Software Architectures for Multiuser Interactive Systems

W. Greg Phillips
Royal Military College
and
T.C. Nicholas Graham
Queen's University

Multiuser interactive systems allow physically separated users to interact with one another and with shared computational objects in real time. Such systems (also called synchronous groupware) are problematic to build since their user interfaces must support multiple, concurrent users, and both their application logic and their user interfaces must be distributed across multiple platforms. This survey explores the range of software architectures that have been proposed to solve this problem. The presentation includes three distinct architectural views: *reference models*, which divide complete systems into named functional elements and specify data flow between those elements; *architectural styles*, which prescribe component and connector types and their allowed patterns of interaction; and *distribution architectures*, which represent the distribution of system functionality across connected computing platforms. The distribution architectures are presented using a new descriptive framework called Interlace. The architectural presentation is complemented by a brief overview of consistency maintenance mechanisms for groupware.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*; H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces—*Collaborative computing, Computer-supported cooperative work, Synchronous interaction*

General Terms: Design, Human Factors

Additional Key Words and Phrases: Multiuser interactive systems, groupware, reference models,

This work was partially supported by the Natural Science and Engineering Research Council (NSERC) and Communications and Information Technology Ontario (CITO).
Name: W. Greg Phillips
Affiliation: Royal Military College of Canada
Address: Department of Electrical and Computer Engineering, Royal Military College of Canada, Kingston, Ontario, Canada, K7K 7B4. greg.phillips@acm.org

Name: T.C. Nicholas Graham
Affiliation: Queen's University
Address: Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, K7L 3N6. graham@cs.queensu.ca

architectural styles, distribution architectures, Interlace

1. INTRODUCTION

The emergence of high-bandwidth networks and inexpensive home computers has lead to the widespread use of computers for communications and collaborative tasks. This has required the study of how to engineer interactive software that supports multiple users. Such software is often called *groupware* [Baecker 1993], or *synchronous* groupware in the special case of software supporting *real time* collaboration.

There are many kinds of interactive multiuser applications, each with different development and ergonomic requirements. Some support pure communications tasks, for example, Internet telephones [Kolon and Goralski 1999] and chat programs [Viegas and Donath 1999] allow people to converse using voice or text, while media spaces [Coutaz et al. 1998] contribute to mutual awareness in work groups by allowing people to see views of others' offices.

Other applications support the collaborative production of work artifacts, such as school assignments [Mitchell et al. 1995], bank loan applications [Kobayashi et al. 1998], software designs [Grundy 1998] or geophysical simulations [Duce et al. 1998]. These applications combine support for real-time discussion and manipulation of shared artifacts.

Ideally, groupware applications should allow people to collaborate at least as effectively as if they were in the same room. Addressing this ergonomic challenge, experimental approaches have examined how collaboration can be aided through communication of gaze awareness [Vertegaal 1999] and facial expressions [Scheirer et al. 1999].

The development of groupware represents an interesting challenge to the software engineering community, due to the ergonimic challenges of replacing face-to-face communication combined with the engineering challenges of building robust, responsive distributed applications. Since communication and collaboration are so fundamental to human activity, we believe that the need to meet these challenges will have a significant effect on how software is developed. Developing synchronous groupware involves most of the challenges of traditional applications, as well as new challenges, These challenges fall into the areas of:

Distributed programming: Groupware applications are implemented as distributed systems, involving traditional problems of network programming, code and data distribution, consistency maintenance of replicated data, security, and fault tolerance.

Real-time performance: In order to effectively support real-time communication, groupware applications need to meet a set of performance bounds. These include feedback time (the time it takes the application to respond to a user's own input), feedthrough time (the time it takes the application to reveal other users' actions), and jitter (the variance of feedback and feedthrough times.) Acceptable values for these performance criteria are application-dependent. As an example, Shneiderman suggests that in highly interactive applications, feedback time should lie in the range of 50-150 ms [Shneiderman 1998].

User interface design: The success of a groupware application depends on the quality of its user interface. Applications must permit users to smoothly blend production and communication tasks, while avoiding conflicts in their activities. The user interface must reveal to an application user not only the state of his or her own interaction, but also the interactions of other users. The user interface must seamlessly blend communication modalities such as speech, video, facial expression or gaze.

