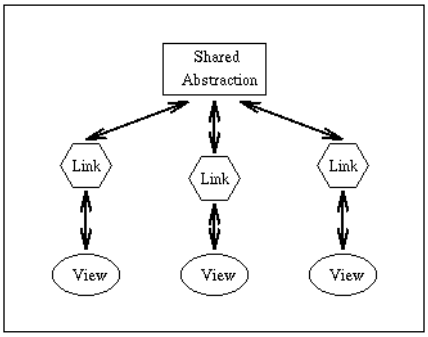



**FIGURE 2.4 Basic ALV architecture**

The run time architecture of an application developed using the Rendezvous system is based on a dynamic tree structure. In addition to the abstract data (called objects) which form a tree, each view forms a separate tree structure. The view trees need not be isomorphic to the abstraction tree. Hence, it is quite possible for a node in the view tree not to be linked to the abstraction tree. Since the structure of the trees can be changed during the run time, links are empowered to move objects within the hierarchy, as well as to create and destroy them.

### 2.7.2 Architecture of the Rendezvous System

A centralized server approach is used for applications developed with the Rendezvous system. In using the Rendezvous system, a programmer needs to develop three separate programs: an application server, links and views. The application server contains the underlying data structure of the system which resides at the server. Hence, only one copy of this program needs to be run for each session. The multiple updates from different sites are queued at the server and are serviced in a FIFO fashion. Links are a set of lightweight processes which act as a two-way constraint between the underlying application data and its various views. The link processes have a common object area (address area) as the

application's data. As a result, the link processes also reside on the central server machine. This provides a consistent state among the single copy of the data and its views.

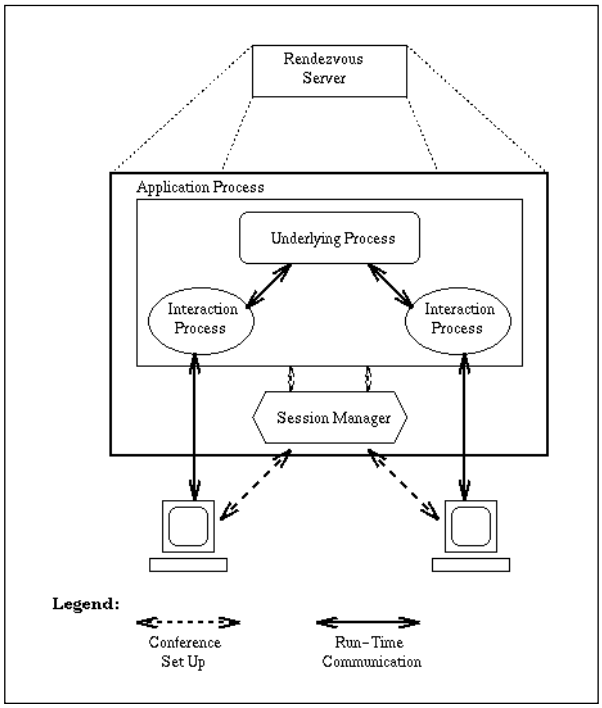



**FIGURE 2.5 The architecture of the Rendezvous system**

Figure 2.6 presents a schematic view of the Rendezvous system. As is shown in the figure, the application and the session manager processes both reside at the (central) Rendezvous server. Clients need to communicate with the Rendezvous server in order to establish a session and also to pass calls and receive call-backs. Special interaction processes act like a mediator between the underlying application process and client processes.

The run-time support of the Rendezvous system also includes a session manager (called the *name server*) which registers every active user and session. Once a session has been set up, the users can join the active session by querying the session manager.

## 2.8 The Suite System

Suite [Dewan 92a,92b,93] (a System of Uniform Interactive Type-directed Editors) was developed by researchers at Purdue University. The primary goals of the Suite system are to provide support for editor-based user interfaces in a distributed environment, while maintaining compatibility with a conventional operating system (UNIX) and programming language (i.e., C). The Suite design is based on a generalized editing model, which allows users to view programs as active data that can be concurrently edited by multiple users. Dialogue control is declaratively specified by issuing calls to the dialogue manager, while the application code is specified using the C programming language.

Suite consists of three constituent parts: Remote Procedure Call (RPC), persistent memory, and user-interface management. Suite RPC allows applications executing in different address spaces, and possibly on different hosts, to communicate with each other by via high-level remote procedures. Suite persistent memory consists of persistent applications whose data are saved on persistent storage. These persistent data are active in that they can trigger actions in response to the modification of data. Suite user-interface management is provided by dialogue managers which display selected data structures of applications. These dialogue managers provide the user with a generic editor-based user interface to modify the data structure in a syntactically correct fashion. They also allow the applications to trigger semantic actions (such as displaying results and communicating with other applications) in response to user changes to the data structures. Applications specify the display properties of the displayed data structures, and a multiple-inheritance scheme is provided for specifying default values of these properties. Applications and dialogue managers can execute on separate computers on a network.

The major features of the Suite systems can be summarized as:

- An elaborate coupling model, allowing different coupling strategies to be adapted in different situations.

- A broad range of collaboration schemes, allowing collaboration transparent applications to be incrementally made more collaboration aware.
- Automatic deployment of single-user codes in multi-user settings. This includes support for multi-user primitives (for programming multi-user-interfaces), based on the existing single-user interface primitives.

Some shortcomings of the Suite system are:

- The groupware applications that can be developed by using the tool are restricted to those which do not require support for direct manipulation interaction.
- The toolkit provides limited support for session management. Suite does not provide a facility through which the users can access the session's information.

### 2.8.1 Programming Environment of the Suite System

Developers using Suite are required to write two sets of programs: dialogue managers and the application itself. As mentioned above, the dialogue control is declaratively specified by issuing calls to a dialogue manager, while the application code is specified using the C programming language. Although Suite is largely based on the imperative C language, the presence of a declarative dialogue manager adds some declarativeness to the language.

In Suite, programmers are forced to resort to global variables to implement semantic feedback. Furthermore, the only kind of module or object which is available in Suite is a C file (a file containing C codes), called a *heavyweight object*, that is compiled into a stand-alone executable program. This lack of fine-grained objects burdens the modularization of the code in Suite.

While slow compilation substantially hinders the incremental development process, the expressiveness of the language is good. Suite does not provide support for the development of graphical groupware systems with a direct manipulation style of user-interface. This restricts the range of groupware applications that can be developed using the Suite

system. Reuse of existing single-user codes is handled very well in Suite. The built-in coupling support in the dialogue managers makes it possible to develop groupware by simply developing a single-user application.

### 2.8.2 Architecture of the Suite System

In multi-user Suite, each user's interaction with a multi-user application is managed by a separate dialogue manager, which manages all interaction entities created for its user. The dialogue managers of different users of a multi-user application are separate Suite objects and hence run as separate heavy-weight system processes, have separate address spaces, access rights and environments, and can be placed on different hosts. They are independently and dynamically placed, created, connected to, and disconnected from the application by their users. Figure 2.7 presents a schematic view of the Suite systems architecture. As is shown in the figure, each Suite client has its own dialogue manager, which is responsible for any communication between the client and the servers and/or other peer clients. Also note that all shared data (objects) are maintained at the server.

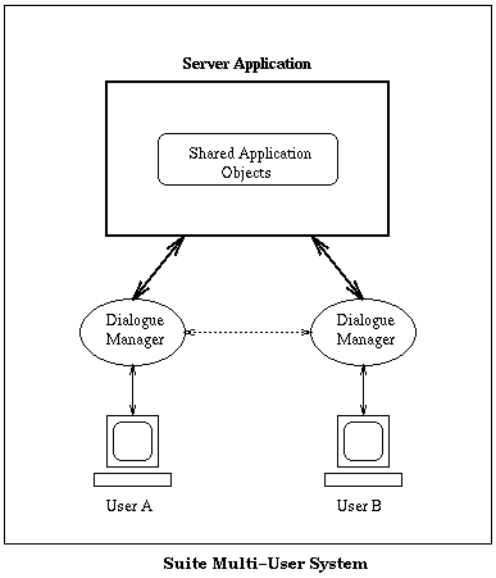



**FIGURE 2.6 The architecture of the Suite System**

The architecture of the Suite system can be regarded as a semi-replicated architecture. Suite supports a central semantic component (application object) and local user-interface components (dialogue managers). The Suite multi-user architecture allows user-interface tasks to be performed locally, and the computations tasks to be performed remotely on the server machine. Under this architecture, the user has to wait for communication delays involved in receiving results, but their displays are formatted locally by the dialogue managers.

## 2.9 The Weasel System

The Weasel system [Urnes 92] was developed partly at Queen's University and partly at the GMD, Karlsruhe. In Weasel applications, the base application code is completely separated from the user-interface code. Weasel employs a relational view model to achieve this separation. The idea is that application data structures and user-interface views are linked by relations. These relations map application data onto graphical views and vice versa. A special purpose functional language (RVL) is used to specify the relations. RVL offers easy customization of views for different users and limited support for providing collaboration aware constructs. Applications are specified using the Turing Plus language, and user-interfaces are generated automatically from the RVL specifications. All issues concerning distribution, communication, and synchronization are handled automatically by Weasel. Generated applications have a semi-replicated architecture and have been proven to scale well as the number of users increase [Graham 92a]. In their study Graham et al., measured the cost of adding a new client (user) to a session of up-to 22 clients. They further normalized the results so that the cost of a view update in a single-user session is one. They acknowledge that the time per client to update views does rise as new clients are added to the session, however, this increase is reported to be modest. In a ten user session, clients are reported to take approximately 1.5 times as long to make updates as in a one

user session. In a twenty user session, this number is reported to increase to approximately 2.5 times longer.

Figure 2.8 presents a pictorial representation of the Weasel system. As illustrated in the figure, the user interface state and the abstraction (i.e., data) are related via constraints. Updates to the user interface are received by interactors which in turn modify the user interface state. Should the state of the user interface change, constraints force the abstraction to change. Since RVL functions, which generate the views, are bound to abstraction, a change in abstraction triggers a recomputation of RVL functions which in turn cause the user interface to be updated. The main features of the Weasel system are:

- Support for incremental development.
- Reuse of single-user code in a multi-user setting.
- Good scalability.

The shortcomings of Weasel are:

- The RVL language provides limited expressiveness. Also, the syntax of the RVL language is somewhat clumsy

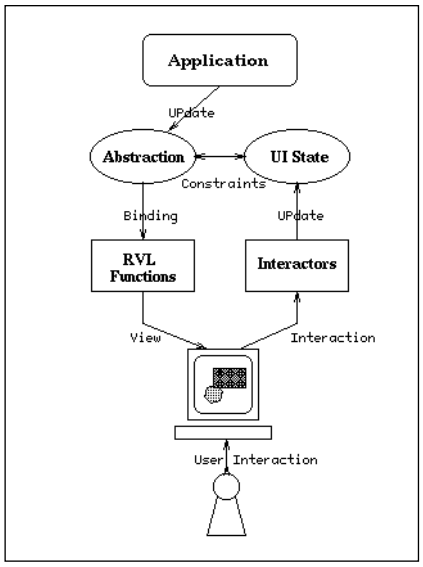



**FIGURE 2.7 The Weasel system model**

### 2.9.1 Programming Environment of the Weasel System

Weasel is based on the Relational View Model (RVL). The principle underlying the relational view model is the specification of a relation to facilitate bi-directional mapping between an abstract and a pictorial form of the same data. The abstract form of the data is typically represented as an abstract data type (i.e., part of the functional core of an application). The states of these data types are represented as graphical views. Both the abstraction and the view can be manipulated; the former by the application program, and the latter by an end user. Continuous maintenance of the relations will map the updates, made to the views, to the abstractions and vice versa. Using Weasel, the programmer need to provide a central application program, written in a traditional imperative programming language (i.e., Turing), and a set of view specifications, written in the declarative RVL language.

### 2.9.2 Architecture of the Weasel System

Weasel has a semi-replicated architecture. Under this architecture, the application and the shared data reside at the central server. The information pertaining to the user-interfaces and client views are replicated for each client. This distribution architecture is employed to improve the scalability of the applications. A full description of this architecture method is given in Section 5.2. Note that the semi-replicated architecture used in the Weasel system is different from Multi-User Clock implementation of semi-replicated architecture. First, in Weasel all data structures are maintained at the central server. Second, the separation of the client and the server data and programs are explicit, such that the executing application resides on the server and only the user interface is kept at the client sites.

## 2.10 Other Groupware Toolkits

In this section we present a brief description of some other toolkits which have influenced many concepts in groupware design and development.

### 2.10.1 GroupIE

The Group Interaction Environment (GroupIE) [Rudebucsh 91, 92] is a “generic environment offering high-level development and run-time support for cooperative applications” [Rudebucsh 92]. GroupIE is developed on top of Smalltalk-80 (extended with distribution support) at the University of Karlsruhe in Germany. The development support is in the form of a library of reusable Smalltalk classes. The use of the Smalltalk environment automatically includes good support for iterative and incremental development. The object-oriented nature of Smalltalk also makes component reuse possible. The shortcomings of GroupIE include the lack of support for session management, as well as the lack of support for converting single-user applications into multi-user applications. Also, while the performance of GroupIE is acceptable for prototyping, it is not suitable for production use [Urnes 94].

### 2.10.2 Lotus Notes

The most successful groupware application and programming tool in the industry is Lotus Notes [Lotus 93]. Notes can not only be used as an electronic mailing system (an example of asynchronous groupware), but it can also be used to build complex work flow applications. Notes can be classified as a replicated database infrastructure for asynchronous groupware applications. In a Notes session, the server processes can run on OS/2 and UNIX platforms. The client applications on the other hand, can run under Microsoft Windows, MacIntosh System(6.0), OS/2 and UNIX.

Notes has a number of advantages, including a flexible multi-server architecture where servers can co-exist in the same machine, same network, or remote sites. The developers of Notes also claim that a transparent communication layer allows the applications to send (or receive) message to (from) different platforms seamlessly. Notes, however, does not support a built-in session management.

### 2.10.3 MMConf

This toolkit was developed by Crowley et al. [Crowley 90] for building real-time tele-conferencing applications. The toolkit has basically two components: a distributed conference manager and a user-interface toolkit. The conference manager which is replicated at each site is in charge of communication with other conference managers and local applications. The user-interface toolkit, on the other hand, handles data access and synchronization, floor control policies and the file transfer mechanism. MMConf also provides some support for annotations and gestures, including a telepointer and audio/video conferencing.

## 2.11 Summary and Conclusion

This chapter reviewed several design features and requirements which are important for real-time (synchronous) groupware systems. These requirements were classified in terms of their contribution to the expressiveness and the ease of use of the groupware toolkit. The requirements for ease of use include support for distributed processes, transparent concurrency control, a flexible data architecture, a high-level abstraction model and a declarative specification language.

The current generation of synchronous groupware toolkits have addressed many of the groupware requirements. However, there are a number of requirements which still remain unresolved. The purpose of this thesis is to satisfy many of the groupware requirements by providing sufficient constructs in the specification language.

We believe that in order to prove themselves useful, the future generation of groupware toolkits need to address the requirements for both the expressiveness and the ease of use.

# Chapter 3

# An Introduction To Clock

## 3.1 Overview

This chapter provides an overview of Multi-User Clock, a programming environment for developing groupware and multi-user interfaces. Multi-User Clock is an extension of single-user Clock [Graham 94, Morton 94], a programming environment for developing single-user interfaces. Single-user Clock was developed by Nicholas Graham, Tore Urnes, Catherine Morton and the author. Multi-User Clock was designed and developed by the author.

This chapter will show how groupware systems are developed via the Multi-User Clock language. The chapter is structured around an example which is developed and extended throughout. All the examples of Clock architectures are screen snapshots of the actual architecture developed within the ClockWorks environment (as described in Section 3.2), and the examples of output generated by programs are screen snapshots of the running Clock program. The functional code given is the code required to create the example applications. The current implementation of Multi-User Clock is a prototype, and not all the presented examples run fully robustly. In addition Multi-User Clock applications have poor performance.

## 3.2 The Clock Language

The Clock language consists of two parts: a visual, object-oriented language which is used to specify the architecture (i.e., structure) of Clock programs, and a functional language (similar to Haskell [Davie 92]) which is used to specify the components of the architecture. Architectures are built using a visual programming tool called *ClockWorks* [Morton 94]. This tool enables the programmer to construct, edit and view architectures via direct

manipulation. The functional language is used to specify how the components are to appear on the display, how components react to input, and how components are made internally consistent. Special request handler components represent persistent state. The states of these components, cumulatively, constitute the state of the application as a whole.

Figure 3.1 illustrates a simple groupware application: a polling program. This program allows any user to make a proposition, and all participants to submit a vote regarding the proposal. While simple, this example serves to demonstrate a number of issues in groupware development. This application provides the users with both shared and private views. While the view containing the proposition and the ballot result are shared by all the users, the view containing the buttons is private for each user. The private views are designed to maintain the anonymity of the participants.

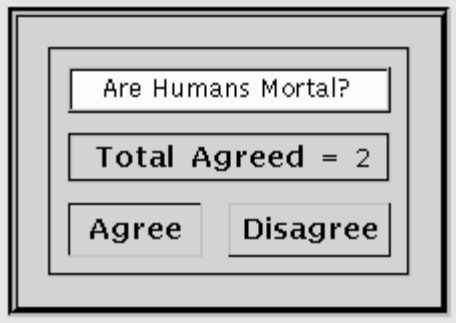



**FIGURE 3.1 A simple groupware application: A polling program. This example illustrates how shared and private views can co-exist in Multi-User Clock applications. In this application, while the views pertaining to the proposition and the poll result are shared by all the users, the view of the radio buttons is private for each user. The private views in this application are intended to maintain the anonymity of the users.**

The remainder of this chapter introduces the features of the Clock language through a number of simple examples. The components of these examples are later used as building blocks to construct the polling program described above.

### 3.2.1 Tree of Components

In Clock, programs consist of trees of communicating components. Clock has two basic components: *event handlers* and *request handlers*. An event handler defines an event handler component in terms of its appearance, its initial value, and its response to events. Event handlers are also responsible for handling I/O including mouse and keyboard input. In other words, event handlers are used to create an interactive display views. Since event handler components are stateless, request handlers are introduced. Request handlers implement abstract data types, which maintain a program's state.

Clock components communicate with each other through a series of internal events. There are two types of internal events in Clock*: requests* and *updates*. Request events are issued by the event handlers to access the state of the request handlers. Request handlers' information is used by event handlers to correctly generate their views and also to make themselves internally consistent. Update events are issued by the event handler components to change the state of a request handler. *Input* events, which are generated as a result of end-user actions, form the external type of events. In general, an architecture tree may have an arbitrary number of event handler and request handler components. In addition, each event handler component may issue, and each request handler may receive, an arbitrary number of requests and updates.

The remainder of this chapter presents an in-depth discussion of each of the Clock components and features. These presentations are accompanied with example programs to further familiarize the reader with the Clock language.

## 3.3 Event Handler Components

Clock programs are built as architectures of connected components. These architectures are specified in a graphical architecture editor within the ClockWorks programming environment. Event handler components form the main structure of the architecture tree.

Request handler components can be associated with event handlers. Each event handler component is an instance of a Clock event handler class. A Clock program may contain more than one instance of a particular event handler class.

Event handlers are responsible for the construction and maintenance of the view of Clock programs, as well as for handling end-user input. Figure 3.2 illustrates a simple example of an event handler component. In this example there is a single component of class *Button*. Clock programs correspond to tree components, and since the component is the root of the tree it is called *root*. While in Clock, simple views can be generated with minimal code, the construction of more complex views is also possible via built-in view constructs. As is shown in the figure, two different views can be defined for the component. Both of these views display a box containing the text “Hello World”. This example is intended to show that components’ code can be arbitrarily complex.

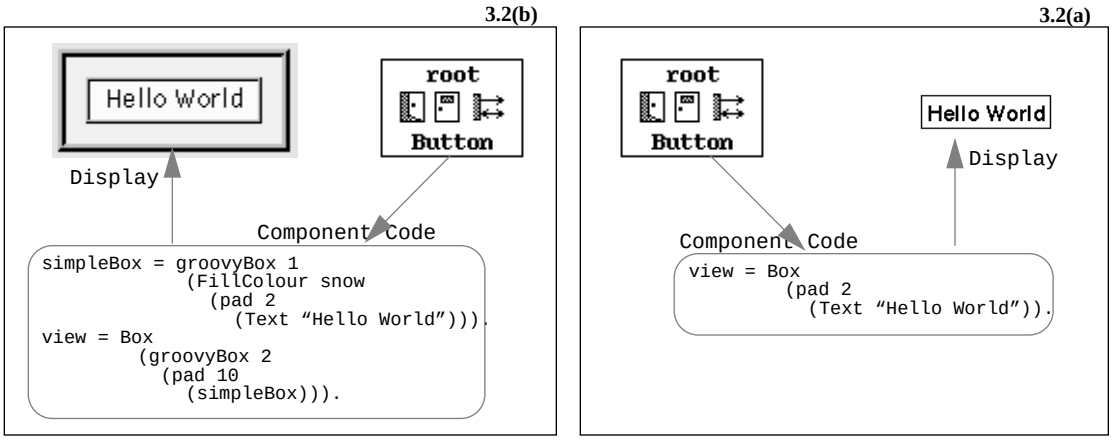



**FIGURE 3.2 Two possible Clock programs displaying “Hello World”.**

Figure 3.2(a) shows how “Box”, “Text”, and “pad” primitives can be used to construct a simple view. These primitives are used to draw a box, a text, and to add spacing between objects respectively. The functional code for the view in Figure 3.2(b) makes use of the predefined view primitives in Clock. The primitives in the Clock view language allow the specification of font, shadowing, relief, colour, border and positioning of display objects.

The *"groovyBox <borderWidth>"* function is an example of these predefined functions provided in the Clock library. In particular the *groovyBox* function maps an arbitrary view to the same view with a "grooved" border around it. The "*borderWidth*" specifies the width of the *groovyBox* border.

In addition to predefined view functions, programmers can also develop their own functions to manipulate views. The code implementing the view in Figure 3.2(b) illustrates this capacity. In this example, the *simpleBox* function is defined by the user.

As shown in Figure 3.2, each event handler component contains a number of icons. Figure 3.3 gives a brief description of each of the icons and their functionality.

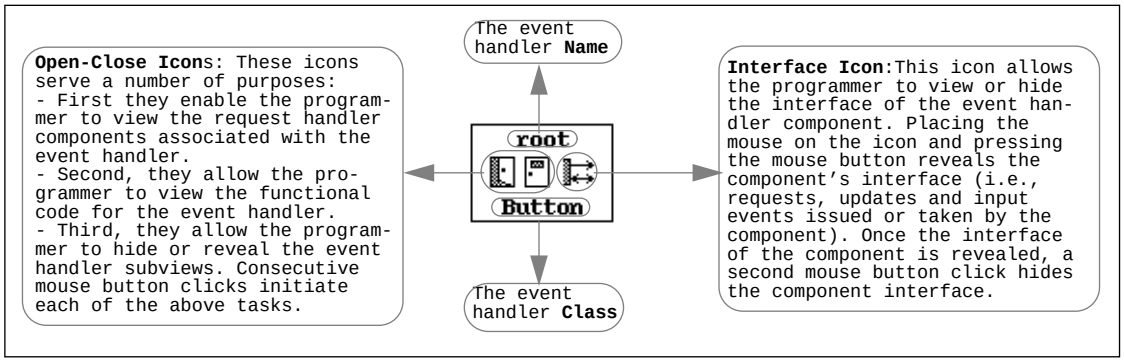



**FIGURE 3.3 A description of the event handler icon.**

The event handler components in an architecture tree are linked together via *subview* relations. This directive resembles the parent and child relationship common in most tree-like data structures. To illustrate how subviews are used, consider a case where we would like to display a view which contains multiple instances of the *Button* component introduced in Figure 3.2. Figure 3.4 illustrates how subviews can be added to a Clock component using a ClockWorks dialogue box. The figure also presents the resulting architecture, and the functional code for the *TwoViews* component. Note that the code pertaining to *Button* component is the same as presented in Figure 3.2(b) and is not shown in Figure 3.4.

In addition to the parent-child relationship, Clock maintains inheritance over the subview boundaries. This means that all the graphical and view properties of a parent event handler component are inherited by all the subview components. The subviews are also treated as first class values in the Clock language. In programming terminology, the first class values refer to those values within the program which all the operations supported by the language can be applied to. In functional languages, these operations are for a value to be passed as an argument to a function or to be returned as a result of a function. In Clock, subviews can be used as an argument to the view function. That is, the view function of a parent component can refer to the subview components through their subview names. For instance, the view function of the *TwoViews* component, shown in Figure 3.4, specifies the view of two *button* subviews, framed by a *groovyBox*.