The body of knowledge of how to build such applications is increasing, particularly as the importance of communication in user interfaces increases. In this paper, we survey one aspect of software engineering knowledge for synchronous groupware systems—the software architecture. As we shall see, the presence of a catalogue of architectural approaches aids developers in creating groupware applications. Our systematic survey of these architecture styles helps to clarify the state of the art of development methods for groupware, showing areas where considerable further work is required.

1.1 Architecture

Software architecture contributes throughout the development lifecycle of software systems [Bass et al. 1998; Shaw and Garlan 1996]. Architectures for synchronous groupware particularly help in the challenges of programming distributed systems with exigent performance and usability requirements. Software architectures help to:

Document classes of design: Deciding how to structure software requires creativity, experience and experimentation. Once a way of structuring software has been found to work well in one project, it is useful to record this structuring approach so that it can be used in similar projects. Such structuring decisions can be recorded as an *architectural style* that can be applied in new situations. Ultimately, software designers should have a catalogue of architectural styles available to them. This catalogue should clearly present the tradeoffs of architectural decisions, allowing designers to choose the architecture most suitable to their application. Applications may be based on a single architecture style, or on a combination of styles. Architecture styles can record mechanisms for efficiently sharing data, for aiding evolution of the architecture and for making the system more tolerant of faults in the underlying distributed system infrastructure.

Record design: Architecture diagrams record the high-level design of an application. Maintenance programmers can make use of this information in understanding a system as it evolves. In novel application domains such as groupware where experimental implementation techniques are being used, maintenance programmers may not be able to easily recover the underlying design from the code. It is therefore beneficial to clearly describe the design.

Factor implementation: Software architectures factor systems into components with well-defined interfaces. This simplifies the implementation and maintenance of systems by breaking the system into work packages suitable for team development, and reducing the complexity of each work package. If the underlying design of the system has been chosen from a catalogue of architectural styles, the factoring properties are well understood before implementation begins, reducing uncertainty in the development process.

Allow early analysis: By exposing system structure early, software architectures can expose system properties before their implementation. In the case of groupware, such properties might deal with, for example, system performance, fault tolerance, cost of construction and amenability to evolution.

Support tools: If a system designer is willing to adhere to a specific architecture style, tools may help in the implementation of the architecture, perhaps abstracting low-level details of the implementation of the architecture style. Tools for the implementation of groupware typically help by abstracting some aspects of the implementation of view consistency, model consistency, code and data replication and network communication. In order to implement these issues automatically, tools require the developer to adhere to some architecture style.

Software architecture is therefore helpful throughout the life cycle of program design, development and maintenance, in reducing cost of design and in better predicting the behaviour of a design throughout the life cycle.

1.2 Structure of Survey

This survey provides an overview of the state of the art in software architecture for synchronous groupware applications. It is organized around a common vocabulary for describing architectural views of groupware (described in appendix B), exposing similarities, deifferences and trade-offs between different approaches. By methodically organizing the work done in software architecture for groupware, the survey shows not only what has been accomplished in the field, but also what problems remain open for further work.

The survey is therefore of benefit to groupware implementers in that it provides a catalogue of different architecture styles and a description of available tools that support those styles. This provides valuable information to developers, helping them choose an appropriate architecture style for implementation.

The survey is further of benefit to researchers in implementation techniques for groupware and groupware toolkits, by exposing not only the work that has been done, but what work remains.

Finally, the survey is of interest to researchers in software architecture in general, by cataloguing examples of architecture styles developed for the emerging domain of computer supported cooperative work.

The survey is structured around the themes of architectural *description*, *design* and *implementation*:

Describing architectures: The first topic in the survey is *reference architectures*. Rather than describing the architecture of a particular system, a reference architecture describes a set of possible architectures. Reference architectures are sufficiently general that they can be used to *describe* the architectures of existing systems. Reference architectures therefore serve to illustrate the space of possible architecture designs for a groupware system, where each articulation point of the reference architecture represents a design choice leading to a concrete system architecture.

Architectures for designing systems: We then move to *architectural styles*, which capture specific techniques for structuring groupware systems. Architecture styles support the implementation design of groupware by capturing design strategies for use in new systems. Designers typically design their systems using one or more architectural styles, then develop implementations that are consistent with those styles. Conceptual architectural styles usually aid the developer by abstracting low-level details of the distributed implementation of groupware, such as strategies for network communication, data replication and consistency maintenance. Groupware development toolkits typically support development using a single architectural style, requiring developers to use that style, and in return providing an implementation that solves these low-level implementation issues.