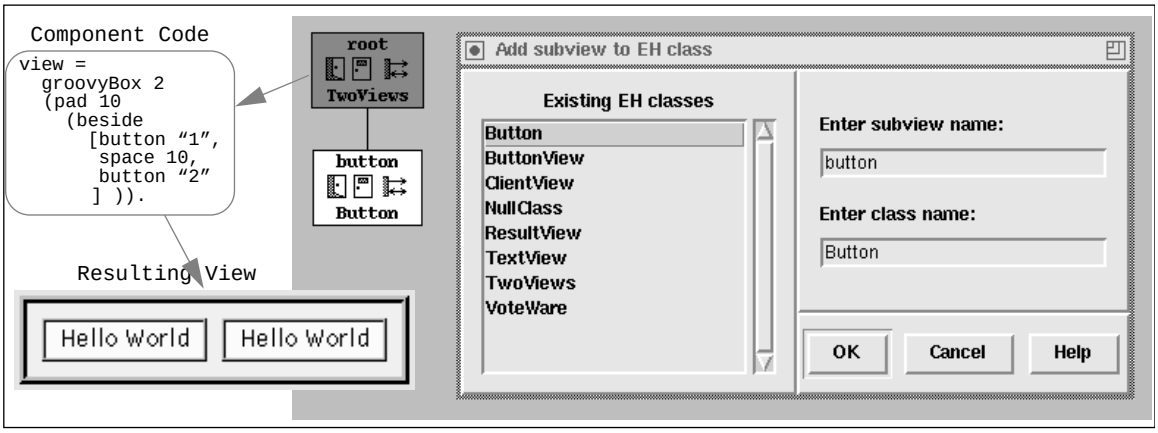



**FIGURE 3.4 Adding a subview to an event handler in the ClockWorks environment.**

Event handlers do more than display views. They are also responsible for handling the end-user input (e.g., mouse button). Section 3.5 illustrates how event handlers are able to handle users’ input.

## 3.4 Request Handler Components

In the previous section we studied event handler components. These components are used to create an interactive display view. However, event handlers are stateless. In order to

provide state, a second type of component, called request handlers are provided. Request handlers are a form of abstract data type which maintain an internal state. An arbitrary number of request handler components can be associated with each event handler component in the architecture tree. Although multiple instances of a request handler component can exist within a Clock application, each instance is associated with exactly on event handler. Request handler components provide an interface through which the event handler components can query or update their internal state.

In the following sections we first present an example of a request handler component, and then show how request handlers are used.

### 3.4.1 Example Request Handler

Request handlers are used to maintain and manipulate persistent state, thereby playing the role of data structures in Clock programs. Figure 3.5 shows an example of a request handler *Id* which implements a simple string identifier. As shown in the diagram, *Id* has two interface operations: *setMyId* is used to update the string identifier, and *myId* is used to query the value of the identifier.

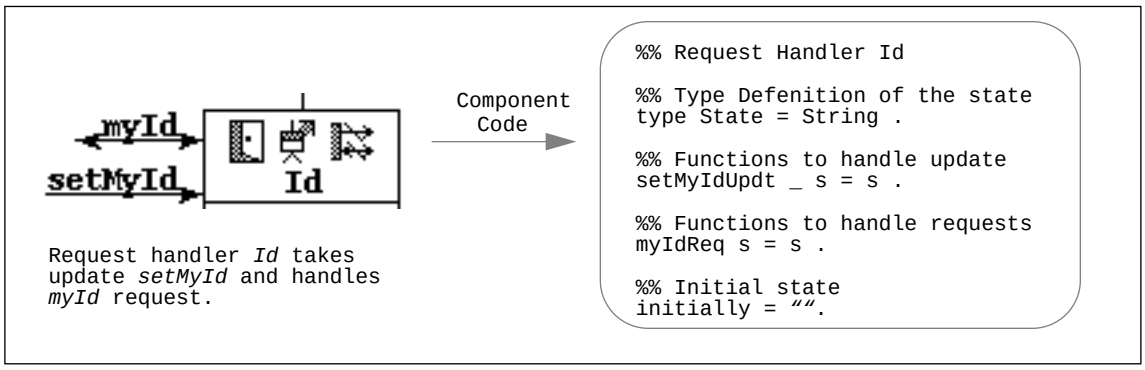



**FIGURE 3.5 A request handler in Clock: *Id*. The "_" represents the current state of the component.**

As shown in Figure 3.5, the code for a request handler requires: a type definition of the request handler's state, a function for each update and request which the request handler is

handling, and an *initially* function specifying the component's initial state. The state type specifies the component's state. The type of a request handler's state can be arbitrarily complex. In *Id* request handler, the type of the state is string. The *initially* function returns a value of state type (i.e., string), and initializes the request handler to the initial state. In the above example the *initially* function sets the initial state to a null string. The update function in a request handler responds to update events sent to the request handler component. An update function takes the current state of the request handler and returns a new state as its result. The request function, on the other hand, responds to the request events directed to the request handler. In the above example, the *setMyId* update sets the state of the request handler to the given string value and the request *myId* returns the current status of the request handler.

As shown in the Figure 3.5, each request handler component contains a number of icons in its display view. Each of these icons serves a different purpose. Figure 3.6 gives a brief description of each of the icons and its functionality.

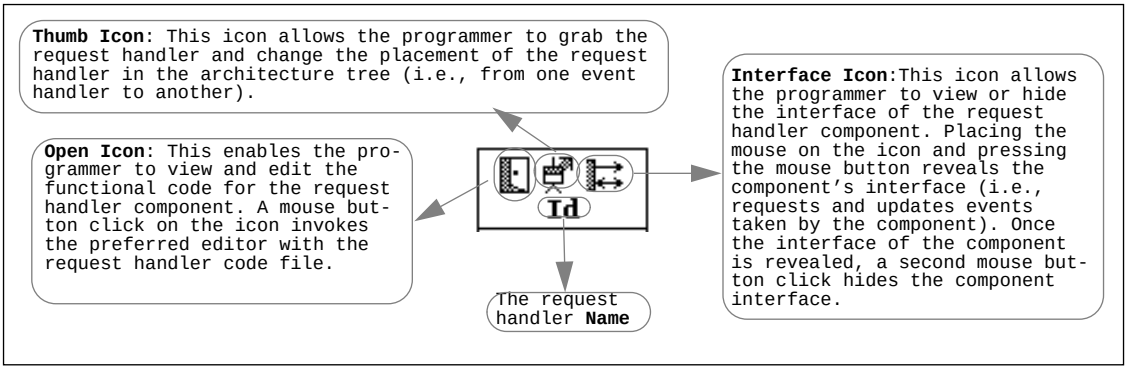



**FIGURE 3.6 A description of the request handler icon.**

### 3.4.2 Using Request Handlers

As an example of how request handlers are used, consider a case where we would like to construct two buttons which have the same functionality and interface but bear different labels. Since the code for both components are similar, we can make use of the *Id* request

handler to create the desired interface. Figure 3.7 shows the component tree, the component code and the resulting interface view.

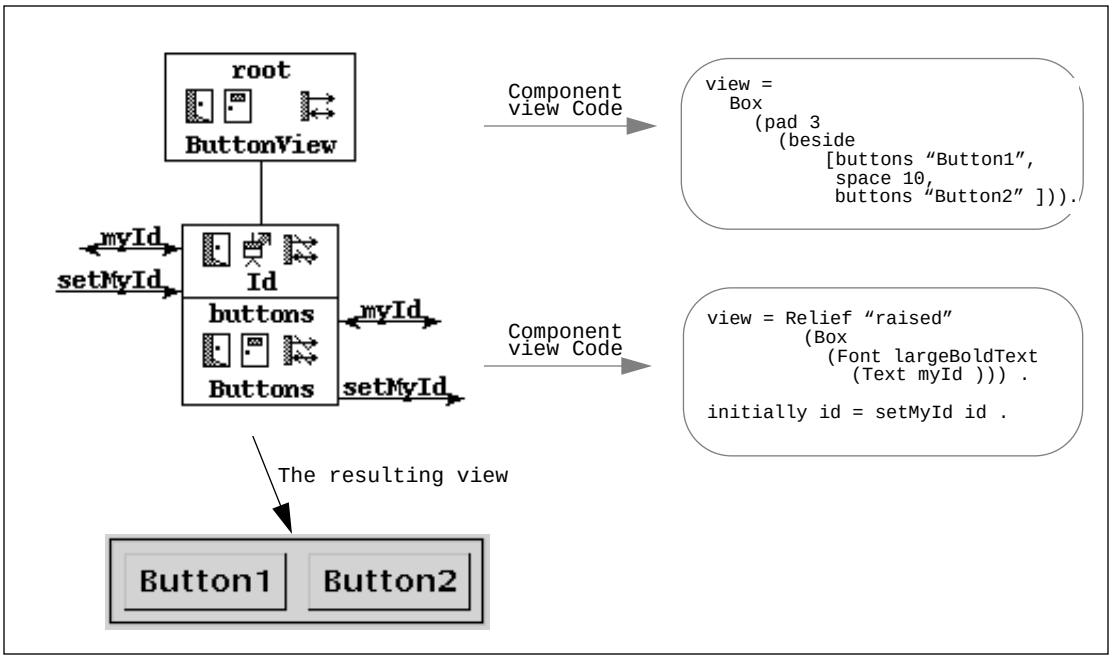



**FIGURE 3.7 An example to show the use of request handlers.**

Our example in Figure 3.7 is similar to excerpts from the polling program presented in Figure 3.1. However, we need to make the buttons presented in Figure 3.7 behave as *radio buttons*, so that clicking a button selects it. As with mechanical radio buttons, only one radio button may be selected at a time; so selecting a new button must cause all of the other buttons to be deselected. The selected button must convey its state (selected or not selected) to the user via graphical cues (e.g., by being highlighted or depressed). To accomplish this, a second request handler, *Selection*, is added to the architecture tree. The resulting architecture can be seen in Figure 3.11.

Clock maintains a library of predefined components. The radio buttons are one of the predefined components in the Clock library. These components can be directly incorporated in any Clock program. Other components which are defined in the Clock library include scroll bars and menus.

The placement of a request handler in an architecture tree determines the set of event handler components which can have access to the request handler state. In our example the *Selection* request handler is added to the *root* component in order to make it accessible to all the instances of the *RadioButton* class. In this particular case, the *Selection* request handler could have been added to the *buttons* event handler without modifying the semantics of the program.

## 3.5 Request, Update and Input Events

In order to provide an up-to-date view, event handler components need to be able to have access to the state of request handlers. Event handler components can query the state of a request handler by issuing a request directed to the request handler component. For instance, in the radio button example of Section 3.4.2, the *Button*s component needs to be able to query the state of the request handler *Id* (in order to portray the correct label) and *Selection* (in order to present the correct view: selected or deselected). Hence, two requests are added to the interface of the *Button* component. Figure 3.8 shows the resulting architecture.

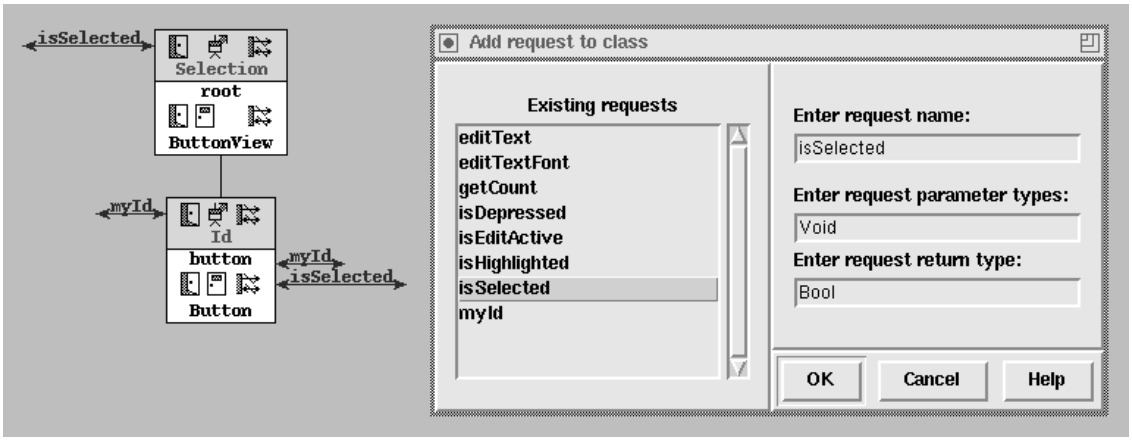



**FIGURE 3.8 The requests needed to implement a set of radio buttons. Requests, in the ClockWorks environment, are represented as double headed arrows. A request icon on the right side of a component indicates that the request is issued by the component. A request icon on the left side of a component indicates that the request is received by the component. The figure also shows how a request is added to a component class via the *add request to class* dialogue box.**

Note that a request must be added to both the event handler component which issues the request, and the request handler component which receives the request. The dialogue box depicted in Figure 3.8 shows how programmers can use a high-level mechanism to add request events to an event handler's interface.

The event handler components should also be able to modify the state of a request handler to reflect the changes in the programs. The event handler components can modify a request handler's state by directing an update event to the request handler. For instance, in the radio button example of Section 3.4.2, the *Button* event handler needs to be able to set the initial state of the *Id* request handler. It also needs to be able to change the state of the *Selection* request handler, to indicate that it is the current selected button. Hence, two update events are added to the interface of the *Button* component. Figure 3.9 shows the resulting architecture. Notice that, as with requests, an update must be added to both the event handler component which issues the update, and the request handler component which receives the update. While both requests and updates can be issued by one or more event handler components, they can only be received by one request handler component. The dialogue box depicted in Figure 3.9 shows how programmers can use a high-level mechanism to add update events to an event handler's interface.

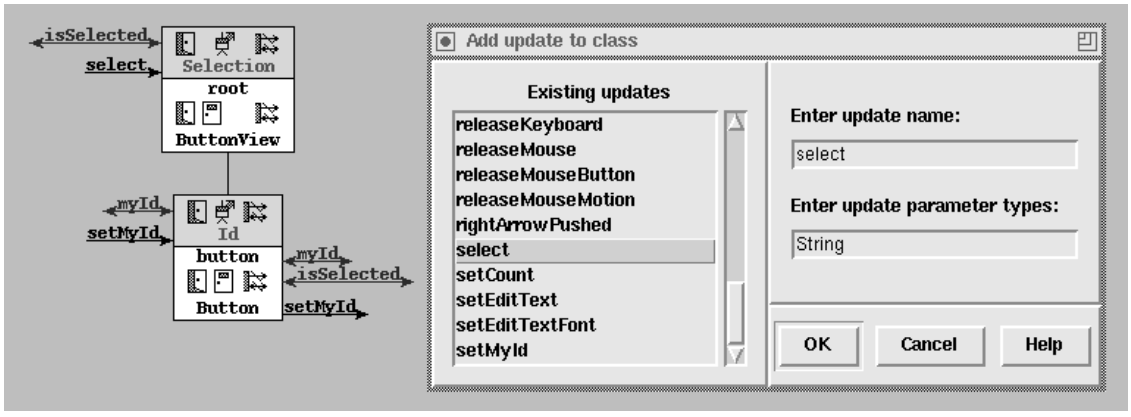



**FIGURE 3.9 The updates needed to implement a set of radio buttons. Updates, in the Clock-Works environment, are represented as single headed arrows. An update icon on the right side of a component indicates that the update is issued by the component. An update icon on the left side of a component indicates that the update is received by the component. The figure also shows how an update is added to a component class via the *add update to class* dialogue box.**

To make Clock programs interactive, we also need to enable event handlers to receive input events issued by the user. For instance, in the radio button example of Section 3.4.2, the buttons must be made sensitive to the mouse button, so that they can react to the *mouseButton* input. The *mouseButton* is one of a set of predefined inputs defined in Clock. Other forms of input which are supported by Clock include mouse motion and keyboard input. Figure 3.10 shows the resulting architecture and the ClockWorks dialogue box for adding an input event to an event handler component.

Once a component is made sensitive to an input, for example the mouse-button-click, whenever the mouse is depressed or released over the view generated by the component, a *mouseButtonUpdate* function is invoked with appropriate *String* parameter (i.e., “up” or “down”). The *mouseButtonUpdate* is a built-in function which handles *mouseButton* input. The code pertaining to the *RadioButton* component in Figure 3.11 illustrates how this function is used. In the Button component, this function specifies that when a button is clicked, it is to be selected.

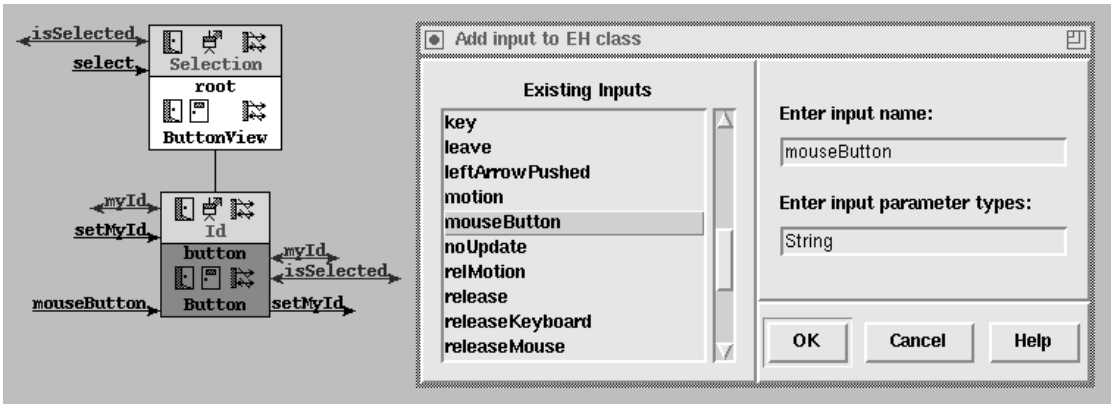



**FIGURE 3.10 The *mouseButton* input event needed to implement a set of radio buttons. Inputs, in the ClockWorks environment, are represented as single headed arrows, appearing on the left side of a event handler component. The figure also shows how an input t is added to an event handler class via the *add input to EH class* dialogue box.**

## 3.6 Declarativeness in the Clock Language

Clock is a declarative language. Clock programmers provide a specification of ‘what’ the program should do. The run-time system then determines ‘how’ to implement this specification. The main advantage of using a declarative language is that the programmer need not be concerned about the execution order of the program.

There are several properties which contribute to the declarativeness of the Clock language. The first property is automatic view updating. In Clock, views are declared via a set of specifications. These specifications dictate what the view should look like. The run-time system automatically determines when a view is out-of-date and how to bring the view up-to-date. This mechanism relieves the programmer from such concerns as to when views need to be updated. For example, consider the radio buttons in the polling program of Figures 3.1 and 3.11. In this example, the programmer need only provide a specification of what the radio buttons should look like in normal and selected states. The run-time system determines when a button is selected, and automatically updates the view of the selected button. It also maintains view consistency among the set of radio buttons as a whole.

The second way in which Clock is declarative is the automatic handling of events in a correct order. As discussed in Section 3.5, a number of internal events (i.e., requests and updates) are issued during the execution of a Clock program. Clock allows a number of *input threads* to co-exists in a Clock program. Each input thread refers to a number of chained events which is initiated as a result of a user input (see Section 4.4 for more details). Events within each input thread, and belonging to different concurrent input threads must be handled in an order that guarantees a correct result. The Clock run-time system guarantees an illusion of single-threadedness to the application users. The system behaves as if only one active input thread exists within the system. This mechanism frees the programmer from sequencing the internal-events in a correct order. A formal description of Clock and its declarative properties are given in [Graham 95].

Multi-User Clock extends the Clock language functionality in order to support development of groupware and multi-user user interfaces. This extension however, does not alter the semantics of the Clock language. In fact one of the main objectives of Multi-User has been to maintain the declarativeness properties of the Clock language.

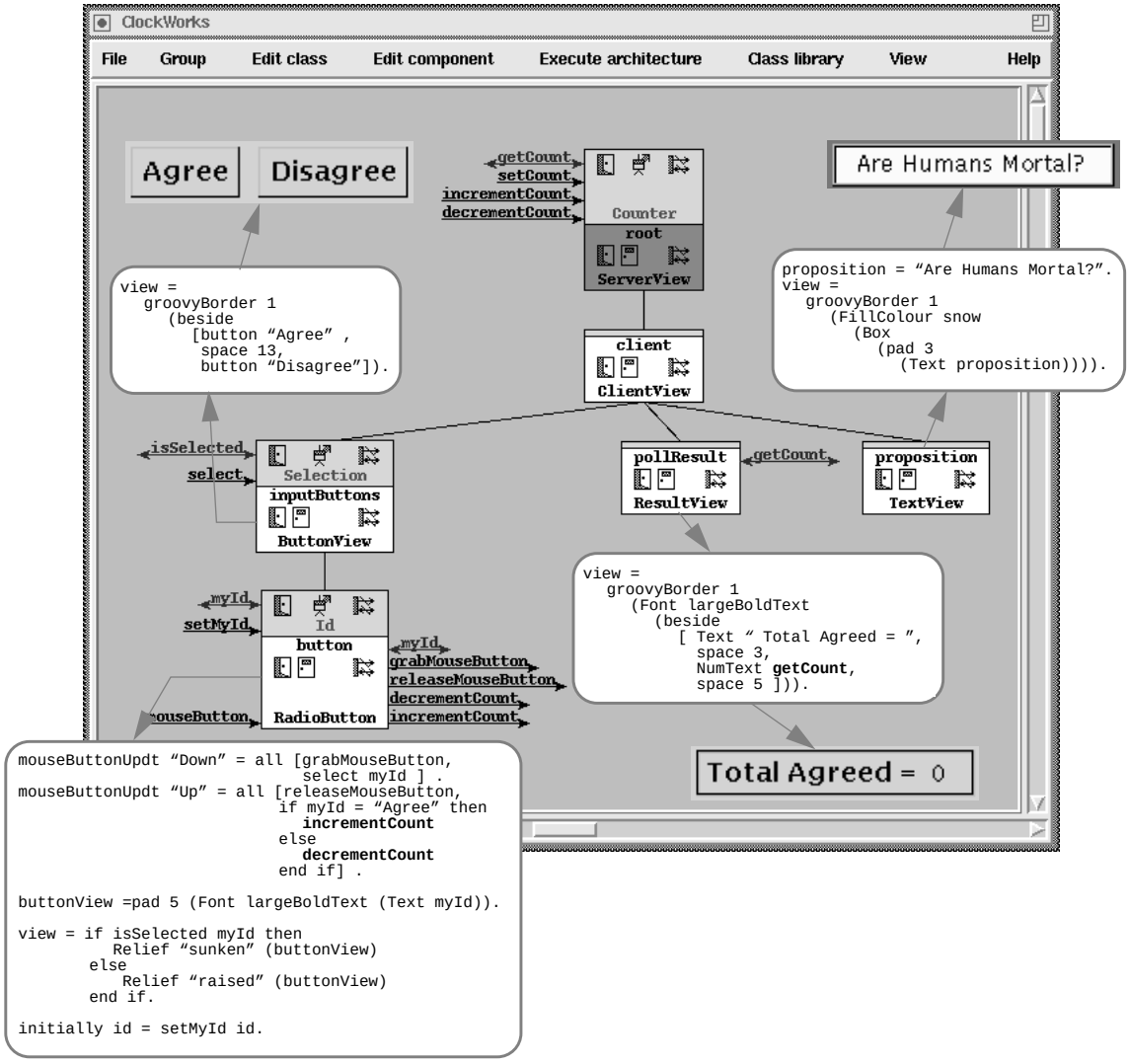
## 3.7 A Simple Multi-User Groupware Program

The previous sections introduced the basic Clock components and illustrated how these components communicate with each other. The presented features thus far, are common to both single and Multi-User Clock. These features were presented to familiarize the reader with the Clock programming environment. The remainder of this thesis presents the material which are exclusive to Multi-User Clock. Multi-User Clock is designed and developed as part of this thesis, using existing infrastructure provided by single-user Clock.

The following sections describe the Multi-User Clock features and show how they can be used to develop groupware using the example of the simple polling program presented earlier in this chapter. The polling program is designed for distributed users to organize a poll in order to decide on a proposition. All the users of the polling program view a proposition, and make their vote known confidentially. The following describes how this simple groupware application is developed using Multi-User Clock.

Figure 3.11 illustrates the complete architecture tree of the polling program. As the figure shows, the views of *TextView*, *ResultView* and *ButtonView* are composed to obtain the complete polling program. The *TextView* subview is designed to display a textual message on the screen (i.e., the proposition). The *ResultView* displays the poll result. The *ButtonView* provides a set of radio buttons through which the users can make their votes known. The complete functional code for each of these components is presented in Figure 3.11. Note that each of the components is responsible for a sub-section of the view generated by the polling program.

**FIGURE 3.11 The complete Clock architecture and functional code for polling program**




The first step in developing a Multi-User Clock program is to identify the building blocks or the components which are needed to construct the application. Each Multi-User Clock application can have an arbitrary number of event handler and request handler components. These components can be arbitrarily large or small, and they can handle an arbitrary number of inputs, requests and updates. The second step in developing a Multi-User Clock application is to structure the application's components in an architecture tree. The components of a Multi-User Clock are structured, in a single architecture tree, via the subview relation. This relation corresponds to the parent and child relation in the architecture tree.

Under the current implementation of Multi-User Clock, the components of an application are distributed in a *semi-replicated* fashion. A full description of the various distribution architectures is given Section 5.1. In order to provide a semi-replicated distribution model, the architectures of the Multi-User Clock applications are split into two sub-architectures: the server and the client architectures. The server architectures contain the components which are to be placed at the server machine. These components are used by the server programs. The request handler components which are contained in the server program constitute the shared data in the application. Likewise, the client architectures contain the components which are to be situated at the client machines. These components are used by the client programs. The client architectures are replicated for each client. This implies that each client has its own copy of the client architecture.

Each of these sub-architectures could contain an arbitrary number of event handlers and request handlers. It must be noted that the server and the client components do not correspond to two different architecture trees; rather they together constitute the application architecture tree as a whole. The server and the client architecture of the polling program are shown in Figure 3.12.

As is shown in the Figure 3.12, the server architecture consists only of the *ServerView* component. The client architecture however, consists of *ClientView*, *ButtonView*, *ResultView* and *TextView* components. Note that the placement of the *Counter* request handler at the server architecture decides that the state of the component is a shared data and hence it is accessible to all replicas of the client architectures. This mechanism allows all the client views to be able to query the up-to-date state of the *Counter* in order to display the correct poll result.

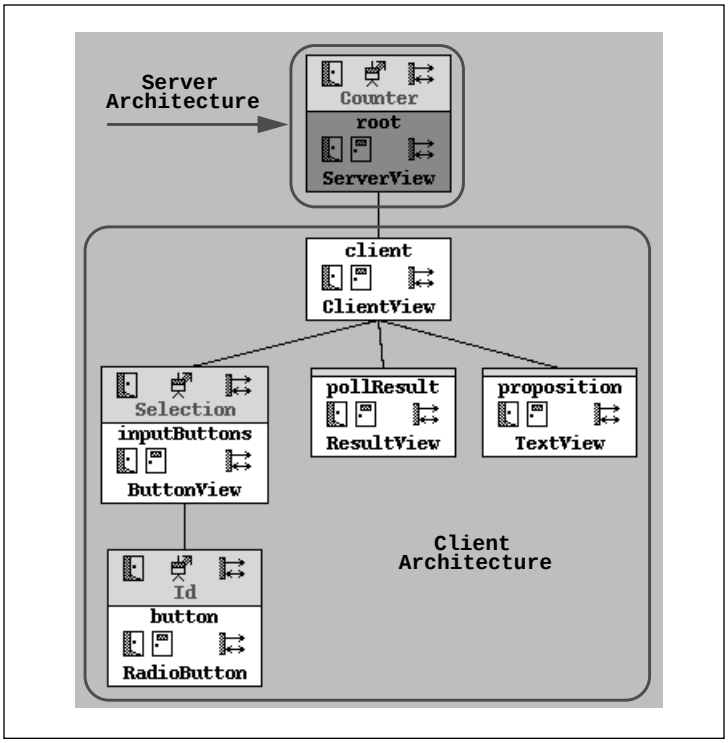



**FIGURE 3.12 The client and the server architecture of the Polling program.**

In an attempt to maintain the compatibility between single-user and Multi-User Clock applications, the subview directives are made such that they span over the network boundaries. For instance, the view function of *ServerView* component of the polling program is defined as: `view = client “”`.This definition is the same as other subview declarations. For instance, the view of the *ClientView* component is defined as: `view=above[textView “”,space 10,resultView “”,space 10,buttonView “”])`. Note that the format in which the subviews are used are exactly the same.

In a semi-replicated distribution architecture, the application data is split into shared and private data. While the shared data resides at the server machine, the private data are replicated to each client machine. Shared data make the construction of WYSIWIS, or shared views, possible. Shared views are parts of the applications interface which display the same views on all participants’ interface. Private or WYSINWIS views are constructed based on the shared data. In Multi-User Clock all interface views are WYSIWIS by default. However, these views can be customized by using built-in requests including

*myClientId* and *myClientName*. The built-in requests allow the programers to query the session information. This information include the unique identifier for each client application and the client name which is assigned by the user of the program upon initialization. A full description of view customization in Multi-User Clock is presented in Section 4.3.

All issues pertaining to the partitioning of the architecture tree and the replication of the client architectures are transparent to the programmers. This mechanism allows the programmers to be able to develop multi-user and groupware applications, with the same ease as single-user applications, without concern about the issues which are inherent in distributed applications. Clock programmers can develop applications without regard to distribution of the application components and communication between these components. The architecture of Multi-User Clock applications is presented in a cohesive form as a single tree of communicating components.

As can be inferred from Figure 3.11, the client components issue a number of requests and updates, some of which are handled by the request handlers within the client architecture, while others are handled by the request handlers residing at the server architecture. For instance, while *myId* and *isSelected* requests are handled by the *Id* and the *Selection* requests handlers respectively (i.e., two local request handlers), the *getCount* request is handled by the *Counter* request handler (i.e., a remote request handler). Likewise, while *setMyId* and select updates are handled by the local *Id* and *Selection* request handlers, the updates *incrementCount* and *decrementCount* are handled by the remote *Counter* request handler. As is evidenced by the functional codes for these components, all these requests and updates are defined using the same syntax. Note that there is no annotation to mark the *getCount* as a remote request, or to mark *incrementCount* and *decrementCount* as remote updates.

Multi-User Clock abstracts all details pertaining to the handling of the local and remote events. This implies that programmers need not be concerned where in the architecture

tree a request or an update is issued or where it is handled. Clock guarantees the correct delivery of requests and updates to the receiving components. This feature allows the Clock programmers to develop multi-user applications, without being concerned about the internal event handling mechanisms between local and remote components. This provides the illusion of developing single-user applications for the developers.

## 3.8 Transparent Constraint Maintenance

As we observed in the previous section, the view function of an event handler component dictates how the view of that component is to appear on the display. We also noted how the view of an event handler component can be constrained to the state of one or more request handlers. For instance, the *pollResult* does not always display the same value. Whenever a user agrees or disagrees, an *incrementCount* or *decrementCount* update is issued. As a result, the display presentation of *pollResult* is changed to reflect the new state of *Counter* request handler.

In Clock, the programmer need not be concerned about when a view function is updated. The language guarantees that whenever a view function is out of date it will be evaluated, and the result of the new function will be displayed automatically. Hence, view functions are a form of *constraint*, specifying the appearance of the display as a function of the current state of the program. The constraint mechanism in Multi-User Clock spans across the distributed copies of the client architecture and the server architecture. All the issues regarding the registration, maintenance and activation of constraints are handled by Clock without involvement of the programmer. The current implementation of Clock performs data flow analysis to determine when views are potentially out of date, providing incremental display updates.

## 3.9 Summary

This chapter introduced the Clock language and many of its features. First, it was shown that the Clock language consists of two constituent parts: an object-oriented visual language and a functional language. While the visual language is used to structure the architecture of the application components, the functional code is used to specify the components' display view, their response to events and their consistency. Second, it was shown that Multi-User Clock applications are represented as communicating components. It was noted that these components communicate via a set of internal events.

We saw how Clock provides a high-level graphical language for structuring Clock architecture trees and how constraints are used to provide consistency in Clock. We also reviewed the features of the Multi-User Clock. This groupware toolkit is designed and developed as part of this thesis. Through simple examples it was shown that the Multi-User Clock programming tool can be used to develop groupware systems.

Chapter 4 will illustrate how Clock provides adequate support for many of the groupware requirements numerated in Chapter 2.

# Chapter 4

# Language Design

## 4.1 Overview

Chapter 3 illustrated the features of the Multi-User Clock language. The purpose of the present chapter is to show how these features address some of the groupware requirements outlined in Chapter 2. Although all the requirements listed in Chapter 2 are important and contribute to the success of a groupware toolkit, due to the time constraint, our work provides support only for a subset of the listed requirements. The supported requirements constitute the fundamental criteria for groupware toolkits. Future work will result in more of the design requirements being satisfied.

In developing Multi-User Clock, we had two design objectives. The first objective was to extend the Clock language while maintaining its declarative properties. The second objective was to provide a high-level programming environment in which all distribution issues are hidden from the programmer.

These objectives are met by providing support for the following requirements: support for transparent distribution of applications and data, a built-in communication infrastructure with a transparent event handling mechanism, support for session management, support for development of collaboration aware applications, support for customization of user interfaces, support for concurrency control and support for a transparent constraint maintenance. The following sections discuss how these requirements are addressed in Multi-User Clock.

The presentation of this chapter is based on the three examples which were designed and developed using Multi-User Clock. These examples are: a multi-user drawing program, a polling program and a Terminal Reservation System (*TRS*). Each of these applications was

designed to illustrate a subset of requirements supported in Multi-User Clock. We draw examples from these applications to show how the selected requirements are supported in Multi-User Clock.

## 4.2 A Shared Drawing Program

Chapter 2 introduced the shared drawing program. This application provides a shared canvas-like interface on which participants can draw objects. These objects are currently limited to boxes and lines. Each participant can add an object onto the shared canvas. All objects drawn by one user are represented by a unique colour. At any given time, each participant's interface depicts all objects on the canvas. This example is used to demonstrate how Multi-User Clock provides support for transparent distribution of application architecture, transparent communication infrastructure including a transparent event handling mechanism, roles, session information, and collaboration aware interactions.


**FIGURE 4.1 A groupware example: A Shared Drawing program.**

A *communication infrastructure* is a set of facilities which enables the remote programs or processes to communicate with each other. *Session information* is the data pertaining to the active connections during a session. *Collaboration aware interaction* allows the participants to be aware of each other's actions. According to Ellis, "A *role* is a set of privileges or responsibilities attributed to a person or to a system module" [Ellis 89].

### Transparent Distribution of Application Architecture and Data

Figure 4.2 illustrates the architecture of the shared drawing program. Multi-User Clock architectures are split into client and server sub-architectures. The shared drawing architecture consists of three event handler components. While the *ClientView* and *Button* components reside on the client architecture, the *ServerView* component resides on the server architecture. The client and the server architectures of the shared drawing program are shown in Figure 4.2. A single copy of the server architecture remains on the server. The client architecture, however, is replicated and one copy is used by each client. The split of the architecture tree and the replication of the client architectures are transparent to programmers. Programmers view the architecture tree as one structure. This automatic replication of architecture and application data provides partial support for the high-level abstraction model.

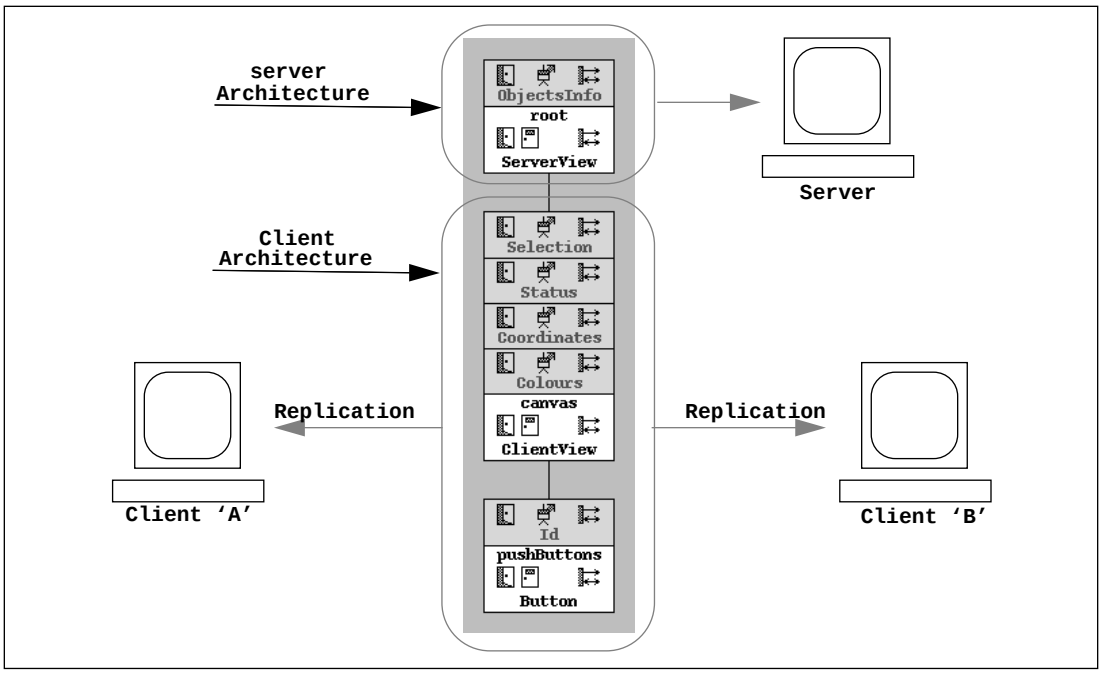



**FIGURE 4.2 The architcture of the shared drawing program**

As shown in Figure 4.2, the shared drawing program also contains six request handler components. While the *Coordinates*, *Status*, *Selection*, *Id* and the *Colours* request handlers

are part of the client architecture, the *ObjectsInfo* request handler is part of the server architecture. Figure 4.3 illustrates the server architecture of shared drawing program along with the functional code for the *ObjectsInfo* request handler. The placement of the *ObjectsInfo* request handler in the server architecture determines that the state of this request handler be part of the shared data and be accessible to all clients. The *ObjectsInfo* request handler maintains the coordinates and the ownership information of the objects currently on the display. The *Coordinates* and *Status* request handlers are used for drawing rubber band objects as the end-user drags the mouse from the point of origin to the final point. The *Id* and *Selection* request handlers are used to implement a set of radio buttons. These radio buttons allow an end-user to select the object type which he/she wishes to draw. The *Colours* request handler maintains a table, in which a unique colour is assigned to each client identification number. In Multi-User Clock, client programs are identified by a unique identification number.

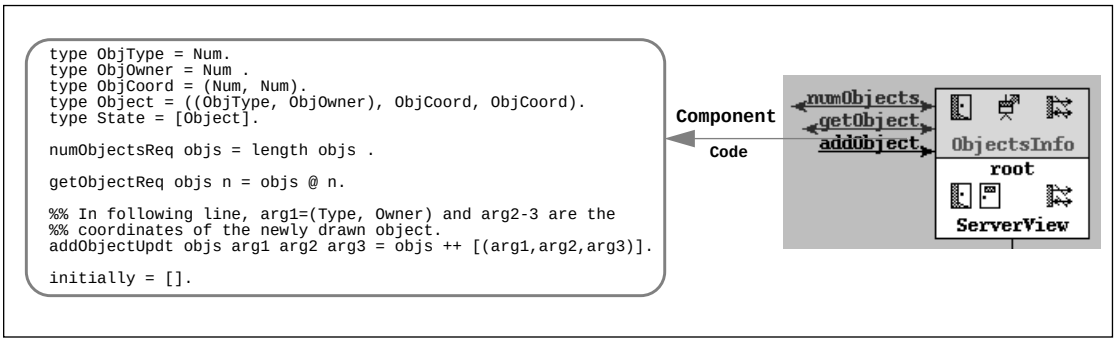



**FIGURE 4.3 The server architecture of the shared drawing program and the code for *ObjectsInfo* request handler. The code illustrates how objects are stored internally. Each object is identified by its coordinates and the client site where it was created.**

### Transparent Communication Infrastructure

Multi-User Clock consists of two executable programs: the server and the client. The server program runs on the server machine and uses the server architecture. The client program, however, is run on the client machines, each of which has its own copy of the client

architecture. In theory, the server and client programs can run on any machines on the Internet. Note that the server and the client programs can also be run on the same machine.

To demonstrate the invocation mechanism of Multi-User Clock programs, consider that one desires to run the server program on *tivoli*`[tivoli.cs.yorku.ca]`, and run a client program on *spruce* `[spruce.cs.yorku.ca]`. Both of these machines are part of the Computer Science graduate laboratory at York University. To invoke the server program, an end-user would type: `goServer MU-Drawing`. This will start the server program on the workstation where the command is issued. To invoke the client program, an end-user would type: `goClient MU-Drawing "Roy" tivoli.cs.yorku.ca`. The first argument specifies the multi-user program which the end-user wishes to run. The second argument specifies the desired name for the user of the program. The third argument is the network name of the server machine where server program is being run.

The Multi-User Clock programmer need not specify a single line of code for establishment, maintenance, or termination of the communication channels between the server and the client programs. In Multi-User Clock, the establishment of communication channels between the clients and the server, management of call and callback messages, and maintenance of session information are all handled transparently. This transparent mechanism relieves the developers from all the issues related to the distributed processes and communication between them. This provides an illusion to the developers as if they were developing single-user applications.

**Transparent Event Handling**

Figure 4.4 shows the complete client architecture of the shared drawing program. The client architecture issues a number of requests and updates, some of which are handled locally by the client program and others handled remotely by the server program. The requests which are handled locally include *getOrigin*, *getFinal* and *currentColour.* These requests are used to query the original and final points of a newly drawn object, and also to

draw the objects in the appropriate colour. Likewise, the updates which are handled locally by the client architecture include *setCoord* and *resetCoord* updates. These updates are used to specify coordinates of a rubber band line or box as the end-user drags the mouse from the origin to the final point.

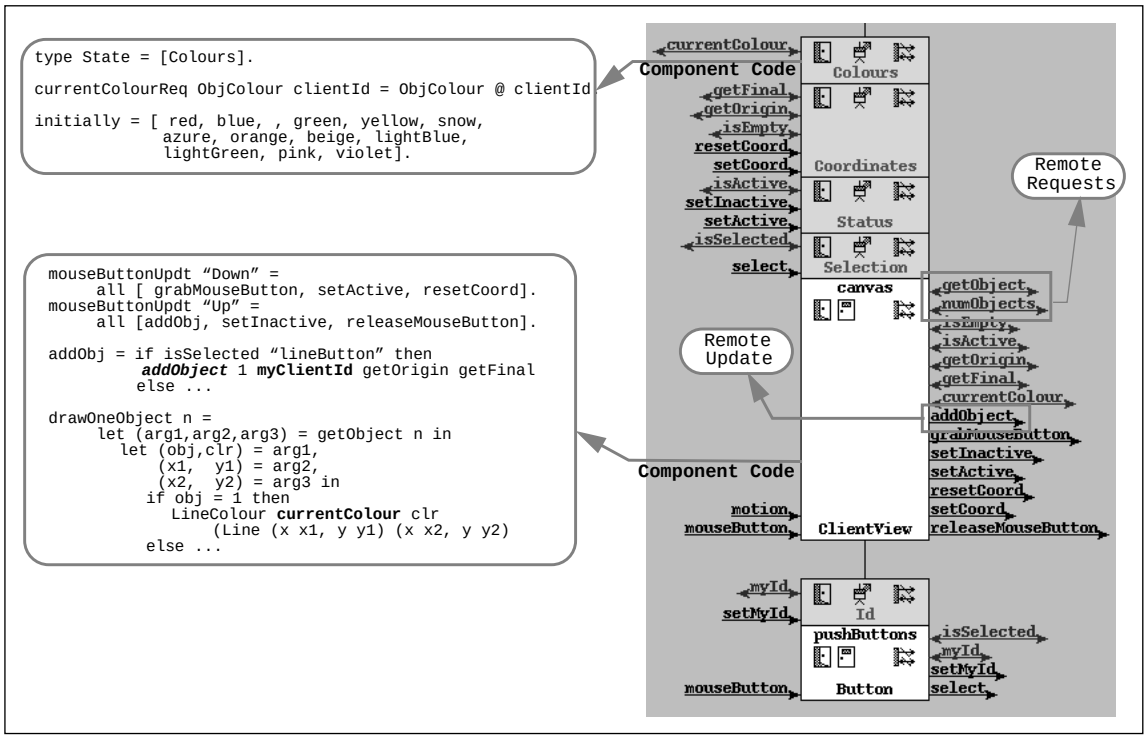



**FIGURE 4.4 The client architecture of the shared drawing program and the code for the *Colour* request handler. The figure also illustrates a segment of code for the *Canvas* event handler which is responsible for addition of a new line to the central data and also for drawing a new line on the canvas.**

In this example, the *addObject* update, which adds a new box or a new line to the central shared data, is treated as remote update. In addition, *numObject*s and *getObject* requests, which query the quantity and the coordinates of the objects respectively, are treated as remote requests. Multi-User Clock handles both local and remote requests and updates in a transparent fashion. This means that all the requests and updates are declared and used via the same mechanism and syntax. This occurs regardless of which part of the architecture tree the requests are being issued from or where they are handled. The Multi-User Clock run-time system automatically decides when a request or an update needs to be

issued and whether it can be handled locally or remotely. This transparent handling of the internal events is part of the Multi-User Clock goal to abstract the execution details from the programmers. The code for *Canvas* event handler and *Colour* request handler components of the client architecture, which are presented in Figure 4.4, illustrate how local and remote requests and updates are defined and used via the same mechanism and syntax. For instance, consider the following code segment which adds a new object (i.e., a line) to the *ObjectsInfo*. In this example, while *addObj* is a local update function, *addObject* is a remote update. As it is evident by the code, both local and remote updates are used in the same fashion.

```
addObj = if isSelected “lineButton” then
            addObject 1 myClientId getOrigin getFinal
         else ...
```

Also, consider the local requests *getOrigin* and *getFinal*, and remote request *getObject*. Note that the syntax for using both types of requests is the same.

**Supporting Roles and Session Information**

A colour coding mechanism is employed to identify the objects drawn by different users. This colour coding scheme not only provides partial support for collaboration between the users of the shared drawing program, but also provides support for *roles* which is presented in the form of the ownership of an object.

To support roles in shared drawing application, a particular colour is assigned statically to each client (user) of the program. Figure 4.4 illustrates the client architecture of the shared drawing program. The figure also presents the code for the *Colour* request handler. This request handler maintains a static colour table, one colour for each client. The figure also shows a portion of the code for the *ClientView* event handler. This code segment is responsible for issuing an update to add the information pertaining to a new line to the central data, and also to display a new line on the canvas. Once an object is drawn on the canvas

(i.e., when mouse button is released) *addObj* function is invoked. As shown in Figure 4.4, the code for adding a newly drawn line to the *ObjectsInfo* request handler is:

```
addObj = if isSelected "lineButton" then
            addObject 1 myClientId getOrigin getFinal
         else ...
```

This function generates an *addObject* remote update to add the newly drawn line to the central data (i.e., *ObjectsInfo* request handler). The *addObject* update has four parameters: the object type (i.e., 1=a line, 2=a box), *myClientId* and the two coordinates. In Multi-User Clock, each client is given a unique identifier during the initialization procedure. The *myClientId* request returns this unique identifier. This information is used to assign ownership information to each object.

The *myClientId* is one of the built-in requests which allows the programmers to query the session information. These requests are used with the same syntax as a regular request. Multi-User Clock maintains session information. This information includes the unique client identifier and client name of each active client program. While the client identifier is assigned by the internal system, the client name is specified by the end-user of the program. The Multi-User Clock programmers have access to the session information. This information can be accessed via a set of built-in requests in Multi-User Clock. The client identifier and name can be queried via *myClientId* and *myClientName* built-in requests. The client identifiers of all clients can be queried via *allClientId*, and can be translated to client names via *clientIdToName* built-in requests. The session information is used in the drawing program to provide the colour coding scheme. The code for drawing a line in the correct colour which is presented in Figure 4.4, is shown below:

```
drawOneObject n =
      let (arg1,arg2,arg3) = getObject n in
         let (obj,clr)=arg1,(x1,y1)=arg2,(x2,y2)=arg3 in
            if obj = 1 then
               LineColour currentColour clr
                  (Line (x x1, y y1) (x x2, y y2)) ...
```

The *drawOneObject* function takes the *ObjNum* parameter. Since objects are stored in a list, this number represents the object index in the list. This function makes a remote request, *getObject*, in order to fetch the information for the *ObjNum* object from the *ObjectsInfo* request handler, residing at the server. The result of this function is a triplet: consisting of two end coordinates and a numerical identifier (i.e., *clientId*) which represent the owner of the object, and the object's type. The *LineColour* is a built-in view command which changes the default colour for drawing lines. The *currentColour* is a local request which is handled by the *Colour* request handler. This request takes the *clientId* as its argument. Upon receipt of this request the *Colour* request handler looks up the colour value in the colour table corresponding to *clientId* and returns that value.

As is evidenced by the code segments presented above, the built-in requests are specified using the same syntax as other requests and are allowed anywhere general requests are allowed.

**Support for Collaboration Awareness**

Groups sense collaboration in various ways. Often the collaboration is supported through a combination of audio-visual and non-verbal cues [Ishii 94]. There are many ways through which a groupware system can convey a sense of collaboration among its participants. These methods include support for audio and video communication, shared viewing, annotation of the shared objects and tele-pointers (for pointing to the shared objects). The ideal groupware toolkit would provide support for all these methods so that the developers can select the collaboration techniques which best suit the developing application. Due to time constraints, our support for collaboration aware techniques is limited to shared viewing only. The shared viewing properties of Multi-User Clock follow directly from its support for development of WYSIWIS (What-You-See-Is-What-I-See) user-interfaces. For instance, in the shared drawing program, the collaboration is sensed among participants by allowing all users to observe the objects drawn by other users.