Architectures for implementing systems: Finally, we present an architectural view that captures the *distributed implementation* of groupware. This view describes mechanisms for realizing groupware applications as sets of communicating processes distributed over a network of computers. Developers typically use a conceptual architecture style to create the high-level architecture of their system, which can then be mapped to a distributed architecture either by hand or using a tool. Since none of the reference architectures descussed in section 2 are adequate to fully describe distribution architectures for groupware, we introduce the *Interlace* reference model as a descriptive framework.

Throughout this presentation, toolkits supporting these architecture styles will be presented. Toolkits usually support development using a particular architecture style; for example, the *RendezVous* toolkit supports the *Abstraction-Link-View* architecture style [Hill et al. 1994], while the *Clock* toolkit supports the Clock architecture style [Graham and Urnes 1996]. Toolkits are important to understanding groupware architectures, as they provide a concrete realization of an architecture style. Often, the only definition of the architecture style is that provided implicitly by the tool's input domain. The success of a tool provides some measure of the success of a particular architecture style. As this is a survey of groupware architectures, however, we restrict ourselves to considering only the architectural issues of these toolkits. We begin with a presentation of reference models for groupware.

2. REFERENCE MODELS

Reference models for groupware are intended to describe the structure of complete groupware systems at a relatively abstract level. Reference models can be viewed as either prescriptive or descriptive. In the prescriptive view, they express a philosophical perspective as to how groupware systems should be constructed. In the descriptive view, they provide a canonical framework within which we can reason about the structures of existing groupware systems.

In this section we present four reference models which are relevant to groupware: Seeheim [Pfaff et al. 1985], Arch [UIMS Tool Developers' Workshop 1992], Patterson's [Patterson 1995], and Dewan's [Dewan 1995]. Seeheim and Arch are the products of workshop committee efforts and are actually models for single user systems. They can be extended in a relatively straightforward way to describe groupware systems. Patterson's taxonomy was the first real attempt to define architectures for groupware systems and focuses on a key issue for groupware: how to maintain the shared objects with which the users interact. Dewan's "generic architecture" extends the Arch approach from single user systems to groupware and generalizes it to include several possible patterns of communication among the replicated instances of an application.

2.1 Seeheim and Arch

Reference models for groupware originate in similar models proposed for single user interactive systems. The key idea is that an appropriate decomposition of an interactive system includes a clean separation between the application's underlying logic or *functional core* and its user interface. This separation allows development of core and interface to be largely decoupled and admits the possibility of different styles of user interface for a single functional core. For example, a list of numbers could be viewed and manipulated as a column of text by one interface, as a bar chart by another, and as a pie chart by a third. A reasonable extension to this separation allows multiple, concurrent user interfaces to be attached to a single functional core; this provides a simple model for a groupware system.

The first widely-accepted reference model for single-user interactive systems was developed in 1985 at a working group in Seeheim, Germany, and later dubbed "the Seeheim model" [Pfaff et al. 1985]. It is illustrated in figure 1(a). The functional core is allocated to the *application* component and the user interface is divided into a *presentation* component, a *dialogue controller* and an *application interface*. Roughly, the presentation component handles lexical aspects of the interaction, the dialogue controller handles syntactic aspects, and the application handles semantic aspects. The application interface is an *adaptor*, mapping between application-level concepts and dialogue concepts to reduce the coupling between functional core and user interface.

The small box shown to the right of the dialogue controller is a "fast switch", which recognizes the requirement for rapid response on the part of the interface. It allows the application to bypass the dialogue controller when dialogue state is not affected by output events.

During the period 1991–1992 the Seeheim model was refined and extended into the Arch model [UIMS Tool Developers' Workshop 1992], shown in figure 1(b). Arch removes the fast switch, introduces a new adapter component in the user interface, and makes explicit the types of data flowing between the components. In Seeheim, the dialogue controller is primarily concerned with processing input syntax; in Arch, this responsibility is moved forward to the logical interaction component and the dialogue component becomes responsible for screening inputs and sequencing the tasks performed by the functional core.