The shared drawing program provides WYSIWIS views. This implies that all the users of the shared drawing program view the same objects at any given time. The interface of Multi-User Clock applications by default provide WYSIWIS views. That is, the programmer need not provide additional code to provide shared viewing.

In order to provide WYSIWIS views which are dependant on the state of a request handler, the programmers need to place the request handlers which maintain the shared data at the server architecture. In this case, information pertaining to the drawn objects (i.e., ownership and coordinates) which are maintained by the *ObjectsInfo* request handler is treated as shared data and is placed in the server architecture. The placement of the *ObjectsInfo* request handler at the server architecture makes the data maintained by the request handler accessible to all client programs and enables the desired WYSIWIS viewing in the shared drawing program.

Multi-User Clock also provides support for control of the granularity of the update events in WYSIWIS interfaces. For example, in the shared drawing program, an object drawn by a user on her/his canvas, is drawn on other participants' canvas only after the object is fully drawn on the owner's canvas. This could be altered so that all participants could view the intermediate steps involved in drawing the objects (i.e., all participants would view the rubber band line or box).

The shared drawing example demonstrated the transparent architecture distribution, transparent communication infrastructure and collaboration aware capabilities of Multi-User Clock. The polling program and *TRS* systems will further demonstrate Multi-User Clock functionality which satisfy other groupware design requirements.

## 4.3 A Polling Program

The polling program example was introduced and discussed in detail in Chapter 3. In this section, this example is revisited to show how Multi-User Clock provides support for cus-

tomized views. The view customization facilities are methods which make the construction of both WYSIWIS (What-You-See-Is-What-I-See) and WYSINWIS (What-You-See-is-Not-What-I-See) views possible.

**Support for Customized Views**

The shared drawing program introduced in Section 4.2 only provides WYSIWIS views. Multi-User Clock also supports the development of WYSINWIS views. In WYSINWIS views, the user-interface of each user may display different images. Figure 4.5 shows how both types of views co-exist in the polling program. While the views for the poll result and the proposition are WYSIWIS, the view for radio buttons are WYSINWIS views.

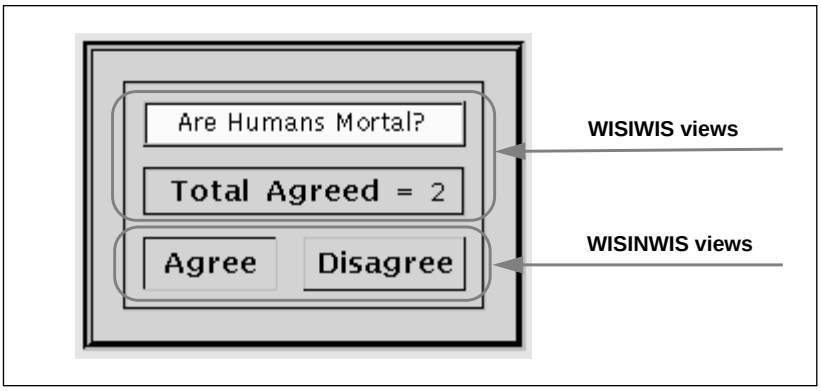



**FIGURE 4.5 Support for different views is shown in the polling program.**

The interface of Clock applications consists of two distinct views: *static* and *dynami*c. The static views are those which remain the same during the course of an application execution. For example, in the polling program, the views for proposition and the button labels are static views. The dynamic views, however, are changed during the execution of the application as a result of users' input. The poll result and the state of the radio buttons (i.e., raised/sunken) of the polling program are examples of dynamic views.

In Multi-User Clock, static views are, by default, purely WYSIWIS. These views, however, can be customized to produce WYSINWIS views. Static WYSINWIS views can be

constructed for each client interface based on the clients' unique identifiers and/or clients' names. For instance, consider that we would like two users of the polling program to have different views for the radio buttons. More precisely, we would like each set of radio buttons to bear a different labels. Figure 4.6 illustrates the functional code, which makes these static WYSINWIS views possible, as well as resulting views. Note that the colour of each user's interface is also customized in that each has a different background colour.

Dynamic views are a function of the request handlers' states. For instance, the view for the poll result is a function of *Counter* request handler state (Figure 3.12, Section 3.6). That is, whenever the state of the *Counter* is changed, so is the view for the poll result. The construction of dynamic WYSIWIS and WYSINWIS views in Multi-User Clock is made possible by support for the flexible placement of data.

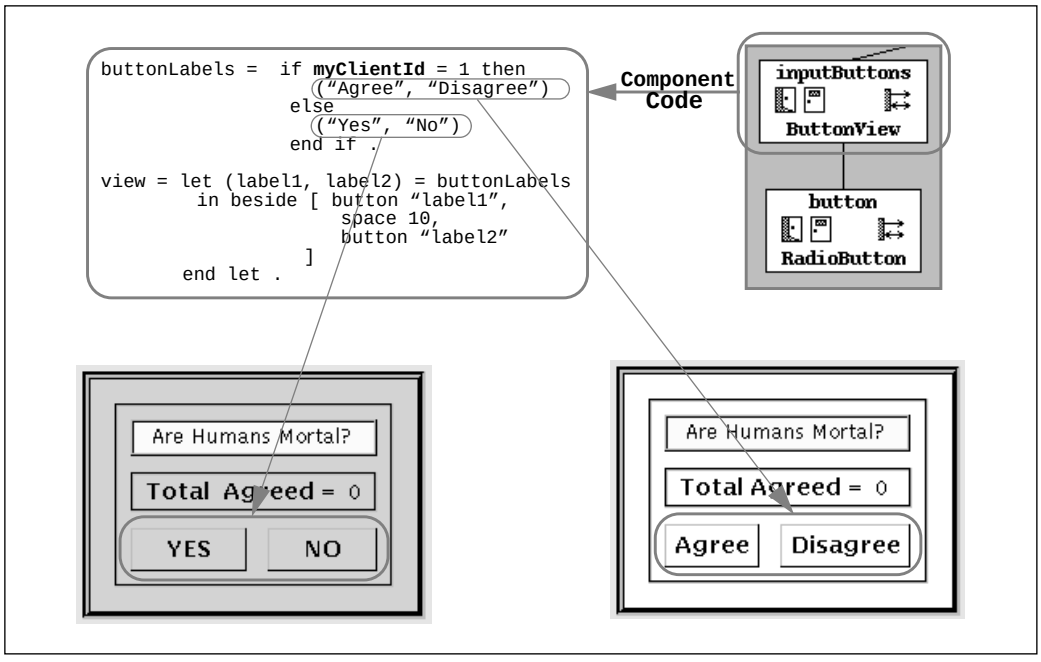



**FIGURE 4.6 Customization of static views in a Multi-User Clock program**

In distributed Multi-User Clock, programmers have the flexibility of choosing where the data should reside. In particular, they have the flexibility of placing the data anywhere on

the local workstation or on the server machine. The support for flexible placement of data is provided by the ClockWorks [Morton 94] visual programming environment. In ClockWorks, the programmer has the advantage of manipulating the placement of any data structure in the architecture tree, using a direct manipulation interaction technique.

The support for the flexible placement of data allows the programmers to place the data at the server or at the client architectures. This mechanism permits the specification of shared and private data. In a groupware application, the shared data is the information which is accessible by all users. The private data, on the other hand, is the set of information to which a particular user has exclusive access. In a Multi-User Clock application, the shared data are maintained by the request handlers which reside in the server architecture. The private data, however, is maintained by the request handlers in the client architecture. Since client architectures are replicated, each client maintains its own copy of private data.

In Multi-User Clock, while shared data can be used to construct dynamic WYSIWIS views, private data make the dynamic WYSINWIS views possible. For example, in the polling application, the poll result information which is maintained by the *Counter* request handler is treated as shared data and hence is placed in the server architecture. However, the state of the radio buttons, which is maintained by the *Selection* request handler, is private data and is only available to individual users. As a result, while all users have the same view of the poll results, the state of radio buttons may be different for each user.

In this section, the polling program was revisited to illustrate how Multi-User Clock provides support for construction of WYSIWIS and WYSINWIS views. It was shown how development of both static and dynamic WYSIWIS and WYSINWIS views are possible in Multi-User Clock.

## 4.4 A Terminal Reservation System

Figure 4.7 shows an example multi-user program built in Multi-User Clock. This system is a Terminal Reservation System (*TRS*), which was designed to be used for the York University undergraduate lab. (The single-user version of the application was designed and implemented by Nicholas Graham. We have extended this application to the multi-user version.) Each student is allowed to have three hours of terminal time per week. The interface of this program consists of three parts: a terminal room map, a reservation panel and a reservation buttons panel. Colour coding is used to show how long each terminal is available. Each terminal's colour ranges from dark green (to show that terminal is available for the next three hours) to grey (meaning that the terminal is not free at all).

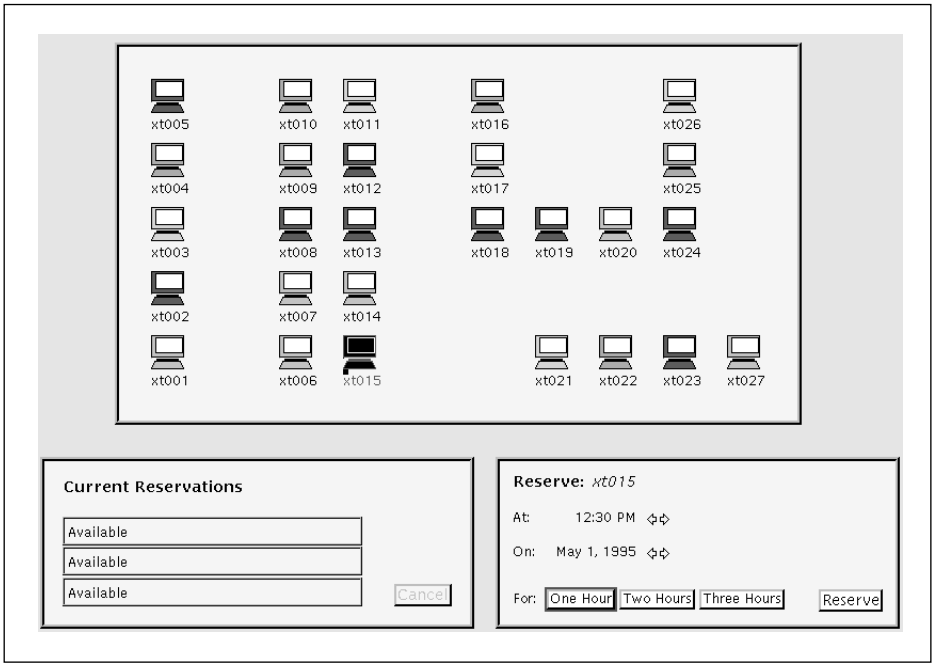



**FIGURE 4.7 A multi-user Terminal Reservation System developed in Multi-User Clock**

The reservation panel shows the reservations which the student currently holds, and allows reservations to be cancelled. The reservation buttons panel allows the student to select a particular day, time and duration in order make a reservation. This example is used to demonstrate how Multi-User Clock provides support for built-in concurrency control,

transparent constraint maintenance and customization of views based on roles and privileges.

**Concurrency Control**

Groupware applications by nature support collaboration and multiple activities among a number of users. These activities are often performed concurrently. This concurrency of events often leads to conflict between parallel processes in gaining access to the services provided by a common resource. Concurrency control is the activity of coordinating the potentially interfering actions of processes that operate in parallel.

The *TRS* program is a typical example of an application in which potential conflicting events can occur. Consider the example in Figure 4.8. In this example two users of the *TRS* program are trying to reserve the same terminal for the same day and time.

In general, there are three types of events in any Multi-User Clock application: input, request and update events. While input events are triggered by the users of the application, the request and update events are triggered by the Clock processes in response to these input events.

Clock applications are reactive systems. This means that the application remains in an *equilibrium* state until an input event is received. In an equilibrium state there are no pending events in the system, and the application is ready to receive a new user input. An input event received by the program triggers a sequence of internal events (updates and requests). The internal events are meant to bring the state of the application and the user-interface up-to-date according to the user input.

In Multi-User Clock, events which are triggered as a result of an input event form an *input thread*. Input threads consist of a series of chained events which are triggered as a result of single user input. To further clarify the notion of input threads, consider the example in

Figure 4.9. This example describes the events which are triggered as a result of the addition of a new reservation.

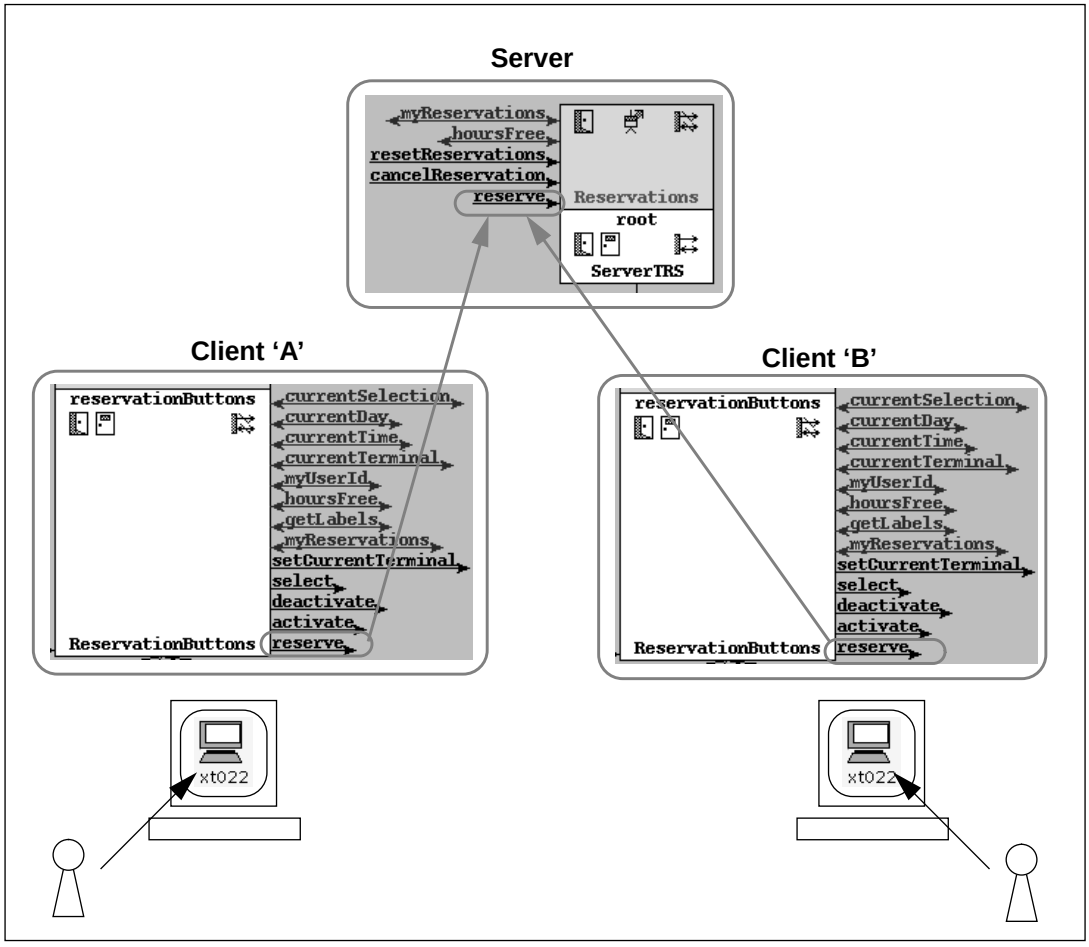



**FIGURE 4.8 An example of an input conflict. In this example both users are attempting to reserve the same terminal for the same day and time.**

Concurrency control in Multi-User Clock operates at the input thread level. Furthermore, the concurrency mechanism in Multi-User Clock is only concerned with concurrent input threads originating from two different client applications. These may trigger an update event which modifies the shared data residing on the server application.

In Multi-User Clock two concurrent input threads are allowed to run concurrently as long as the update events generated by these threads are restricted to the local (private) data.

The *conflicting threads* are input threads which cause concurrent updates to the shared data. The concurrency control mechanism in Multi-User Clock guarantees that all the events pertaining to one of the conflicting threads are serviced, and that a state of equilibrium is reached prior to servicing the events pertaining to a second conflicting thread. This concurrency control mechanism strives to provide the illusion of single threadedness; that is, the illusion that at any given time there is only one active thread in the system.

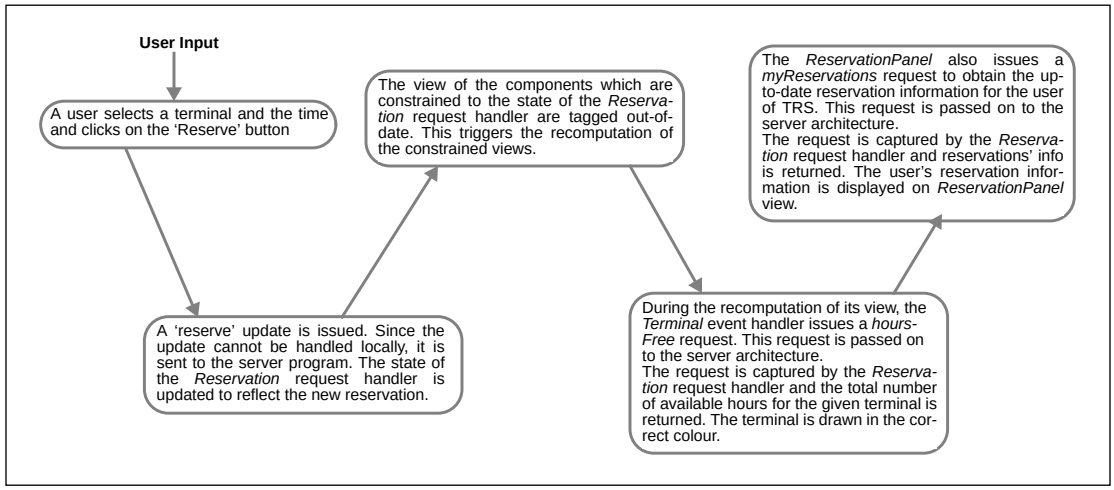



**FIGURE 4.9 An example of an input thread. This input thread is triggered as a result of addition of a new reservation by a user of TRS program**

## Transparent Constraint Maintenance

In Multi-User Clock, the relation between the states and the views which represent these states is maintained via a number of internal constraints. The result is that whenever a display view is out of date, the Clock system automatically forces the view function to be recomputed. The maintenance and triggering of constraints are all transparent to the Multi-User Clock programmer, and are performed automatically without the programmer's intervention.

For instance, in the *TRS* program the colour of each terminal provides some visual cues with respect to its availability. The display view of each terminal (i.e., the terminal display colour) is constrained to the reservations which are currently made for that terminal (i.e., the state of the *Reservation* request handler). The constraint between the terminals' view and the central reservation data spans across multiple users of the *TRS* program. It is crucial for all the users' views to display the correct information with respect to the availability of a terminal to avoid over-booking. Once a user reserves a particular terminal, the display colour of the terminal on all users' interface will change to reflect the new reservation. Figure 4.10 illustrates how transparent constraint mechanisms maintain the correct display colour of terminal views.

In the *TRS* program, the reservation information is treated as the shared data and is kept at the server architecture. To illustrate the reservation information correctly, the view function of *Terminal* component queries the reservation information from the server. This is done via *hoursFree* remote request. The exact code for querying the terminal availability is: `myHoursFree = hoursFree myId currentDay myUserId .`

Once a component has made a request for the state of a request handler, the Multi-User Clock constraint mechanism forms a constraint between the state of the request handler and the component view function which requested it. This constraint guarantees that the user component is informed of any changes made to the state of the request handler in the future.

As shown in Figure 4.10, once a user selects and reserves a terminal, an update (i.e., *reserve)* is issued by the client program to inform the central shared data of the new reservation. The exact code for adding a reservation is: `reserve termId day time userId`.

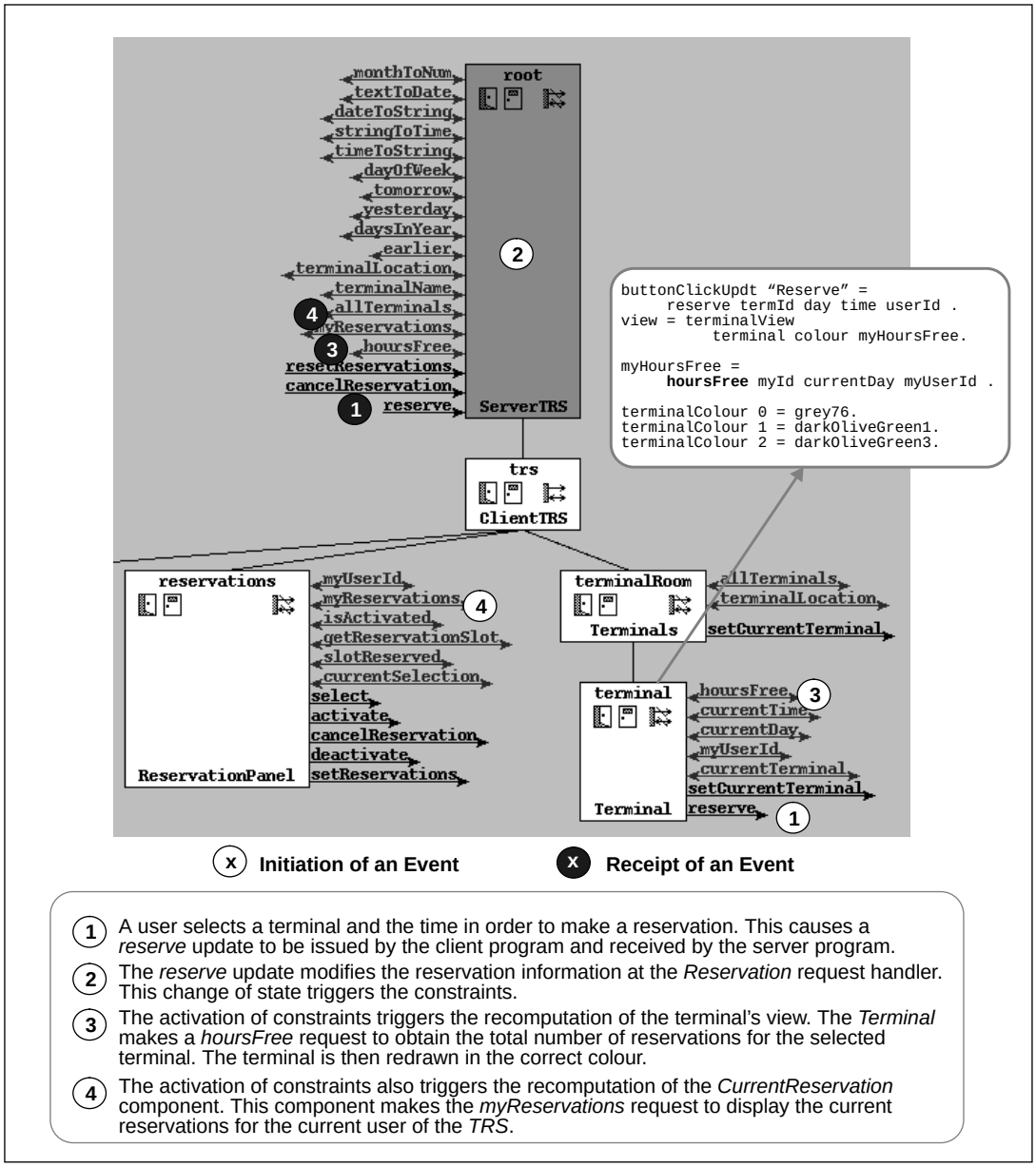



**FIGURE 4.10 An example of the transparent constraint maintenance mechanism in Multi-User Clock.**

Each new update which is applied to the state of a certain request handler activates the constraint bound to the request handler component. Since the views of terminals are constrained to the state of a shared request handler, the constraint mechanism informs the view functions that their views are now out-of-date. Consequently, the view functions request the up-to-date state of the reservations in order to recompute their views. Once

their view is regenerated, the reserved terminal colour is changed to reflect the new reservation.

The transparent maintenance and triggering of constraints in Multi-User Clock relieve the programmers from the low-level details pertaining to consistency control and maintenance of an up-to-date view. The terminal reservation system demonstrated the implementation of transparent concurrency control and transparent constraint maintenance. Transparent concurrency control is designed in such a way as to give the illusion of single threadedness resolution to parallel events which have the potential of conflict. Transparent constraint maintenance is designed to trigger a series of recomputations of interdependent components based on the constraints.

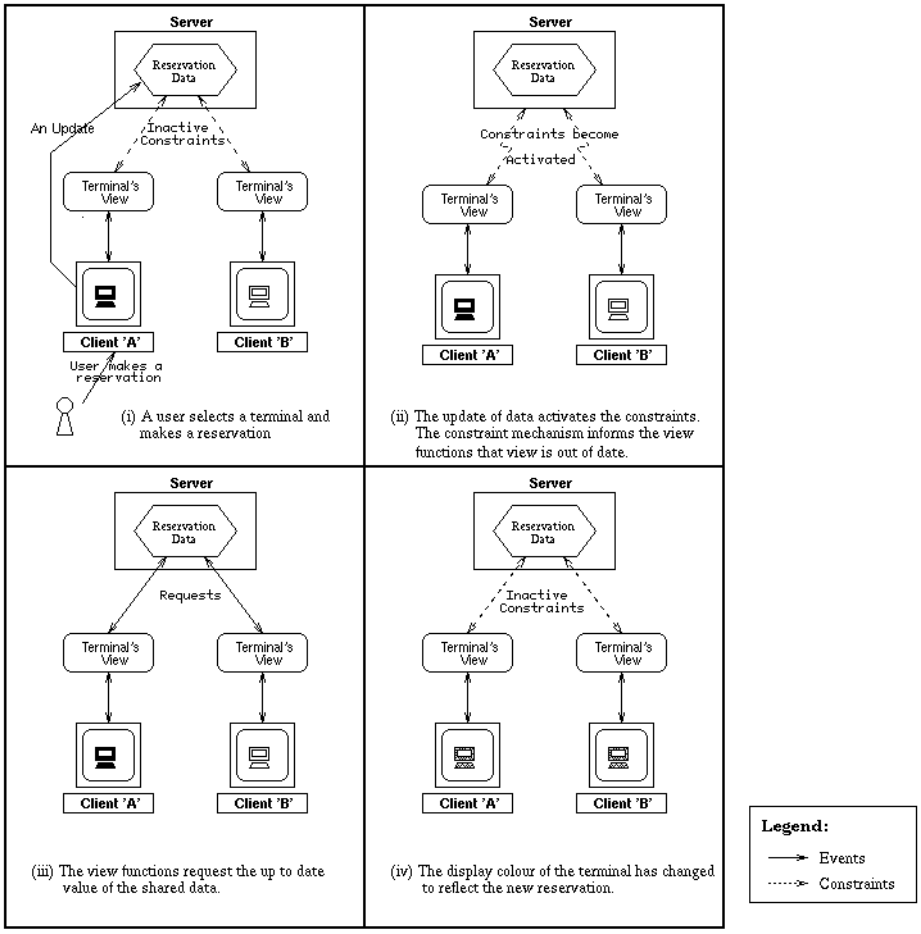



**FIGURE 4.11 The pictorial presentation of constraint maintenance mechanism in Multi-User Clock**

## 4.5 Summary

This chapter discussed how the various design requirements detailed in Chapter 2 are satisfied in Multi-User Clock. The chapter was organized around three applications developed using Multi-User Clock. By means of examples, we showed how Clock provides support for the selected requirements. The selected requirements which are supported in Multi-User Clock are:

- *A High-Level Declarative Programming Language:* Multi-User Clock programmers need only specify a set of rules and logic which specifies the problem without providing additional information as to how to solve the problem. The run-time processes are capable of determining the execution order. Multi-User Clock also provides a high-level programming environment in which all the issues pertaining distribution and remote event handling are transparent to programmers.
- *Transparent Distribution of Application Architecture:* Multi-User Clock provides support for the transparent distribution of the server and the client architectures to their respective machines. The programmers construct and edit the application tree as if they were constructing single-user applications.
- *Transparent Communication Infrastructure:* Multi-User Clock establishes and maintains the communication channels between the distributed multi-user programs. These procedures are all transparent to the programmer. As part of the communication infrastructure, Multi-User Clock manages all communication between the applications transparently.
- *Session Information:* Multi-User Clock programmers have access to the session information which includes the unique clients' identifiers and the clients' names. This information can be queried via a set of built-in requests.
- *Support for Development of Customized Views:* Multi-User Clock provides support for development of customized views. This is achieved via support for construction of both WYSIWIS and WYSINWIS user interfaces.

- *Support for Development of Collaboration Aware Applications:* The construction of collaboration aware applications are supported in Multi-User Clock through shared views.
- *Transparent Constraint Maintenance:* Multi-User Clock provides a transparent constraint maintenance mechanism. This mechanism establishes the dependence relations between the data and the views which represent these data. The mechanism triggers the recomputation of the components' views should the data which they represent change.
- *Transparent Concurrency Control:* The transparent concurrency control mechanism, as implemented in Multi-User Clock, resolves any conflicts which may arise as multiple client processes work concurrently.

The following chapter discusses the implementation of Multi-User Clock.

# Chapter 5

# Language Implementation

## 5.1 Overview

This chapter discusses the various implementation techniques involved in the development of Multi-User Clock. The chapter begins by describing the distribution architecture used in Multi-User Clock and the transparent distribution of application parts based on this architecture. It then proceeds to discuss other technical issues involved in Multi-User Clock's communication infrastructure (such as remote event handling). A description of the technical issues involved in providing a transparent concurrency control mechanism concludes this chapter. Multi-User Clock is implemented on SUN workstations, running under UNIX. The Multi-User Clock is developed using Turing and C programming languages.

## 5.2 Distribution Architecture

One of the primary design decisions in a groupware system is selecting a distribution architecture. The difference between various architecture implementations lies in how the application and its data are distributed over the participating machines, and how these parts communicate. The choice of architecture directly impacts various aspects of a groupware system. For instance, performance has been considered to be one of the most important requirements of distributed groupware systems. Protocols for communication, concurrency and synchronization influence the performance of a groupware system. These protocols, in turn, depend on the architecture of the application. This section discusses the various communication architectures, their advantages and disadvantages, and our design decisions.

There are three basic architecture alternatives to consider when implementing interactive multi-user applications. The first approach is the *centralized* architecture which is basically a client-server architecture where a single instance of the multi-user application is shared by all users [Lauwers 90]. In this scheme, all user inputs are forwarded to the single instance of the application, running on the server machine, and the application's output is multiplexed to all the clients' machines.

The second alternative is referred to as a *replicated* architecture [Crowley 90]. Under this approach, a copy of the multi-user application is replicated, and executed locally on each user's workstation. User input to a shared application is distributed from the user's window system to all instances of the application. Output from each copy (i.e., application feedback), however, is delivered only to the local window system.

The third alternative, the *semi-replicated* architecture, is a hybrid between the replicated and the centralized architectures [Graham 92a]. Under this architecture the application consists of two parts: the server and the client. Also, the application data is distributed among the server and the clients such that the shared data is maintained at the server and the local data is replicated between each client.

All of these methods bear a number of advantages and disadvantages. A brief description of each communication architecture, as well as their advantages and disadvantages, will be presented below.

The architecture of Multi-User Clock applications can be mapped to any of the above mentioned architectures. The current implementation of Multi-User Clock is based on a semi-replicated architecture. The architecture of a program is split into the server and the client sub-architectures which are distributed to the server and the client machines respectively. A schematic presentation of the distributed Clock programs is depicted in Figure 5.1. The figure illustrates Multi-User Clock's two executable programs: the server and the

client program. The figure also shows how the server and the client programs, which run on the server and client machines respectively, communicate with each other using a message passing scheme. In addition, the figure illustrates how the architecture of a Multi-User Clock application is divided into the server and the client architectures, and how the server architecture is used by the server executable program and how the client architecture is replicated to each client executable program.

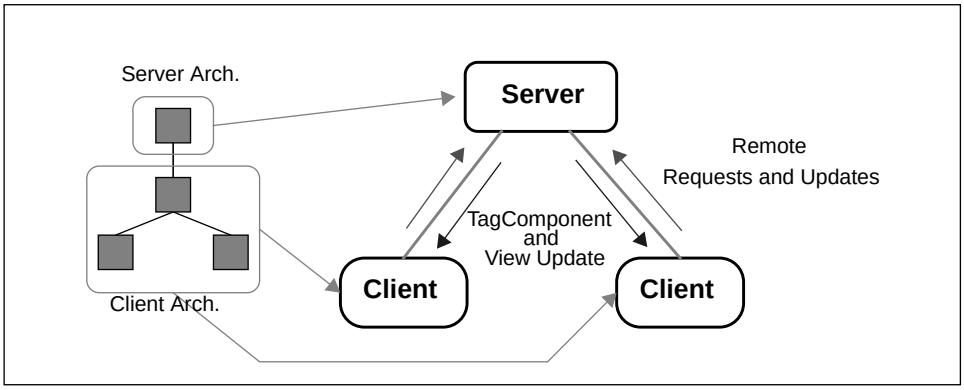



**FIGURE 5.1 The semi-replicated architecture as implemented in Multi-User Clock.**

A description of each of the above architectures is given below to justify our decision. A discussion of server and client programs in Multi-User Clock follows. The server and the client programs each consist of a number of processes, which communicate with each other via a number of messages. A description of these processes and their communication methods is also presented.

### 5.2.1 Centralized Architecture

This architecture is based on a client-server model, in which a single instance of a multi-user application is shared by all users. In this model, each participant's workstation has a minimal program (the client) which handles the screen and accepts the participant's input. The real work of the application is performed by the server, which runs on the central computer and holds all application's data. An example of a system based on a centralized architecture would be the Rendezvous system by Hill, et. al [Hill 92]. Figure 5.2 illustrates

the architecture of a centralized system in a schematic form. As is shown in the figure, user's inputs (from host A and Host B) are passed to the communication manager. The communication manager in turn issues a call to the server, where the application resides. The call to the server is captured by the communication manager at the server (e.g., X-Multiplexor) which in turn issues a local call to the application program. The application program will issue a call-back to the communication manager residing on the server, and the communication manager will dispatch the call-back to the communication manager on the client machine. The communication manager on the client machine at last passes the call-back to the terminal to reflect the changes triggered by the user's input.

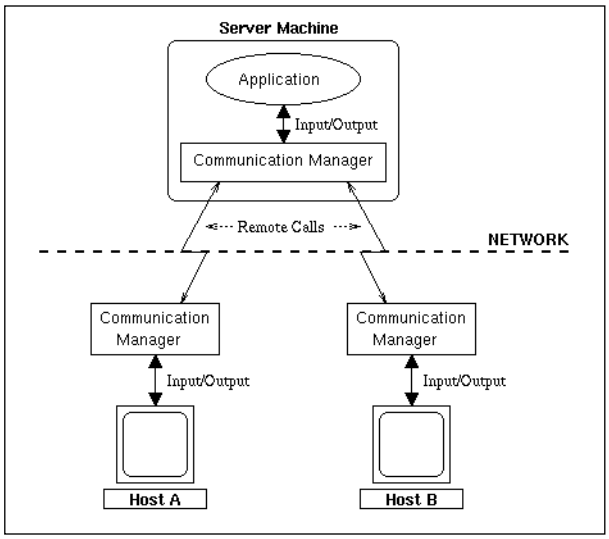



**FIGURE 5.2 Centralized Architecture**

The advantages of having a centralized architecture are:

- *Simple implementation*: This is due to the fact that only one copy of the multi-user application exists. Hence the programmer need not be concerned with issues such as data coherence, concurrency control and synchronization.
- *Good performance with small number of users:* If the number of users of a groupware application is known to be small, a centralized architecture is the best choice. The application programmer can take advantage of the simplicity of the architecture without

sacrificing overall performance. While simplicity is achieved by avoiding complex concurrency and synchronization mechanisms, the performance advantage is achieved by avoiding the overheads which are typically associated with these mechanisms. Under this scenario, it is less likely that participants will experience delayed feedback or feed-through due to network congestion or server overload.

- *Existing Support:* The joint viewing requirement of computer supported cooperative work (CSCW) applications can be addressed by an X Protocol Multiplexor (MUX) [Baldesch 93]. A MUX is a special program which exploits properties of the X Window System to allow joint viewing of X applications. The MUX achieves its objectives by intercepting the X client/server connection. This allows the MUX to filter the calls from the X server to the client and conversely multiplex the call-backs from the client application to all X servers. Note that in X terminology, the X server is the portion of the system which resides on the client machine and the X client refers to the application program which is on the server machine.
- *Easier to port a single-user application to multi-user application:* With the aid of existing support such as *X Protocol Multiplexor (MUX)* a single-user X application can be used in a multi-user setting with minimal modification. However, this mechanism only provides support for development of interfaces with WYSIWIS views. In other words, the feedback generated by the central application, in response to a user action, is multiplexed to all participants' display.

The main disadvantage of a centralized architecture is that it has *poor scalability* [Graham 92a]. Poor scalability can be defined as the inability of a system to maintain a relatively constant performance level as the number of users grow. The poor scalability will result in:

- *Delayed feedback and poor response time with a large number of users:* As the number of users increase, the server and network eventually becomes overloaded and performance degrades. Since all requests must be passed from all clients to the server, and the responses then passed back to the clients, an increase in the network traffic can also be

experienced. This will eventually lead to a scenario in which all users experience a delay in response time (i.e., feedback and feed-through). This is due to the fact that the centralized architecture does not take advantage of the larger number of available processors as the number of users increases.

### 5.2.2 Replicated Architecture

In this model, each workstation runs its own copy of the application. These copies need to communicate in order to keep their data structures consistent with one another. Systems with a replicated architecture need an efficient synchronization and concurrency control mechanism to ensure that replicas remain synchronized, consistent, and that they process inputs from different users in the same order. Such mechanisms are usually hard to implement. The synchronization and consistency maintenance in these applications is likely to have an impact on applications' performance. In a replicated architecture, each replica handles its own user's feedback, as a result this architecture provides the most rapid feedback. Each replica must also update the screen in response to messages from other replicas. Examples of applications which employ a replicated architecture are: the MMConf System by Crowley, et. al [Crowley 90], and GroupSketch by Greenberg et al [Greenberg 92]. The remainder of this section discusses the advantages and disadvantages of having a replicated architecture. Figure 5.3 illustrates the replicated architecture in pictorial format. As is shown in the figure, all the calls (generated as a result of user's input) are serviced by the local copy of the application. The localization of the services naturally leads to faster response time and hence better performance since the user is able to see the result of his/her action faster(e.g., faster screen updates). As is shown in the figure however, the application replicas need to interact with each other to ensure data coherence. The line across the network which links two communication managers together represents the replicas inter-communication. The remainder of this section discusses the advantages and disadvantages of having a replicated architecture.

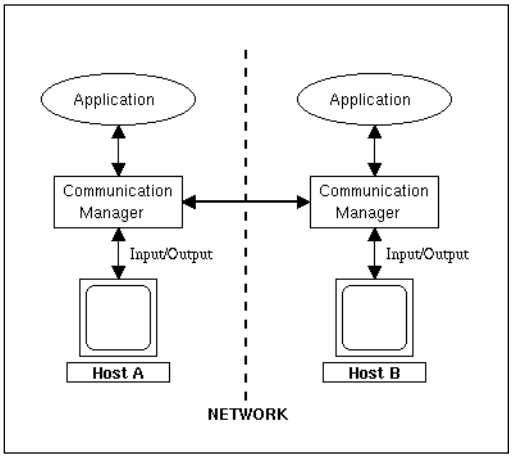



**FIGURE 5.3 Replicated Architecture**

The advantages of a replicated architecture are:

- *Good Performance with Large Number of Users:* The main advantage of the replicated architecture is its rapid response time and fast feed-through which, in turn, translates into improved performance [Crowley 90]. Response time is better than in a centralized system because all requests and computations are captured and serviced locally by the local copy of the shared application. As a result, the response time is independent of the number of participants. In contrast to the centralized architecture in which output data must be sent across the network to all window servers (as implemented on each participant's workstation), in a replicated architecture each replica sends output only to the local window server, and no output data needs to be distributed across the network (excluding the communication between replicas for concurrency control).

- *Versatility:* In contrast to centralized architectures, replicated architectures work equally well with *server-based* or *kernel-based* systems. In a server-based window system, input events and output requests pass through a network connection, which provides a good handle for obtaining the output to distribute (e.g., X Windows). These systems are the only environments in which a centralized architecture can be utilized. In a kernel-based window system, library or system calls handle input events and output requests, leaving no handle on the output (e.g., Microsoft Windows). Hence, a centralized architecture cannot function in a kernel-based system. Replicated architectures,

on the other hand, need only a handle on the input, which is much easier to intercept (window systems typically offer a single entry point for obtaining input, but a multitude of entry points for generating output). More importantly, however, replicated architectures appear much better suited to accommodating heterogeneous hardware environments. “Replicated applications running on different workstations with window systems need only translate the input events. Centralized systems on the other hand must struggle even to adapt to minor discrepancies in display geometry or colour maps” [Lauwers 90].

- *Fault Tolerance:* Under this architecture each client machine has an identical copy of the application and the data. Hence, failure of a single processor/client will not lead to the failure of the application as a whole. Under the centralized architecture, however, the failure of the server machine will result in failure of the whole application. In a replicated architecture, failure of one client will lead to termination of the application on the faulty machine only.

The disadvantages of a replicated architecture are:

- *Difficult Implementation:* The replicated architecture is difficult to implement [Crowley 90, Lauwers 90]. The difficulty arises since one needs to maintain consistency among all application replicas. In addition, the concurrency maintenance requires sophisticated schemes to prevent concurrent updates by two users. The concurrent events often lead to *race conditions*, in which two input events compete to make use of services provided by a common source. A CSCW system must properly handle race condition.
- *Multiple Semantic Operations:* In addition to the above mentioned problems, replicated architectures further complicate matters by executing semantic operations several times, once for each application copy. Semantic operations are those operations which influence the state of the system [Lauwers 90]. For example, consider a scenario in which the users of an application with a replicated architecture need to read a data file. In this setting, we need to open/read/close the file as many times as the number of cli-

ents. A related problem is maintaining *single-execution semantics* [Lauwers 90]; that is, to ensure that executing multiple replicas would have the same effect on the global application state as running a single replica.

- *Lack of a Globally Available State:* Full replication also leads to difficulties when bringing late-comers into a group session. Since there is no globally available state, fully replicated systems may have to keep a full history of actions performed by all users in order to allow a late-comer to attain the state currently held by other participants. A global state represents the state of the application as a whole.

These disadvantages will lead to such impediments as: mis-ordered input events, non-deterministic applications and accommodation of late-comers into an active session [Crowley 90]. Despite the potential benefits of replicated architectures, most groupware applications have avoided this type of architecture, due to the significant synchronization problems that are associated with replicated execution [Lauwers 90]. Keeping replicas of a shared application consistent requires that each replica receive (semantically) equivalent input from each input source, and that input events originating from different sources be delivered to all replicas in the same order. The reader is advised that many of these issues are application dependant.

### 5.2.3 Semi-Replicated Architecture

The problems associated with the two previous architectures have motivated the implementation of a hybrid system which adapts the benefits of the two basic architectures while avoiding their drawbacks. The new approach is known as a semi-replicated architecture. In a semi-replicated architecture, only a portion of the application and data are replicated to each client. The most typical approach is to keep the main application and the shared data on the server, while the code and the data related to user interface are replicated to client machines. Examples of systems which make use of this architecture are the Weasel System [Urnes 92] and the Suite System [Dewan 92b].

Figure 5.4 illustrates how a semi-replicated architecture is implemented in Multi-User Clock. In this model, the executable programs and the architecture tree are divided into two portions: the client and the server. While the server executable and the server portion of the architecture tree are kept on the server machine, the client executable and the client portion of the architecture tree are replicated to each client. As is shown in the figure, the server and the client executable programs communicate with each other via a set of messages. The communication between the server and client programs is handled by the server and client communication managers respectively. The communication managers' task is to hide distribution and networking issues from both the server and the client programs. Note that here the server and the client programs refer to Multi-User Clock executable programs, not the application program which is written by Multi-User Clock programmers.

We believe that employing a semi-replicated architecture has a number of advantages over the centralized and/or the replicated architectures. These advantages are listed below with annotation to identify the respective architectures.

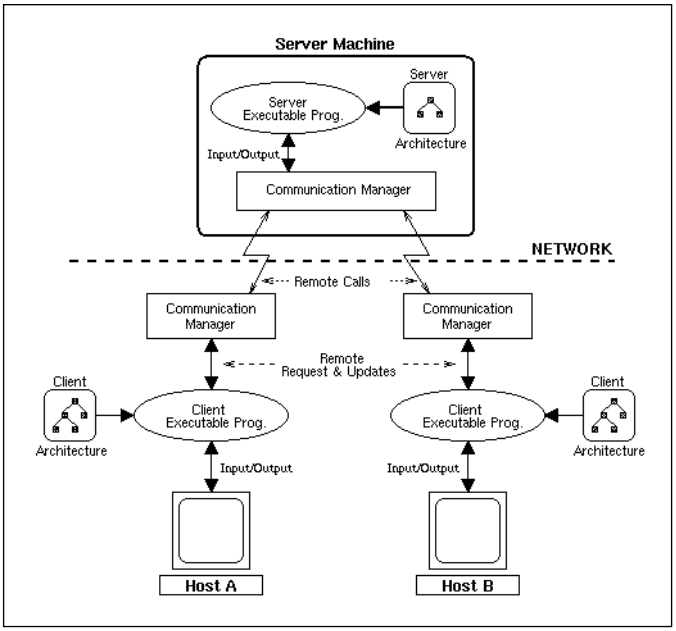



**FIGURE 5.4 Semi-Replicated architecture in Multi-User Clock**

- *Improved Scalability (vs. Centralized):* Compared to a centralized architecture, the semi-replicated architecture maintains better performance as the number of users increase. Empirical studies of Weasel [Urnes 92], a system with a semi-replicated architecture, have shown that adding new participants to a session does not degrade the overall system performance as much as it would in a centralized setting.
- *Reduced load on the server (vs. Centralized):* In comparison with centralized architectures, in a semi-replicated architecture the load on the server is possibly reduced by placing some of the computation on client machines.
- *Improved accommodation of late-comers (vs. Replicated):* In a semi-replicated architecture, the application state is divided into a shared *global* state at the server, and a *local* state for each participant at the client. The global state represents the state of the shared data while each local state represents the state of private client data. Thus, the local state of two applications can be different. Under a semi-replicated architecture, a late-comer is brought up-to-date by obtaining the global state from the server and initializing a new local state.
- *Simplified consistency mechanism (vs. Replicated):* Under the semi-replicated architecture, the shared data is placed at the server. Since there is only one copy of the shared data, there is no need for a mechanism to ensure consistency among shared data replicas as required in replicated architectures.

Semi-replicated architectures are not, however, free from faults. Under the current implementation of Multi-User Clock, when a client program issues a remote request call it blocks, and the client program remains idle, until it receives the result of the request from the server. The latency associated with this mechanism may degrade the application's performance in two ways. First, in certain applications (e.g., shared editing) the client processes need to constantly poll the central server for the up-to-date state of the shared data (i.e., edited document). As a result of high frequency of the client calls to the server, the cumulative idle time will result in poor response time. Second, as the number of the users

grows, the server processes need to process more number of calls from clients. This will result in a longer idle time as client processes wait for a response from the server.

## 5.3 Transparent Distribution of Application Architecture

Section 5.2.3 described the semi-replicated distribution architecture as it is implemented in Multi-User Clock. This section presents the communication protocols in the semi-replicated implementation of Multi-User Clock. Moreover, it shows how the client and the server components of the architecture tree are distributed to their respective sites in a transparent fashion. Multi-User Clock consists of two distinct executable programs: the server program and the client program. These programs reside and run on the server and the client machines respectively.

Clock programmers use the ClockWorks visual programming environment to construct the application architecture tree. ClockWorks generates an *architecture description file* in which all the architecture tree specifications are saved in textual format. This file contains all the information pertaining to all the classes, request handlers and event handlers for both the server and the client programs. Figure 5.5 shows how this file is used by both the server and the client programs to extract architecture information.

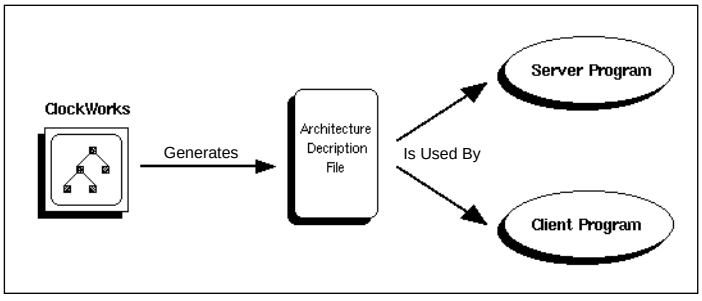



**FIGURE 5.5 ClockWorks generates an Architecture Description File, in which all the architecture specifications are stored in textual format. This file is used by the server and client programs to extract architecture specifications.**

The programmer does not need to specify two distinct architectures. ClockWorks automatically annotates the components as being a server or a client component. As shown in Figure 5.1, both the server and client programs infer their component architectures from the architecture description file upon initialization.

The server and client programs are only aware of the part of the component architecture which pertains to their program. These two programs communicate with each other via request and update messages.

To further familiarize the reader with the functionality of these two programs, a brief description of their start-up procedure (already outlined in Section 4.2) follows. The server and client programs have similar start-up procedures. Figure 5.6 illustrates the server programs's initialization procedure. The syntax for invocation of the server program is: `goServer <program name>`, where `<program name>` is the multi-user program which an end-user wishes to run. For instance, to run the server program of the shared drawing program, an end-user would type: `goServer MU-Draw`.