The Arch model also has an associated "Slinky" meta-model which recognizes that tasks can migrate between components in the architecture depending on the developers' needs, the relative importance of different development goals, and the nature of the system. The metamodel implies that the relative size and importance of the model's components may vary from system to system. In some systems, one

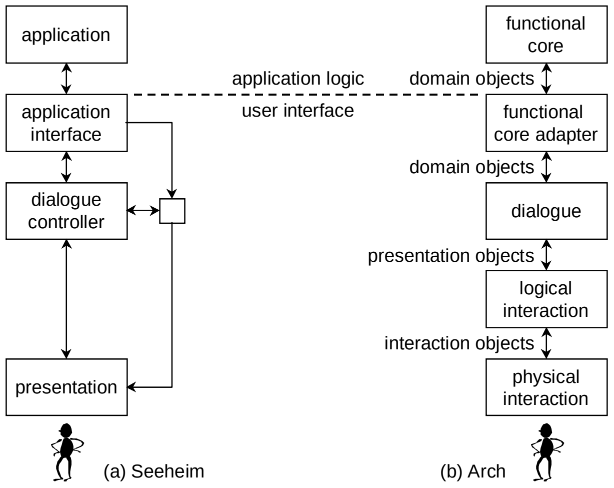


Fig. 1. The Seeheim and Arch reference models for single-user interactive systems. The Arch model is often drawn with its five components in the shape of an arch, the dialogue component forming the "keystone".

or more components may be compressed out of existence, while others may expand to cover multiple functions implied in the original architecture. The mental image behind the name is the Slinky child's toy, which can be compressed and expanded when passed from hand to hand.

The Seeheim and Arch models are motivated by the conviction that a system divided into layers is easier to develop, maintain and extend. By defining the layers such that they map easily onto existing systems (*e.g.*, the physical interaction component of Arch maps directly onto the X-Window system), the architectures' proponents hoped to support ease of implementation using pre-existing software components. In section 3 we present a number of architectural styles which can be seen as specializations of the Seeheim/Arch reference models.

2.2 Patterson's Taxonomy

The first reference model developed specifically for groupware systems appears in a taxonomy proposed by Patterson [Patterson 1995]. It is motivated by the observation that "the primary challenge for synchronous groupware applications is to maintain shared state." Patterson divides application state into four levels: *display* state, which is implemented in the video hardware that drives the user's physical display; *view* state, a logical visual representation of the underlying data; *model* state, the underlying data itself; and *file* state, the persistent representation of the model. The taxonomy intentionally leaves all computational aspects of the application unspecified.

Patterson's taxonomy is illustrated in figure 2.[1] It proposes three classes of architectures for groupware: those based on actual shared state, those based on

[1]Where diagrams require multiple users we normally show only two; the reader should imagine an arbitrary number of users with similar local architectures and communication paths.

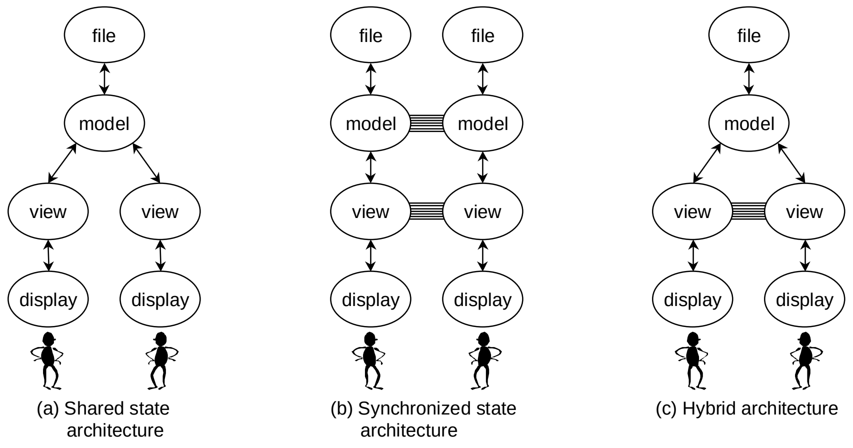


Fig. 2. Examples from Patterson's taxonomy.

replicated state with synchronization, and hybrids including both approaches. The key benefit of actual shared state is simplicity; the benefits of replicated state are potentially improved performance and the ability to turn synchronization on and off.