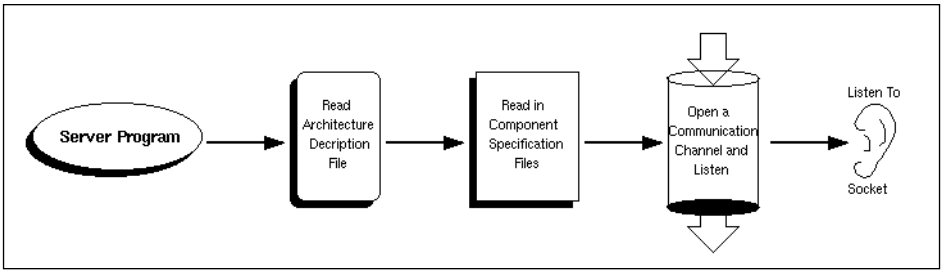



**FIGURE 5.6 Clock server program's initialization procedure. The cylinder with two arrows represents a communication channel.**

As is shown in figure 5.6, upon invocation, the server program first reads in the architecture description file which is generated by ClockWorks. The server program then reads in all the specification files pertaining to the different server components. The component specification files are written by Multi-User Clock programmer as part of the application

being developed. It then opens a communication channel (a socket) and registers its name with the operating system. This allows future connection requests to be routed to the server program process. Once the communication channel is established, the server program listens to the socket for any connection request originating from the client programs.

The client program's initialization procedure is shown in Figure 5.7. The syntax for invocation of a client program is: `goClient <program name> <client name> <server host>`. The `<program name>` is the multi-user program which an end-user wishes to run. The `<client name>` is a string identifier which is chosen by the end-user. The Multi-User Clock processes maintain a unique numerical identifier for each client. Since mapping a client numerical identifier to actual participants is rather hard for the application users, Multi-User Clock allows the users of a program to assign a preferred name to themselves. This information is maintained as part of the session information and can be queried by the Multi-User Clock programmers. The `<server host>` is the unique network name of the server machine where the server program is being run. For instance, if the server program of the shared drawing program is known to be running on `tivoli.cs.yorku.ca`, an end-user can invoke the client program by typing: `goClient MU-Draw "Roy" tivoli.cs.yorku.ca`.

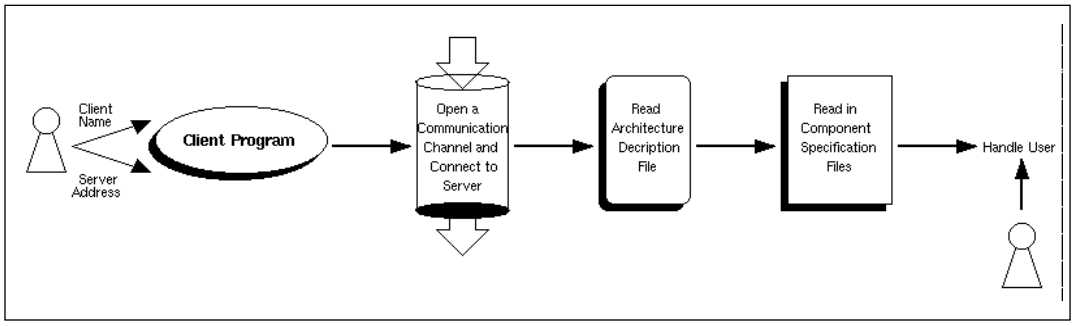



**FIGURE 5.7 Clock client program's initialization procedure. The cylinder with two arrows represents a communication channel.**

As is shown in figure 5.7, upon invocation, the client program creates a communication channel in order to establish a dialogue with the server program. Having done this, it then reads the architecture description file and specification files pertaining to all the client components. After successful completion of the initialization procedure, the client program initializes the display view and begins to accept and process user inputs.

The server and client programs each serve a different purpose. The primary purpose of the client program is to provide an up-to-date view of the user interface, and to obtain and process user inputs. The client program is also responsible for receiving the updates sent from the server and modifying the user interface accordingly. The server program, on the other hand, acts as the central data repository where all the shared data are stored. The server program services client requests for the state of the shared data, and also multicasts the updates made to the shared data to all clients.

## 5.4 Communication Infrastructure

Sections 5.2 and 5.3 described the client and the server programs, and showed how the application architectures are distributed and used by these programs in a transparent fashion. In Multi-User Clock, each of these programs consists of a set of processes. More specifically, Multi-User Clock applications consist of several processes arranged in a semi-replicated architecture. The communication infrastructure in Multi-User Clock serves two objectives: first, to establish, maintain and terminate communication channels between remote processes, and second, to handle remote events in a transparent fashion. This section describes how the communication infrastructure in Multi-User Clock meets its first objective. Section 5.5 will discuss the transparent event handling mechanism.

We will now examine a process view of the communication architecture. Some example scenarios are presented to elaborate on the processes' behaviour. Our goal is to describe the motivations behind each of these processes and explain their tasks. In addition, we

describe how these processes communicate with each other and how their tasks are synchronized.

### 5.4.1 Server Processes

This section introduces the different processes that comprise the server run-time program. The server run-time program consists of a number of light-weight Turing processes. These processes execute within and share the same address space. The invocation of the server program instantiates one process on the server machine. The purpose of this process, the *client registrar process*, is to listen to the server socket for any connection request from the clients. Once the process detects a new client's connection request, it initiates a client connection procedure and forks a new process responsible for communicating with the new client. This procedure is called client-server *handshake*. The handshake procedure involves assignment of a unique identifier for the newly connected client, informing the client of the new identifier, obtaining the new client's name and including the client's data in the central data base. After successful completion of the handshake procedure, the server's registrar process forks a new process which handles all communication calls from the client in the future. This process is named the *communication handler process*. Hence, at any given time for *N* clients, there are *N* concurrent communication handler processes and a total of *N+1* processes at the server. This naturally limits the system's scalability to a maximum *MAX_USER_PROCESSES-1* clients, where MAX_USER_PROCESSES is the maximum number of processes allowed by the system, in which the server program runs.

The remainder of this section briefly describes the client-server handshake procedure. Figure 5.8 illustrates the client-server handshake procedure in pictorial format. A handshake procedure begins with one client sending a connection request to the central server. The server acknowledges the connection acceptance by sending a "*connectionGranted*" message. This message also contains the unique identification of the client which is assigned by the server. This unique identification is used in all future client-server communications.

Having received this message, the client sends another message informing the server of its preferred '*name*'. Although client names are not used by the internal processes, they are available to the programmers. After successful completion of the above message exchanges, the server's registrar process forks a new process to handle all the communication to/from the newly connected client.

All information pertaining to the active clients (clients with live communication sockets) is kept by the central registrar process at the server. The programmer can query this information through a set of built-in requests, such as: *myClientId*, which returns the unique client identifier in a session, and *myClientName*, which returns the name which end-user has chosen during the start-up procedure.

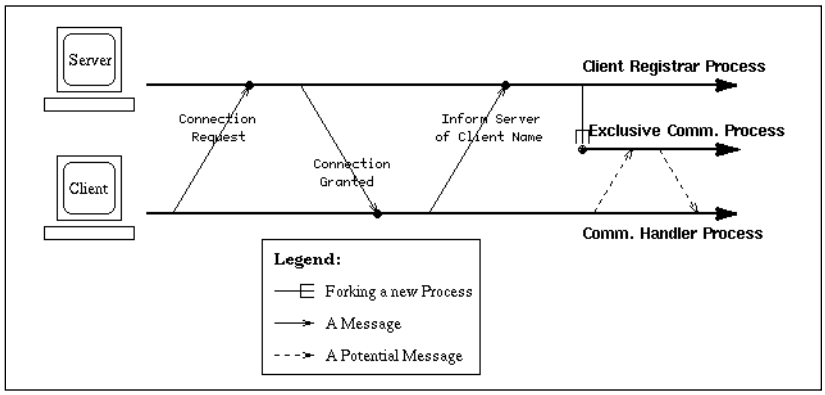



**FIGURE 5.8 Client-Server handshake procedure**

### 5.4.2 Client Processes

Each client application consists of two light-weight Turing processes. These processes execute within and share the same address space. These processes communicate with each other via shared memory. Each of these processes serves a different purpose. This section introduces these processes and discusses their functionality. The client processes are:

- *Event Handler Process:* This process is instantiated after completion of a communication handshake with the server. The purpose of this process is to handle all user interactions (local events) and update the interface accordingly. This process is unaware of the distribution issues. Note that this process has nothing to do with event handler components.
- *Communication Handler Process:* This process is instantiated during the client program initialization. The purpose of this process is to handle all communication between the client event handler process and the remote server process. As well, it acts as a mediator between the local event handler process and the remote server process, making the distribution transparent to the local event handler process.

To further illustrate the functionality of these processes we revisit the shared drawing program introduced in Chapter 4. Recall that the drawing application would allow all the users to concurrently draw objects on their own display canvas. Due to the WYSIWIS nature of this application, all objects are synchronously displayed on all of the participants' screens. Consider the example in Figure 5.9. This example illustrates how processes of the client program handle two input events: a local input event (local user draws a rectangle) and a remote input event (a remote user draws another rectangle). Since the state of the drawn objects are considered to be shared data, addition of an object will cause a remote update which passes the information pertaining to the newly drawn object to the server processes.

In the case of the first input event, the event handler process obtains the local user's input. In this example, the local user draws a rectangle. The screen is updated accordingly to reflect the changes made by the user. Since the drawing program provides WYSIWIS views, the information pertaining the drawn objects are treated as shared data and are kept at the server. Hence, after applying the changes to the local interface, an update is passed to the communication handler process to inform the server process of the changes.

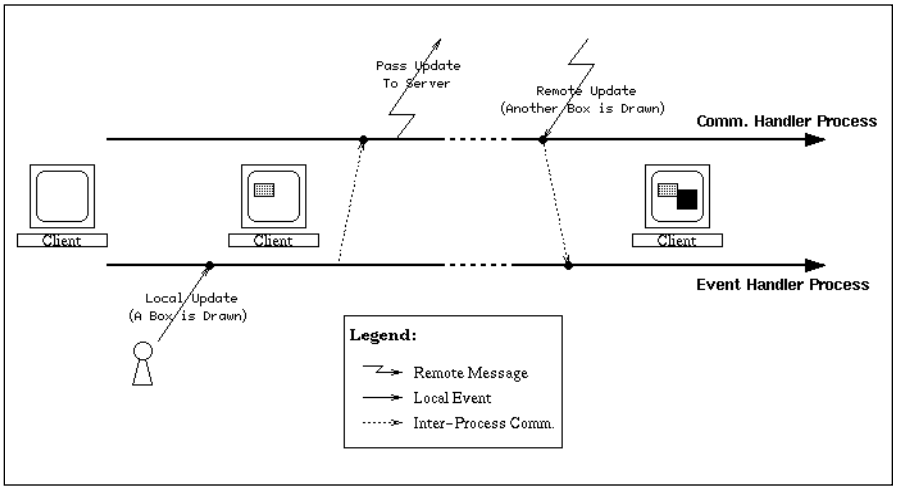



**FIGURE 5.9 Client program processes**

Now consider the second event: an update made by a different user. The communication handler process receives an update from the server indicating addition of another rectangle by some other user. The communication handler process passes the information to the event handler process, which in turn initiates a screen update to reflect the remote update.

## 5.5 Transparent Event Handling

Multi-User Clock can be described (in distributed system terms) as a loosely coupled system in which each client has (and operates in) its own address space. The server also operates in its own address space. The central server and the clients communicate with each other via remote events. These events, which consist of *remote requests* and *updates*, are those which are initiated locally by a site (client or server), and are serviced remotely.

As part of the built-in communication infrastructure, Multi-User Clock provides support for transparent handling of local and remote events. The event handling mechanism extends over network boundaries so that it is able to handle remote requests and updates transparently.

To further clarify the remote message handling in Multi-User Clock, we revisit the shared drawing program introduced in Chapter 4. Each of the remote message descriptions is accompanied by an example of a typical execution of the shared drawing program. In what follows, a brief description of the drawing program architecture, and the client and server programs pertaining to this application is provided.

Figure 5.10 illustrates the architecture of the shared drawing program. To simplify the architecture, only remote requests and updates are shown in the picture. As shown in the figure, the *addObject* update, which adds a new object (box or a line) to the central shared data, is treated as a remote update. In addition, the *numObjects*, *getObject* requests, which query the quantity and the coordinates of the objects respectively, are treated as the remote requests.

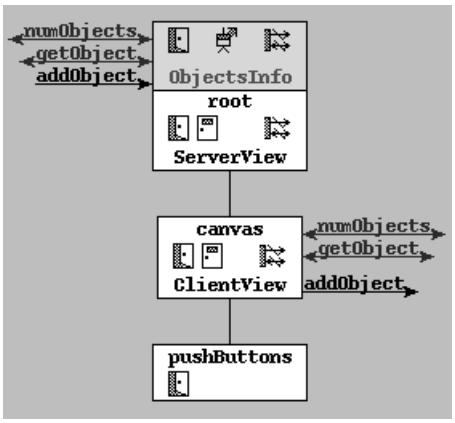



**FIGURE 5.10 The architecture of the shared-drawing program, showing the remote requests and updates.**

When a client program determines that an event cannot be handled locally, the event is assumed to be handled by the server program and hence is sent to the remote server machine. In response to an update event, the server program initiates a tagging procedure to inform the remote clients' components of the update. The following sections present a discussion of the client and the server remote calls.

### 5.5.1 Message Structure

In this section we briefly describe the Clock message structure. All communication messages in Clock follow a predefined format. All the data included in a message are of character type. Any other data types are converted to the character format prior to their inclusion in the message. Figure 5.11 presents a pictorial description of the Clock messages. In what follows we describe each data field included in a Clock message.

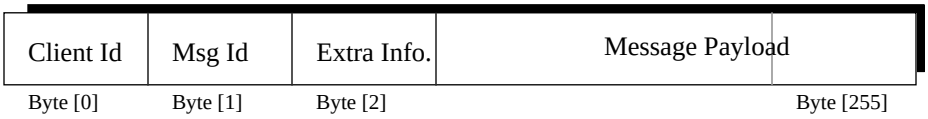



**FIGURE 5.11 Message Format**

- *Client Id:* This field holds the id of the sending host. Each client is assigned a unique id during the initial connection with the server. Id (0) is reserved for the server.
- *Message Id:* This field indicates the message payload type. Each of the client's and the server's messages are given a unique identifier. This field is used by the message recipient to determine the type of the message payload.
- *Extra Info:* Several messages need to carry additional information. For instance, when passing arguments over the network it would be convenient to know the number of these arguments when they are being extracted from a message. The high number of messages which require an additional form of information justifies the use of this field.
- *Message Payload:* This field, which is 253 bytes long, contains the actual message. Placing a limitation on the message length has a number advantages and disadvantages. The disadvantage is that messages which are longer than 253 bytes must be split into two (or more) 'shipments'. Also, for any message length less than 253 bytes, we still need to send 253 bytes. The advantage is that both sender and recipient know the length of each message in advance.

### 5.5.2 Remote Event Handling

This section describes the remote events which are initiated by a client machine and are executed at the server machine. Client programs may issue two remote calls: a remote request or a remote update. While a remote update modifies, a remote request queries the state of the shared data at the server. These remote calls are discussed in more details in the next two sections.

#### 5.5.2.1 Remote Update Handling Mechanism

The goal of this section is to introduce and discuss the remote update handling concept and mechanism. The aim is to demonstrate how shared data are updated in Multi-User Clock.

Figure 5.12 illustrates a schematic presentation of a remote update handling mechanism in Multi-User Clock. A remote update is considered to be an update event which is issued by a client program to make changes to the shared data residing on the server. As is illustrated in the figure, a remote update mechanism begins with an '*update*' message followed by zero or more '*update argument*' messages, and concludes with an '*end update argument*' message. To further clarify remote update handling in Multi-User Clock, we incorporate an example pertaining the shared drawing program and show how a simple update (i.e., *addObject* update) is handled in the Multi-User Clock.

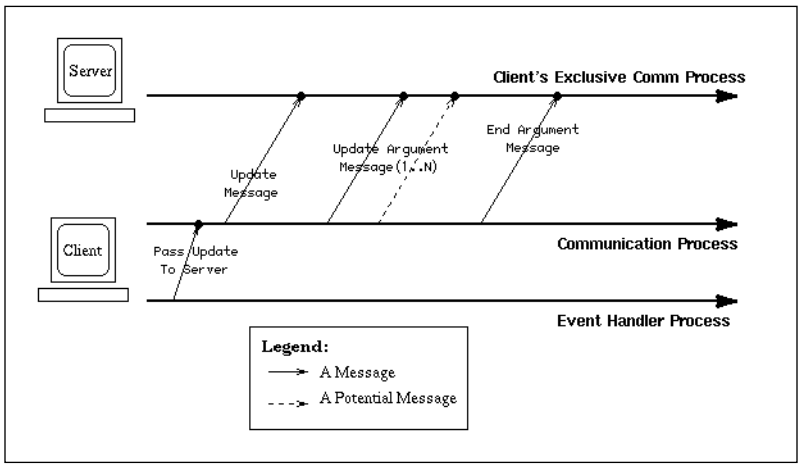



**FIGURE 5.12 Remote update mechanism in Multi-User Clock.**

Figure 5.13 shows the update message specifying the addition of a box on the display canvas. A remote update procedure begins with the client sending an '*update*' message to the server. This message conveys a variety of information. First, it informs the server process that the client wishes to make a change to the shared data. Second, it carries the update identifier, allowing the server process to know which update must be applied. All update events are internally represented with a unique numerical identifier. The *addObject* update is internally represented as update number 14. Third, it advises the server of the number of argument messages which follow this update message. The updates often carry an arbitrary number arguments. Each of these arguments are sent via a separate message. The number of arguments in the update message instructs the server process to defer the update procedure (in the case where one or more arguments are to follow), or to initiate an update procedure immediately.

If the update message has a number of arguments associated with it, the server process initiates a loop to obtain all the arguments. The update arguments are sent in an *updateArg* message form. In the case of *addObject* update there are four arguments: the object's type, the object's owner and two end coordinates of the object. Figure 5.14 illustrates a possible

example of an *updateArg* message pertaining to the *addObject* update. This example shows the *updateArg* message for one of the end coordinates of a newly drawn object.

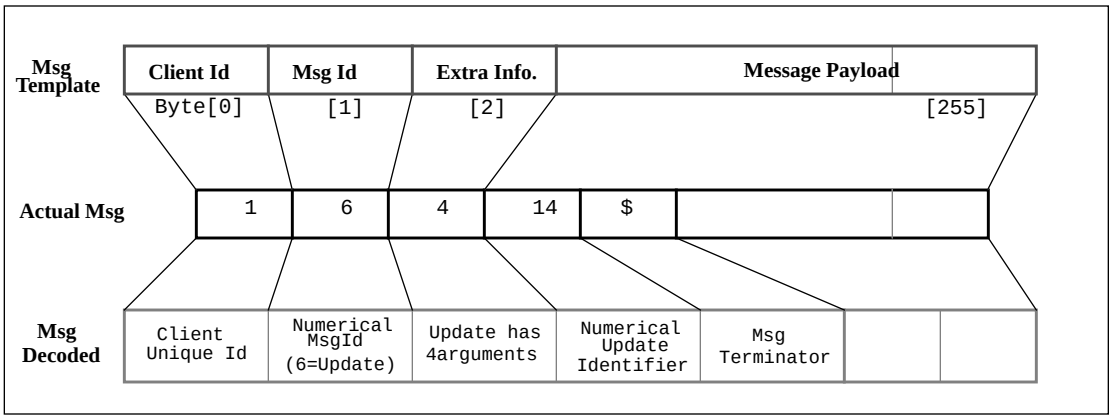



**FIGURE 5.13 An example of an update message. This update message pertains to the *addObject* update of the shared drawing program. This update is internally represented with update number 14. This update also has four arguments which correspond to the type, owner, and coordinates of the object.**

An *endUpdateArgs* message terminates the loop and signals the server to initiate the update procedure with the given update and arguments. No extra message is sent back to the calling client to inform it about the status of the update.

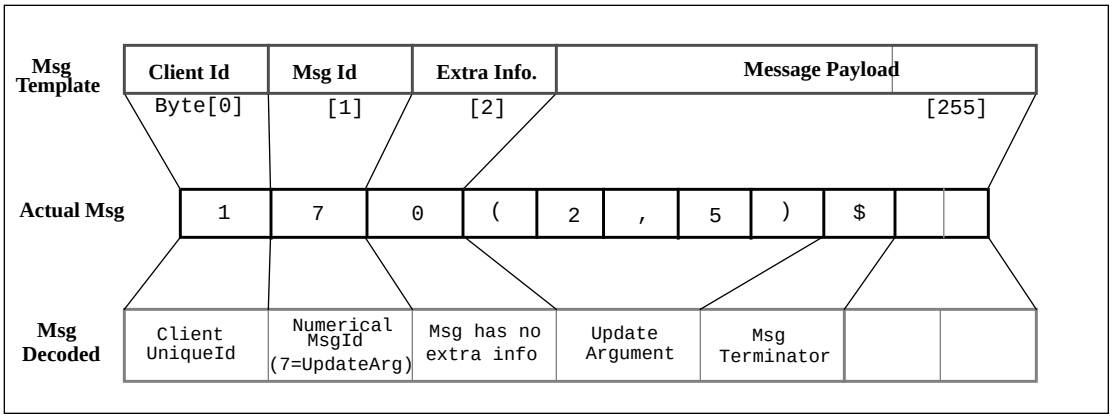



**FIGURE 5.14 An example of *updateArg* message. This message shows the third argument of *addObject* update. The third argument carries the coordinates of the one end point of the object, in this case (2, 5).**

#### 5.5.2.2 Remote Request Handling Mechanism

This section describes the remote request handling mechanism in Multi-User Clock. It also describes the protocols and mechanisms used to implement this feature. Figure 5.15 presents an schematic presentation of a remote update handling mechanism in Multi-User Clock.

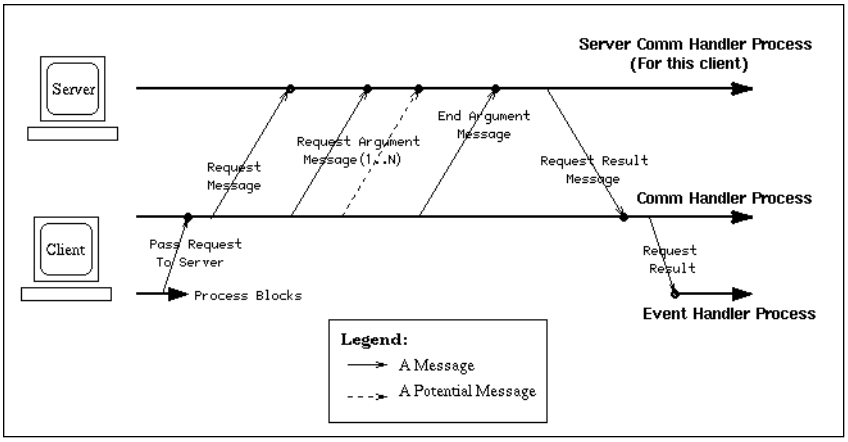



**FIGURE 5.15 Remote request handling. The client first sends a request message to the server. Subsequently, an argument message is sent for each argument associated with the request. The server then returns a message containing the result of the request.**

A remote request is considered to be a request event which is issued by a client program to query the state of a request handler residing at the server architecture tree. To further clarify remote request handling in Multi-User Clock, we incorporate an example pertaining to the shared drawing program to show how a simple request (querying the coordinates of an already drawn object) is handled.

A remote request handling procedure is initiated by a client program upon realization that it is incapable of servicing a request locally. Consequently, the client program sends a *request* message to the server program. A request message informs the server of a client's need for the certain shared data. This message carries the request identifier (each request has a unique identifier) and the number of arguments which the client will be sending following this message. If the number of arguments is determined to be zero, the server process initiates request handling immediately; otherwise it proceeds to receive the expected

number of arguments. Figure 5.16 illustrates a typical request message issued by a client of the shared-drawing program.

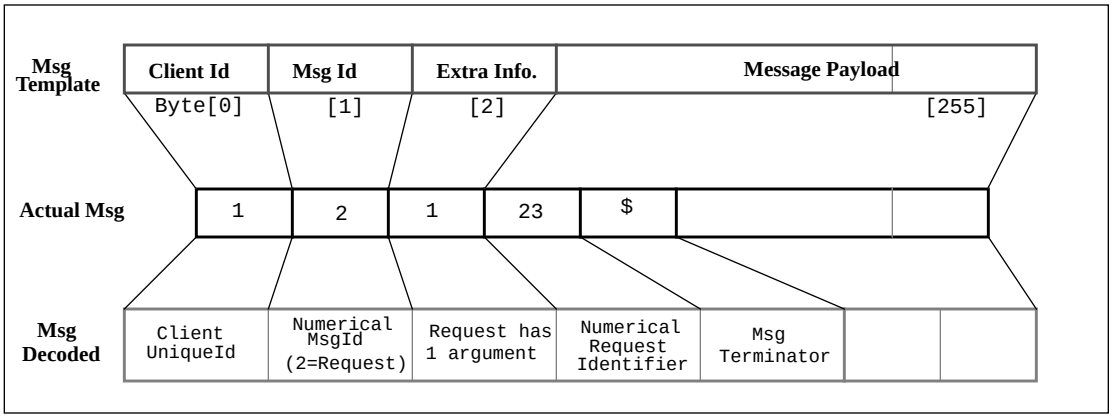



**FIGURE 5.16 An example of a request message. This request message pertains to the *getObject* request of the shared drawing program. This request is internally represented with request number 23. The request also has one argument which corresponds to the object number.**

If the request carries one or more arguments, a *requestArg* message is sent following a *request* message. This message carries one of the arguments associated with the sent request. The *endRequestArgs* message informs the server of the termination of all request arguments. After receiving this message, the server proceeds with the request handling process. The server returns the result of the request via a *requestResult* message. Figure 5.17 illustrates a typical server message which may be sent in response to the *getObject* request of Figure 5.16.

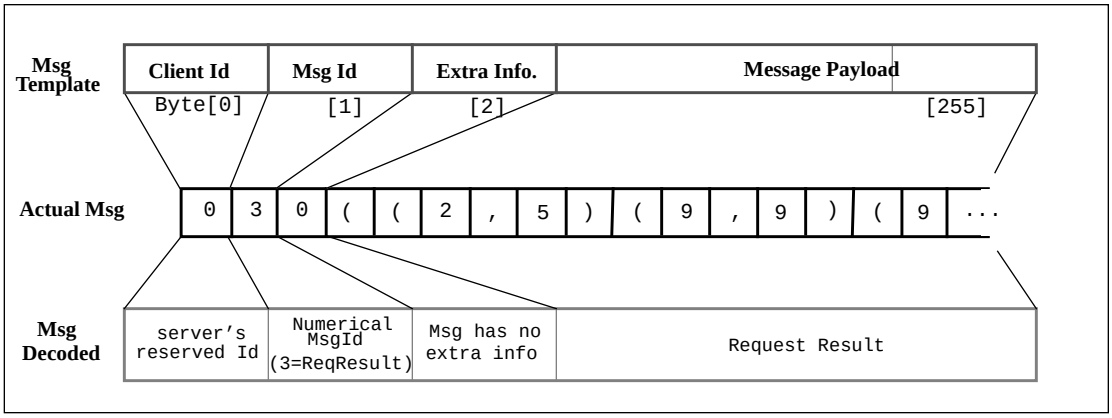



**FIGURE 5.17 An example of a *requestResult* message. This message returns the type, owner and the coordinates an object in response to a *getObject* request.**

#### 5.5.2.3 Session Information

As it is described in Chapter 4, Multi-User Clock programmers have access to the session information. For instance, in the shared drawing program of Chapter 4, it was shown how the unique client identification number of each client program was used to assign a unique colour to objects drawn by the user of a client program. Multi-User Clock programmers can query the session information via a set of built-in requests. These requests are used with the same syntax as other requests in Multi-User Clock programs. A brief description of these built-in requests are presented in this section.

- *myClientId:* This local request returns the unique id of the client. This information is determined during the connection establishment with the server and is kept at the client machine.
- *myClientName:* This local request returns the ‘name’ of the client. This information is passed to the client application as command line argument during the program invocation and is kept at the client machine.
- *allClientId:* This remote request returns a list of all client unique ids. This information is maintained by the server application.
- *clientIdToName:* This remote request takes a client id as its argument and returns the client ‘name’. All client information is maintained by the server application.

### 5.5.3 Constraint Maintenance Mechanism

This section describes the techniques used in implementing the transparent constraint mechanism. In particular, it illustrates how constraints are generated and maintained in Multi-User Clock. There are two remote messages which are implemented as part of the constraint maintenance mechanism. These messages which are issued by the server program and are executed at the client machines, are: *tagComponent* and *updateView.* A *tagComponent* informs the client’s components of a recent change on the shared data. An *updateView* triggers a view recomputation at the client’s sites. As described in Section 4.4,

constraints are relations which link data to their visual representations. Figure 5.18, illustrates the constraints between the views and the shared data in a schematic format.

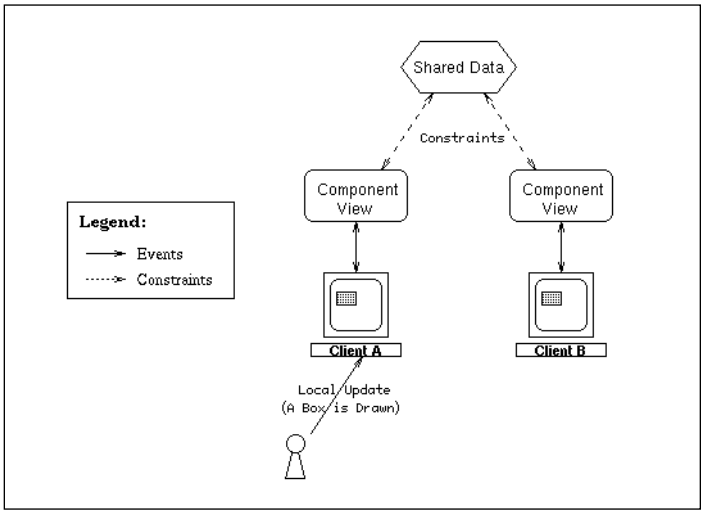



**FIGURE 5.18 Constraint mechanism in Multi-User Clock.**

In Clock, request handlers are abstract data structures where data is kept. The constraint mechanism forces each persistent data structure (request handler) to maintain a dynamic list of components which make use of their services (i.e., a user's list). These lists are transparently maintained by Multi-User Clock. Figure 5.19 shows a schematic view of the user's list maintained by the *ObjectsInfo* request handler of the shared-drawing program.

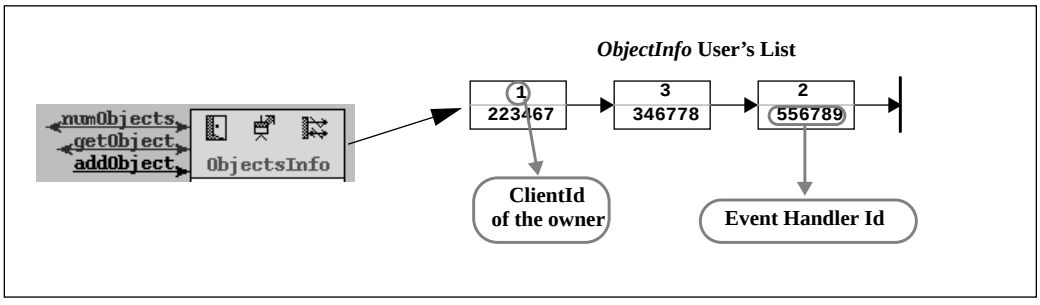



**FIGURE 5.19 An example of a request handler's user list. In this example, the first node in the list indicates that event handler [223467] which resides in the client architecture of client 1, has previously made a request handled by the *ObjectsInfo* request handler.**

Should one of the Clock data structures be modified, the Clock run-time processes make use of these users' lists to inform the client component of the changes made to the data structure. Clock notifies these components by invalidating their old copy of the data. This mechanism is called component tagging. This, in turn, initiates a procedure forcing the components to obtain the updated value from the request handler. This procedure will also lead to a re-evaluation of the functions which makes use of the affected components.

#### 5.5.3.1 Remote Component Tagging

To further illustrate the tagging procedure, consider an example in which a client of the shared drawing program makes an update to the shared data (e.g., addition of a new box). This update triggers a tagging procedure, by the server (communication handler) process, in order to invalidate the data kept by the users of the *ObjectsInfo* request handler. In order to tag a remote component the server program walks through the user's list of the *ObjectsInfo* request handler and sends a *tagViewUser* message to each of the client programs whose name is found in the user's list. The purpose of this message is to inform the clients of the changes made to the shared data. This message triggers an action to tag those client components whose view functions are dependant on the server data that was just updated. Figure 5.20 presents a schematic presentation of a typical *tagViewUser* message issued by the server program.

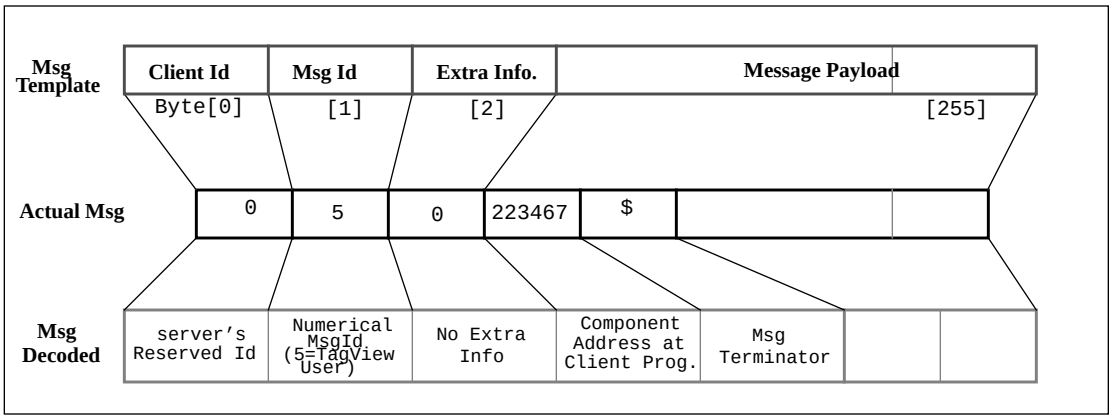



**FIGURE 5.20 An example of a *tagViewUser* message.**

#### 5.5.3.2 Remote Invocation of View Regeneration

The *ViewUpdate* message, initiated by the server (communication handler) process, causes client programs to update their interface views. This call is forwarded to those clients with a component which makes use of the shared data that has been recently updated. This call is typically issued following an invalidation of the client copy of the shared data (i.e., *tagViewUsers*). The invalidation, in turn, is performed following an update on the shared data. Any input thread is concluded by triggering a view update for all the clients' interface which contain components that are effected by the updates initiated by the input thread.

## 5.6 Concurrency Control

As mentioned in Chapter 2, groupware systems should provide support for: (i) concurrent access to the shared data, and (ii) a mechanism guaranteeing fair access to shared data. Fair access refers to preventing any one user (client) from monopolizing the use of central resources. Concurrency control mechanisms are a set of constraints which are imposed on the concurrent processes to satisfy these requirements.

This section describes the concurrency control requirements and mechanisms in Multi-User Clock, and illustrates how these mechanisms satisfy the above two requirements. The section begins by presenting the terminology and implementation assumptions that are relevant to the concurrency control mechanism. We then proceed to identify the concurrency requirements in Multi-User Clock, and how our solution satisfy these requirements. A description of the implementation of the concurrency control in Multi-User Clock concludes this section.

### 5.6.1 Definitions

The goal of any concurrency control mechanism is to maintain: *temporal order* and/or *causal order* among a number of concurrent processes. Both of these orderings attempt to

define the *precedence* (or *happen before,* $\rightarrow$ ) relation between a number of concurrent events. In the following definitions we assume that each process executes its events sequentially. Furthermore, sending and receiving a message in the system are each assumed to be one atomic action. These relations are defined as follows.

- *Temporal Order*: Is the ordering of concurrent events, according to a common (universal) clock. This states that if an event $e_1$ occurs at time $t_1$, and another event $e_2$ occurs at time $t_2$ and $t_1<t_2$ then $e_1 \rightarrow e_2$, (i.e., $e_1$ *happens before* $e_2$) and hence $e_1$ must be processed before $e_2$.

- *Causal Order*: Is the ordering of concurrent events, according to their *causal relation.* The following description is a variation of casual relation defined in [Lamport 78]. In Multi-User Clock there are three types of events: requests (*req*), updates (*updt*), and user input (*inpt*). A user input event triggers a number of updates and requests to be issued in the system. These chained events form an input thread (see Section 4.4). The causal relation is defined as: (i) if $inptThrd_1$ and $inptThrd_2$ are two input threads initiated at the same client, and the client executes $inptThrd_1$ prior to $inptThrd_2$ then $inptThrd_1 \rightarrow inptThrd_2$, (ii) if $inptThrd_1$ and $inptThrd_2$ are two input threads initiated at two different clients, and both cause an update on the shared data, then if the update of *inptThrd1* gets executed at the server first then $inptThrd_1 \rightarrow inptThrd_2$, else $inptThrd_2 \rightarrow inptThrd_1$, (iii) if $inptThrd_1 \rightarrow inptThrd_2$ and $inptThrd_2 \rightarrow inptThrd_3$ then $inptThrd_1 \rightarrow inptThrd_3$, in other words, the ( $\rightarrow$ ) relation is transitive; (iv) within each input thread the causal relation is as follows: $inpt \rightarrow updt \rightarrow req$. (v) two events $e_1$ and $e_2$ can be executed concurrently if $(e_1 \not\mapsto e_2)$ and $(e_2 \not\mapsto e_1)$. To illustrate the causal relation between concurrent events, consider the example presented in Figure 5.21. The figure shows three processes and a number of events initiated in the system. In particular the figure illustrates three input threads caused by input events 1.1, 2.1, and 3.1. While input threads caused by inputs 1.1 and 1.3 contain updates which modify the shared data, input thread caused by input 1.2, causes only local updates. Thus, the causal relation for client *B* is: $inptThrd_1 \rightarrow inptThrd_2 \rightarrow inptThrd_3$. However, causal relation for client *A* is:

$inptThrd_1 \rightarrow inptThrd_3$. In Clock, the concurrency control mechanism preserves causal relation between input threads. This may violate the temporal order among the two input events (with respect to a common/universal clock).

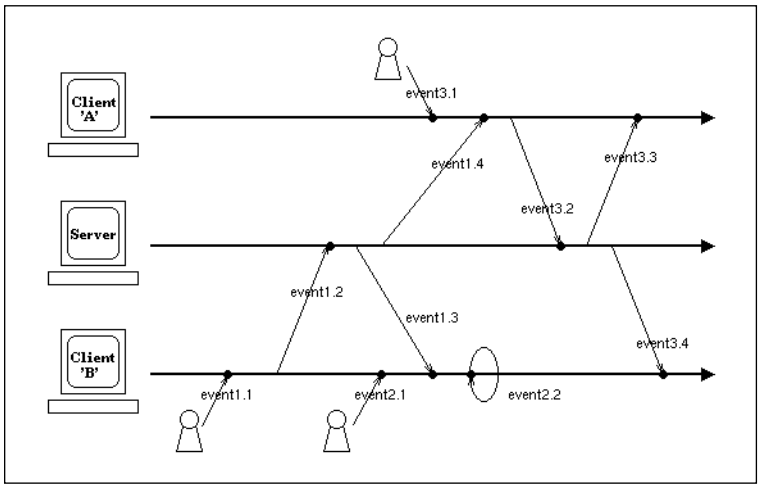



**FIGURE 5.21 Event diagram showing causal relation between input threads in Multi-User Clock applications. Input threads caused by input 1.1 and 3.1 cause updates to shared data. The input thread caused by input 2.1 however makes local updates only.**

In addition to temporal and causal relations we also need to define the following:

- *ReadSet (inptThrd)* = Set of all the request handlers that will be *queried* as a result of all the requests triggered within *inptThrd*.
- *ModSet (inptThrd)* = Set of all the request handlers that will be *modified* as a result of all updates triggered within *inptThrd*.

### 5.6.2 Assumptions

The concurrency control requirements and mechanisms which are presented in the following sections are based on a number of assumptions. These assumptions are:

- We assume a perfect client/server network. In particular, our model does not consider any network and/or machine failures.
- We assume a perfect message handling system in which all message/packets will get delivered to their respective destinations in the same order as they are generated, with no packet loss.

- The number of the input events generated by the users of a CSCW application is insignificant compared to the number of events which are generated by internal processes. Most concurrency control methods which are designed for the CSCW applications have been borrowed/inspired from the distributed systems domain [Greenberg 94]. In this domain, the researchers are concerned with providing concurrency control among events generated by different processes. There are two major differences between the two systems: (i) In a given period of time, the system processes generate more events than events produced as a result of human inputs in an interactive application. (ii) The processes generate events more frequently than humans (i.e., machine processes produce events at a rate of one per fraction of a second vs. CSCW users who generate inputs at one per every few seconds).

### 5.6.3 Concurrency Control Requirements in Clock

Here we present the concurrency control requirements for the Multi-User Clock system. These requirements are a simplified extract from the formal semantics of the Clock language which is presented in [Graham 95].

Two input threads $inptThrd_1$ and $inptThrd_2$ can be processed concurrently iff:

$$\begin{aligned} &ModSet(inptThrd_1) \cap ModSet(inptThrd_2) = \varnothing \ \wedge \\ &ModSet(inptThrd_1) \cap ReadSet(inptThrd_2) = \varnothing \ \wedge \\ &ReadSet(inptThrd_1) \cap ModSet(inptThrd_2) = \varnothing \ . \end{aligned}$$

That is the set of request handler components which are modified or queried as a result of events triggered by $inptThrd_1$ and the set of request handler components which are modified or queried as a result of events triggered by $inptThrd_2$ must be mutually exclusive.

### 5.6.4 The Concurrency Control Mechanism in Multi-User Clock

The concurrency control mechanism, as it is currently implemented in Multi-User Clock, is based on the *serialization* model. There are a number of serialization algorithms developed in research and practice. A full discussion of these concurrency algorithms can be

found in [Bernstein 87]. Based on the terminology of [Bernstein 87], the adapted serialization model can be considered to be *non-optimistic*. The serialization algorithms work by synchronizing events so that atomic transactions (which may consist of several events) are executed serially across the entire system. The input threads form the atomic events in Multi-User Clock. Hence, the concurrency control mechanism ensures that the input threads are executed serially. However, events within each input thread may be executed in parallel. The concurrency control mechanism ensures that all events will be received by all processes in the same order. This guarantees that events are executed in the correct order by all processes, and that no further repair is needed in the future (i.e., all the executions are final). However, this assurance is likely gained at the cost of slowing down the execution of a sequence of events.

The serialization of events is achieved by enforcing a locking mechanism for the use of the server resources. This method provides privileged access to the shared objects for a period of time. Hence, at any given time, only the lock holder(s) can have access to the shared objects. Two distinct types of locks are employed in the implementation of the Multi-User Clock: *Write Lock* and *Read Write Lock*. Placing a write lock on the server (i.e., shared data) prevents any client process (including the lock holder) from modifying (writing) the shared data. In other words, when a write lock is issued, the clients can only query (read) the shared data. Placing a read-write lock on the server, provides exclusive read/write access to the shared data for the lock holder. Hence, when a read-write lock is issued no client other than the lock holder can query or modify (read or write) the shared data.

Based on this locking mechanism, at any given time, a server program can be in one of the three possible states: *Idle, WriteLocked, ReadWriteLocked*. Figure 5.22 presents a table which illustrates server states and all permissible operations during these states.

.

| *Client Operations* / *Server States* | **ReadWrite Lock Request** | **Write Lock Request** | **Release ReadWrite Lock** | **Release Write Lock** | **Request** | **Update** |
|---|---|---|---|---|---|---|
| **IDLE** | GRANTED | GRANTED | ERROR | ERROR | ERROR | ERROR |
| **ReadWrite Locked (by client C)** | QUEUED | QUEUED | RW Lock RELEASED | ERROR | If requester=C then service request else ERROR | If updater=C then service update else ERROR |
| **Write Locked (by [LockHolders])** | QUEUED | GRANTED | ERROR | W Lock RELEASED | If requester in [LockHolders] then OK else ERROR | ERROR |

**FIGURE 5.22 Possible server states and permissible operations during these states.**

In order to obtain a lock, each client process needs to issue a lock request. There are two types of requests for two types of locks: *WriteLock Request* and *Read-WriteLock Request*. While in the idle state, the server grants both write lock and read-write lock on a first-come-first-serve basis. Although multiple write locks may be granted at a time, only one read-write lock is granted at any given time. This implies that the clients can query the state of the shared data in parallel, but only one client can modify the shared data at any given time. Once the process with the lock no longer needs access to the shared object, it *releases* the lock. As is shown in the table, after granting a read-write lock the subsequent requests for the same type of lock are queued.

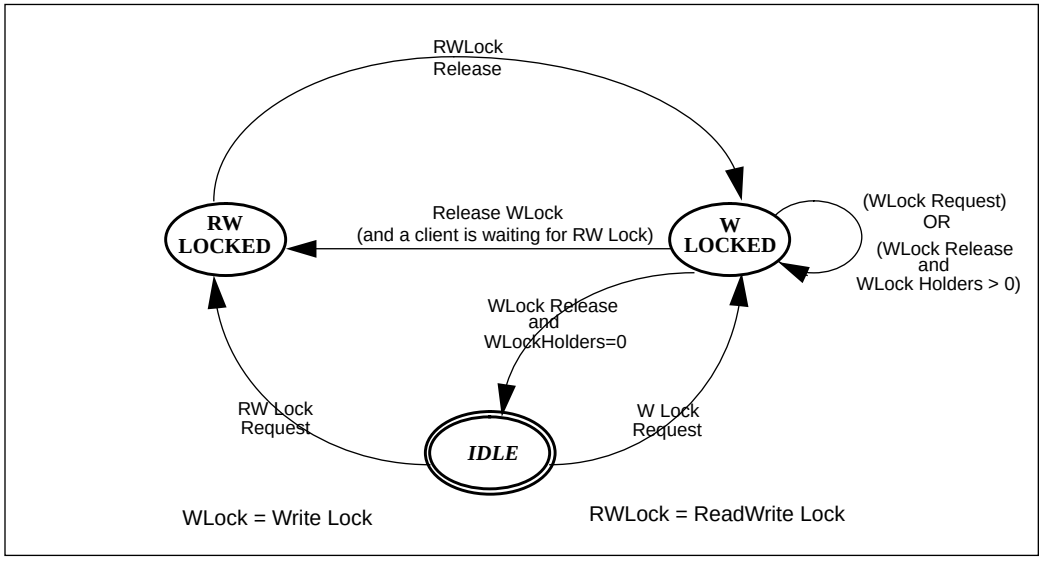



**FIGURE 5.23 The state diagram showing the possible server states and transitions between states.**

Figure 5.23 illustrates the state diagram for the server program. While in an *Idle* state, the server may grant a write lock or a read-write lock. Granting a read-write lock will change the server's state to the *ReadWriteLocked* state. While in this state, the server does not grant any additional locks. Once the lock is released (i.e., *ReadWriteLockRelease*) by the holder, the state of the server is modified to *WriteLocked* state. This automatic transition permits all the client sites to obtain write locks, and hence update their views prior to admittance of a second read-write lock. While in a *WriteLocked* state, the server grants multiple write locks. This is to allow the server to service the requests of all clients in parallel. The state of the server is not altered until all write lock holders release their locks. Once all the lock holders release their locks, if there is a pending request for a read-write lock the server's state is altered to a *ReadWriteLocked* state to accommodate the request. Otherwise, the server returns to an *Idle* state.

Figure 5.24 presents a process view of the concurrency control mechanism in Multi-User Clock. As can be seen in the diagram, while only one read-write lock is granted to a client process, multiple write locks may be granted to the client processes.

The concurrency control method in Multi-User Clock enforces a *non-optimistic* locking mechanism. This method forces the requesting site to wait until a request is granted before it is allowed to manipulate the shared objects. In current implementation of Clock, this request mechanism is *blocking*. That is, after issuing the request the requester should wait for a grant before it proceeds with manipulation of the object. This can be improved by employing a *non-blocking* request handling mechanism. A non-blocking locking mechanism permits the requester to perform other actions while waiting for the resources to be granted. However, under no circumstances should the requester be allowed to manipulate the object prior to receiving a lock grant.

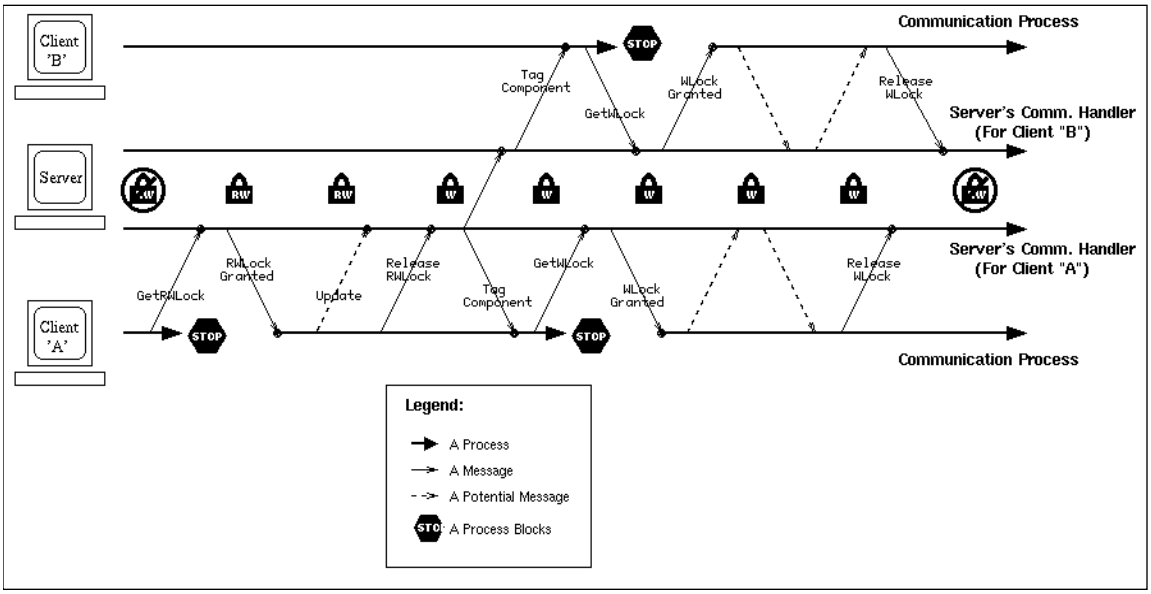



**FIGURE 5.24 A process view of concurrency control mechanism as implemented in Multi-User Clock. The figure shows that the lock request mechanism is blocking. Also it illustrates that while multiple client processes can obtain a write lock, only one client process may have a read-write lock at a time.**

The concurrency control mechanism, as implemented in Multi-User Clock, satisfies the concurrency requirements which were presented in section 5.7.2. To illustrate this point, consider that two input threads $inptThrd_1$ and $inptThrd_2$ initiated in two client programs: $CP_1$ and $CP_2$. According to the specified requirements, we need to ensure that:

$$\begin{aligned} &ModSet(inptThrd_1) \cap ModSet(inptThrd_2) = \varnothing \ \wedge \\ &ModSet(inptThrd_1) \cap ReadSet(inptThrd_2) = \varnothing \ \wedge \\ &ReadSet(inptThrd_1) \cap ModSet(inptThrd_2) = \varnothing \ . \end{aligned}$$

Let us further assume that $inptThrd_1$ is registered earlier than the $inptThrd_2$. Based on our assumptions, we can make this arbitrary decision. As described in section 5.7.4, the serialization of the input threads guarantees that one input thread is completed prior to admittance of a second input thread. Thus, a second input thread will not be serviced while the server is servicing the calls pertaining to a prior input thread.

Prior to applying any update to the shared data, $CP_1$ needs to obtain a read-write lock. Let us further assume that the server is in the *Idle* state and a read-write lock is granted to $CP_1$. As a result, the state of the server is altered to a *ReadWriteLocked* state. While in this state

no additional lock is granted. Upon completion of the update, $CP_1$ issues a read-write lock release. The state of the server is then automatically changed to *WriteLocked* state. While in this state no read-write lock is granted. After all of the client programs obtain the new state of the shared data, the write lock is released and the state of the server is changed to *Idle*. Only at that time may a read-write lock be granted to $CP_2$.

## 5.7 Summary

This chapter presented a detailed description of implementation techniques employed in Multi-User Clock. It began by providing a description of several existing distribution architectures and their advantages and disadvantages. Later, it described how Multi-User Clock is implemented to provide a semi-replicated architecture. A description of server and client programs and processes within these programs was presented. It was also shown how distributed processes communicate with each other via a message passing mechanism. A description of the concurrency control mechanism concluded this chapter. As mentioned in Chapter 3, the current implementation of Multi-User Clock is a prototype and Multi-User Clock applications have poor performance.

# Chapter 6

# Summary and Conclusion

## 6.1 Overview

This chapter presents a summary of the thesis, and its contributions to the field of groupware toolkit design. It also suggests future research directions for Multi-User Clock.

## 6.2 Thesis Summary

Groupware and multi-user applications are inherently difficult to develop due to the need to support concurrent collaborative activities between distributed users. The design of groupware applications is also, by its very nature, multi-disciplinary. Having a set of tools available to groupware developers will accelerate the development process and lead to a better product. The tools necessary for developing groupware must provide support for the transparent distribution of application parts, a built-in communication infrastructure, and support for the development of collaboration aware applications.

Multi-User Clock assists developers by providing a high-level of abstraction which hides the low-level details of networking and concurrency. This mechanism relieves the programmer of the problems inherent in distributed systems and allows her/him to concentrate on the application itself. More specifically, Multi-User Clock provides a mechanism for the transparent distribution of application codes and data, and a communication infrastructure which handles all internal events in a transparent fashion. Multi-User Clock provides a built-in concurrency control mechanism, and a transparent constraint mechanism, to maintain consistency amongst the application's components. Support for these requirements is intended to give the illusion that the developer is designing a single-user application which is to be run on a single machine. Other groupware requirements, which are

addressed in Multi-User Clock include support for the development of customized views and collaboration aware interfaces.

## 6.3 Thesis Contributions

Multi-User Clock provides support for a number of groupware requirements which have not been addressed by existing groupware toolkits. Identifying and supporting these requirements provides insight and guidance for future generations of groupware toolkits. The particular features of Multi-User Clock, which can be extended to other environments, include:

- *Use of Declarative Languages:* There are many difficulties associated with the development of groupware and multi-user applications. Developers of such applications should be able to express what they ‘want’ to achieve, rather than ‘how’ to achieve it. Declarative techniques, as used in Multi-User Clock, allow issues of replication, communication, concurrency control and consistency maintenance to be largely hidden from the programmer.
- *Transparent Event Handling:* Multi-User Clock offers a transparent communication infrastructure. This includes a number of facilities which handle establishment, maintenance and termination of communication channels between distributed programs. But more importantly this infrastructure allows all the internal events to be handled in a transparent fashion. As far as a programmer is concerned there is no difference in definition or handling of an internal event (i.e. requests and updates). This mechanism abstracts all of the networking and communication issues from the programmer.
- *Constraint Maintenance:* In Multi-User Clock, the relation between the states and views which represent these states is maintained via a number of internal constraints. The result is that whenever a display view is out of date, the Multi-User Clock system automatically forces the view function to be recomputed. The maintenance and triggering of constraints are all transparent to the Multi-User Clock programmer, and are performed automatically without the programmer’s intervention. This mechanism relieves

groupware programmers from maintaining consistency amongst the user interfaces of distributed users. This constraint mechanism is one of the declarative techniques which is used in Multi-User Clock.

- *Concurrency Control*: Concurrency control mechanisms are the factors which encourage or discourage the use of a particular groupware system. Thus, they are vital to the acceptance of any (synchronous) groupware system. The concurrency control mechanism as implemented in Multi-User Clock not only resolves possible conflicts between concurrent processes, but also provides fair access to the shared data such that all participants have equal access to it.
- *Customized views:* Multi-User Clock supports the development of both WYSIWIS and WYSINWIS views. More specifically, Multi-User Clock permits both static and dynamic views to be constructed in a WYSIWIS or WYSINWIS fashion. In Multi-User Clock, all views are by default WYSIWIS. That is, the programmer need not add additional code to obtain these type of views. The development of static and dynamic WYSINWIS views are made possible by Multi-User Clock's support of allowing developers to query the session information, and by the flexible placement of data in the application's architecture trees.

## 6.4 Future Work

The work presented in this thesis offers a useful framework for the design and development of toolkits which are aimed to facilitate and ease the construction of groupware and multi-user applications. While the current implementation of Multi-User Clock satisfies a number of requirements, there are other issues left to be resolved. The following list indicates the future plans for enhancing the Multi-User Clock system.

### 6.4.1 Addressing Other Groupware Requirements

Although all of the requirements listed in Chapter 2 are important and contribute to the success of a groupware toolkit, due to time constraints, this thesis provided support only

for a subset of the listed requirements. Future work will result in more of the design requirements being satisfied, and new design requirements being derived.

### 6.4.2 Performance Optimization

The current implementation of Multi-User Clock suffers from relatively poor performance. As a result, it is only practical for development of certain applications. Some experimental research is currently being conducted to locate the source of the latency in Multi-User Clock applications. The optimization methods will follow.

Furthermore, the remote request handling mechanism is currently blocking. That is, a client, after issuing a remote request message, blocks until the result of the request is received from the server. This can be altered to a non-blocking mechanism in which the requests are pipelined during view generations.

Also, the view generations in each client of a Multi-User Clock program are currently performed serially. To improve the performance, the computation of subviews in each client can be done in parallel.

### 6.4.3 Providing a Suite of Distribution Architecture Options

As described in Sections 5.2-5.4, there are various distribution architectures from which the groupware developer can choose. The current implementation of Multi-User Clock is based on a semi-replicated architecture (see section 5.4). Ideally developers should be able to select different architectures for different applications. Furthermore, the developers would be able to conduct experimental research to establish which architecture would better suit the application being developed.

### 6.4.4 Providing a Suite of Concurrency Control Mechanisms

There is no single concurrency control mechanism which works best with all groupware applications. The right choice is based on the nature of the application, type of users, and

the implementation environment. Hence, groupware toolkits must provide an array of concurrency control mechanisms. This will enable the developers to select the mechanism which is most suitable to the application which is being developed. This will also assist the iterative refinement of the groupware applications with respect to performance.

### 6.4.5 Support for Component and Code Reuse

A suite of well-designed, generic components which can be readily incorporated in the developing application, accelerate the development process and possibly lead to development of a better product. Clock already provides a library of generic components, including scrollbar and radio buttons, which can be incorporated into Clock applications. These components, however, do not include groupware specific components like tele-pointers and shared scrollbars. Incorporating a set of groupware specific components, similar to multi-user widgets provided in GroupKit [Rose 92], will assist in the development of a wider range of collaboration aware applications.

## 6.5 Conclusion

The goal of this thesis has been to identify the requirements which facilitate the development of groupware and multi-user user interfaces, and to validate these requirements by addressing them in a toolkit with a special purpose language.

This research classified these requirements into two inter-related groups: requirements for expressiveness, and requirements for ease of use. A number of requirements were identified for each class, and Multi-User Clock was shown to satisfy many of these requirements.

As noted in the introduction of this thesis, the field of computer supported cooperative work and groupware are relatively recent and are still evolving. This thesis by expanding on the requirements, and supporting these via a customized programming language establishes a new framework for future research in this domain.

# Glossary

- ***A**bstraction Mechanisms:* A set of facilities to hide the low-level details from the programmer. These mechanisms provide a system model which more closely represents the programmer's mental model.
- *Asynchronous Groupware:* Groupware systems which support users working at different times (e.g. e-mail).
- ***C**entralized Architecture:* A distribution architecture for distributed applications. This architecture is based on a client-server model, in which a single instance of multi-user application is shared by all users.
- *Client Architecture:* The portion of multi-user Clock applications architecture tree which is replicated to each client.
- *Clock Events:* Clock components communicate with each other through a series of internal events. There are two types of internal events in Clock*: requests* and *updates*. *Input* events, which are generated as a result of end-user actions, are the external events.
- *Clock Language:* The Clock language consists of two parts: a visual, object-oriented language which is used to specify the architecture of Clock programs, and a functional language which is used to specify the components of the architecture.
- *ClockWorks:* An object-oriented visual programming environment. This environment is used by the Clock programmers to construct the application architecture.
- *Collaboration Aware:* A class of interaction techniques in which users' actions are reflected on other participants' work space, in order to provide an awareness of the other users' activities.
- *Concurrency Control*: A mechanism for resolving conflicts between the concurrent processes.

- *Constraints:* A relation between the states, and the views which represent these states. These relations force view recomputation should the view become out-of-date.
- *Computer Supported Cooperative Work (CSCW):* An interdisciplinary research area focused on the role of computer and communication technology to support group work.
- ***D**eclarative Programming:* A programming model which allows high-level specification of applications. Declarative languages allow the programmers to think at a much higher level than the traditional imperative programming languages. This feature enables the programmers to concentrate on ‘what’ they want, rather than, on ‘how’ to make it happen.
- *Direct Manipulation:* The manipulation of objects (e.g. buttons and menus) in a user-interface in order to cause an action in the underlying application.
- ***E**ase of Use:* The simplicity of developing, desired applications using a programming language.
- *Event Handlers:* One of the two primary components of the Clock language. Event handlers are responsible for generation of interface views and handling input events.
- *Expressiveness:* The extent to which a programming language supports the development of diverse applications. This capability allows the developers to construct the desired application.
- ***F**eedback Time:* Or *response* time, is the time necessary for the actions of one user to be reflected by his/her own interface.
- *Feed-through Time:* Or *notification time*, is the time necessary for one user’s actions to be propagated to the remaining users’ interfaces.
- ***G**roupware:* Computer based systems which are explicitly designed to support groups of people working together.
- ***I**nput Thread:* A series of chained update and request events, which are triggered as a result of a single-user input.

- *Iterative Refinement:* The process of iterating between design, user testing, and redesign.
- ***L**ocal Events*: Events which are initiated and serviced at the same site.
- *Private Data:* In multi-user Clock programs, the private data are those which are replicated to each client. Private data make the construction of WYSINWIS, or private views, possible.
- ***R**apid Prototyping:* Rapid prototyping allows the groupware developer to perform usability evaluation by deploying working prototypes at early stages, thus providing a basis for making early critical decisions about the application being developed.
- *Remote Events:* Events which are initiated locally at a site (client or server), and are serviced remotely, in a site other than the one from which they are issued.
- *Replicated Architecture:* A distribution architecture in which each workstation runs its own copy of the application. These copies need to communicate in order to keep their data structures consistent with one another.
- *Request Handlers:* One of the two primary Clock components. These components are abstract data structures which maintain the state of the application as a whole.
- *Role:* "A *role* is a set of privileges or responsibilities attributed to a person or to a system module" [Ellis 89].
- ***S**emantic Feedback:* The system response to a user input. This type of response modifies the system state.
- *Semi-Replicated Architecture:* A distribution architecture for distributed applications, in which the application and the data are split into two parts: server and client. The components of the server part stay at the server, the components of the client part are replicated to each client.
- *Session:* An invocation of a groupware system is informally called a *session.* Formally, session (in groupware terminology) is defined as "a period of time when two or more members of a group are working together synchronously" [Ellis 91].

- *Server Architecture:* The portion of multi-user Clock applications architecture tree which is resident at the server machine.
- *Shared Data:* In multi-user Clock programs, the shared data are those which are kept at the server. Shared data make the construction of WYSIWIS, or shared views, possible.
- *Sub-views:* The parent-child relation between the components of a multi-user Clock architecture tree. This relation corresponds to the containment relation in user-interface views.
- *Synchronous Groupware:* Groupware systems which support users working at the same time (e.g. video-conferencing)
- ***W****YSIWIS (What-You-See-Is-What-I-See):* A kind of interface for multi-user user interfaces, in which all participants share the same views.
- *WYSINWIS: (What-You-See-Is-Not-What-I-See):* A kind of interface for multi-user user interfaces, in which each participant may have a different interface view.

# References


[Ahuja 90] Ahuja, S. R., Ensor, J. R. and Horn, D. N., “The Rapport Multimedia Conferencing System”, *Proceedings of the Conference on Office Information Systems, SIGOIS Bulletin*, Vol. 11, Iss. 2, pages 238-248, ACM Press, 1990.

[Baldesch93] Baldeschwieler, J. E., Gutekunst, T., and Plattner, B., “A Survey of X Protocol Multiplexors”, *Computer Communication Review,* Vol. 23, Iss. 2, pages 16-24, ACM Press, April 1993.

[Baecker 93] Baecker, R. M., ***Readings in Groupware and Computer-Supported Cooperative Work, Assisting Human-Human Collaboration***, Morgan Kaufmann Publishers, 1993.

[Bass 93] Bass, L., “Architectures for Interactive Software Systems: Rationale and Design”, In Bass, L. and Dewan, P. editors, ***User Interface Software***, Chapter 2, John Wiley & Sons., 1993.

[Bernstein 87] Bernstein, P., Goodman, N. and Hadzilacos, V., ***Concurrency Control and Recovery in DataBase Systems***, Addison-Welsley Publishing Company, 1987.

[Bowers 93] Bowers, J. and Rodden, T., “Exploding the Interface: Experiences of a CSCW Network”, *Proceedings of INTERCHI, (Amsterdam, The Netherlands, April 24-29),* pages 255-262, ACM Press, 1993.

[Budde 92] Budde, R., Kautz, K., Kuhlenkamp, K. and Zullinghoven, H., ***Prototyping, An Approach to Evolutionary System Development***, Springer-Verlag, 1992.

[Cortes 94] Cortes, M.,“CSCW Survey: Concepts, Applications and Programming Tools”, Technical Report, Dept. of Computer Science, State University of New York, Stony Brook, 1994.

[Crowley 90] Crowley, T. and Milazzo, P. et al., “MMConf: An Infrastructure for Building Shared Multimedia Applications”, In Haslasz, F. editor, *Proceedings of the Fourth Conference on Computer Supported Cooperative Work (Los Angeles, Ca., Oct 7-10),* pages 329-342, ACM Press, 1990.

[Davie 92] Davie, A. J. T., ***An Introduction to Functional Programming Systems Using Haskell***, Cambridge University Press, 1992.

[Dewan 92a] Dewan, P., “A Guide to Suite”, Technical Report SERC-TR-60-P, Software Engineering Research Centre, Purdue University, February 1992.

[Dewan 92b] Dewan, P. and Chudhary, R., “A High-Level and Flexible Framework for Implementing Multiuser User Interfaces”, *ACM Transactions on Information Systems*, Vol. 10, Iss. 4, pages 345-380, October 1992.

[Dewan 93] Dewan, P., “Tools for Implementing Multiuser User Interfaces”, In Bass L. and Dewan, P. editors, ***User Interface Software***, Chapter 8, John Wiley & Sons, 1993.

[Ellis 89] Ellis, L. and Gibbs, S. J., “Concurrency Control in Groupware Systems”, *Proceedings of the International Conference on The Management of Data (Portland, Or., May 31- June 2),* Vol. 18, Iss. 2, pages 399-407, ACM Press, 1989.

[Ellis 91] Ellis, L. and Gibbs, S. J. and Rein, G. L., “Groupware: Some Issues and Experiences”, *Communications of the ACM*, Vol. 34, Iss. 1, pages 38-58, January 1991.

[Graham 92a] Graham, T. C. N. and Urnes, T., “Relational Views as a Model for Automatic Distributed Implementation of Multi-User Applications”, *Proceedings of the Fourth Conference Computer Supported Cooperative Work (Toronto)*, pages 59-66, ACM Press, 1992.

[Graham 92b] Graham, T. C. N., “Constructing User Interfaces with Functional and Temporal Constraints”, In Myers, B. A. editor, ***Languages for Developing User Interfaces***, Chapter 16, pages 279-302, Jones and Bartlett, 1992.

[Graham 92c] Graham, T. C. N., “Future Research Issues in Languages for Developing User Interfaces”, In Myers, B. A. editor, ***Languages for Developing User Interfaces***, Chapter 22, pages 401-418, Jones and Bartlett, 1992.

[Graham 95] Graham, T. C. N., “Declarative Approaches to User Interface Development”, Ph.D. Thesis dissertation, Technishe Universitat Berlin, 1995.

[Greenberg 92] Greenberg, S. and Roseman, M., “Issues and Experiences Designing and Implementing Two Group Drawing Tools”, *Proceedings of the Twenty-fifth Annual Hawaii International Conference on the System Sciences*, Vol. IV, IEEE Computer Press, pages 139-150, January 1992.

[Greenberg 94] Greenberg, S. and Marwood, D., “Real Time Groupware as Distributed System; Concurrency Control and its Effect on the Interface”, Research Report 94/534/03, Department of Computer Science, University of Calgary, February 1994.

[Grudin 90] Grudin, J., “Groupware and Cooperative Work: Problems and Prospects”, In Laurel, E. editor, ***The Art of Human-Computer Interface Design***, pages 171-185, Addison-Wesley, 1990.

[Grudin 91] Grudin, J., “CSCW: The Convergence of Two Development Paradigms”, *Proceedings of Conference on Human Factors in Computer Systems*, pages 91-97, ACM Press, 1991.

[Hill 90] Hill, R. D. and Patterson, J. F., “Rendezvous: An Architecture for Synchronous Multi-User Applications”, In Haslasz, F. editor, *Proceedings of the Fourth Conference on Computer Supported Cooperative Work (Los Angeles, Ca., Oct 7-10)*, pages 317-328, ACM Press, 1990.

[Hill 92a] Hill, R. D., “Languages for the Construction of Multi-User Multi-Media (MUMMS) Applications”, In Myers, B. A. editor, ***Languages for Developing User Interfaces***, Chapter 9, pages 125-143, Jones and Bartlett, 1992.

[Hill 92b] Hill, R. D., “The Abstraction-Link-View Paradigm: Using Constraint to Connect User Interfaces to Applications”, *Proceedings of Conference on Human Factors in Computer Systems (Monterey, Ca., May 3-7)*, pages 335-342, ACM Press, 1992.

[Hill 93] Hill, R. D., “The Rendezvous Language and Architecture”, *Communications of the ACM*, Vol. 36, Iss. 1, pages 62-67, ACM Press, Jan. 1993.

[Hudak 89] Hudak, P., “Conception, Evaluation, and Application of Functional Programming Languages”, *ACM Computing Surveys*, Vol. 21, Iss. 3, September 1989.

[Ishii 94] Ishii, H, Kobayashi, M. and Arita, K., “Iterative Design of Seamless Collaboration Media”, *Communications of the ACM*, Vol. 37, Iss. 8, pages 83-97, August 1994.

[Kawell 88] Kawell, L., et al., “Replicated Document Management in a Group Communication System”, In Marca D. and Bock, G. editors, *Proceedings of the Second Conference on Computer Supported Cooperative Work*, pages 226-235, ACM Press, 1988.

[Lamport 78] Lamport, L., “Time, Clocks and the Ordering of the Events in a Distributed System”, *Communications of the ACM*, Vol. 21, Iss. 7, pages 558-565, ACM Press, July 1978.

[Lauwers 89] Lauwers, J. C. and Lantz, K. A., “Desktop Teleconferencing and Existing Window Systems: A poor match”, Technical Report, Olivetti Research California.

[Lauwers 90] Lauwers, J. C. et al., “Replicated Architecture for Shared Window Systems: A Critique”, *Proceedings of the Conference on Office Information Systems* (Cambridge, Ma), pages 249-260, ACM Press, April 1990.

[Lotus 93] Lotus Development Inc., “Lotus Notes: An Overview”, October 1993.

[Mandavida 94] Mandavida, M. and Olfman, L., “What Do Groups Need? A Proposed Set of Generic Groupware Requirements”, *ACM Transactions on Computer Human Interaction*s, Vol. 1, No. 3, pages 245-269, September 1994.

[Morton 94] Morton, K., “Tool Support for Component-Based Programming”, Master’s thesis, York University, North York, Canada, June 1994.

[Myers 92] Myers, B. A., “Ideas from Garnet for Future User Interface Programming Languages”, In Myers, B. A. editor, ***Languages for Developing User Interfaces***, Chapter 22, pages 401-418, Jones and Bartlett, 1992.

[Ousterhout 94] Ousterhout, J. K., ***Tcl and the Tk Toolkit***, Addison-Wesley Publishing Company, 1994.

[Peyton 86] Peyton Jones, S. L., editor, ***The Implementation of Functional Programming Languages***, Prentice Hall International, 1986.

[Peyton 89] Peyton Jones, S. L., Clack, C. and Salkild, J., “High-Performance Parallel Graph-Reduction”, ***Proceedings of PARLE***, pages 193-206, 1989.

[Roseman 92] Roseman, M. and Greenberg, S., “GroupKit: A Groupware Toolkit for Building Real-Time Conferencing Applications”, In Turner, J. and Kraut, R. editors, *Proceedings of the Fourth Conference on Computer Supported Cooperative Work*, pages 43-50, ACM press, November 1992.

[Roseman 93a] Roseman, M., “Design of a Real Time Groupware Toolkit”, Master Thesis, Department of Computer Science, University of Calgary, February 1993.

[Roseman 93b] Roseman, M., Yitbarek S. and Greenberg, S., “GroupKit Tutorial”. Included in the public domain GroupKit distribution, available by anonymous ftp from ftp.cpsc.ucalgary.ca under pub/grouplab/software, December 1993.

[Rudebucsh 91] Rudebucsh, T. D., “Supporting Interaction within Distributed Teams”, In Gorling, K. and Sattler, C. editors, *International Workshop on CSCW*, Berlin, Germany, pages 17-33, 1991.

[Rudebucsh 92] Rudebucsh, T. D., “Development and Runtime Support for Collaborative Applications”, In Bullinger, H. J. editor, *Proceedings of the Fourth International Conference on Human Computer Interactions*, Stuttgard, Germany, pages 1128-1132, Amsterdam, 1991.

[Urnes 92] Urnes, T., “A Relational Model for Programming Concurrent and Distributed User Interfaces”, Master’s thesis, Norwegian Institute of Technology, University of Tronheim, Norway (also available as Arbeitspapiere der GMD 643, Germany), 1992.

[Urnes 94] Urnes, T. and Nejabi, R., “Tools for Implementing Groupware: Survey and Evaluation”, Technical Report No. CS-94-03, York University, 1994.