In shared state architectures, all levels above the first shared one are assumed to be shared as well. This leads to architecture diagrams which look as if they have been "unzipped" from the bottom; Patterson's reference model is occasionally referred to as the "Zipper" model for that reason. In figure 2(a) both the model state and file state are shared but views and displays are separate, providing users with independently constructed views of common application data.

Figure 2(b) shows a shared view system implemented using a synchronization architecture. The synchronized elements are intended to contain exactly the same information and are replicated for performance or other pragmatic reasons. In the example, users would see exactly the same views on their displays, a situation sometimes referred to as strict What-You-See-Is-What-I-See or WYSIWIS [Stefik et al. 1987]. This contrasts with figure 2(a) in which the users could have arbitrarily different views of the same underlying state. Patterson suggests that where views are synchronized it is also necessary to separately synchronize models.

The example in figure 2(c) is a hybrid architecture in which the model is shared and the views are synchronized. This architecture provides a simple mechanism for ensuring consistency of the model state and allows view sharing to be switched on and off as desired by the users.

Patterson's taxonomy provides a simple representation of possible distribution architectures for groupware systems but abstracts away issues of computation, concurrency and distribution. We revisit these issues in section 4. We suggest that only sharing at the view and model levels is of direct interest for synchronous groupware: interaction through shared files is difficult to achieve in real time and, as Patterson notes, synchronization of physical display information (that is, information normally found in the video hardware itself) is not generally feasible and is not necessary if views are synchronized or shared.

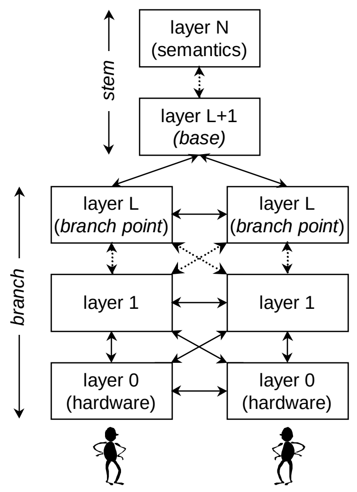


Fig. 3. Dewan's generic architecture. Dotted lines represent ellipsis. All communication is via events; however, the event indicator (see appendix B) has been omitted from the arrows for clarity.

2.3 Dewan's Generic Architecture

Dewan's reference model [Dewan 1995], illustrated in figure 3, can be seen as a combination and generalization of the Seeheim/Arch models and Patterson's taxonomy. As in Seeheim/Arch, a system is modeled as layered components increasing in abstraction as one moves up the diagram (away from the user). As in the Patterson's taxonomy, the structure may be "unzipped" up to some level and direct communication between the replicated layers is permitted. Dewan proposes no specific number of layers.

Objects in layers closer to the user are called *interactors* of *abstractions* one layer higher. An object in the middle of the hierarchy will be both an interactor and an abstraction. An interactor creates a *presentation* of its abstraction, which may include a transformation of the abstraction (*e.g.*, a text field representing a number, a bitmap representing a text field) as well as additional information which may be viewed as "syntactic sugar" (*e.g.*, scrollbars and menus).

Communication among objects in the model is via *events*, which may be synchronous or asynchronous. Events reflecting a single user's interaction with the system are called *interaction events* and flow strictly up and down the tree. Events reflecting collaboration among users are called *collaboration events* and may flow up and down the tree, if destined for a layer in the shared *stem*, or across, if destined for *peer* replicated layers. An event traveling up (down) the tree can also be "fanned out" to all layers one level above (below), effectively replicating input (output) events across the branches of the system. Dewan suggests and motivates a number of event classes supporting activities such as locking, commands, undo, merge, and join/leave.

Components including code that directly supports collaboration are called *collaboration aware*; those without such code are *collaboration transparent*. The model allows either type of component to appear at any level in the hierarchy.

2.4 Discussion

Seeheim, Arch, and Dewan's generic architecture all model systems as a series of layers going from concrete at the physical interface to abstract at the functional core. In Arch and Seeheim there are three main layers concerned with physical interaction, the control of dialogue, and semantic aspects of the application's data structures. The adapter layers simplify the main layers and permit them to be constructed in relative isolation. In Dewan's model an arbitrary number of layers is permitted.

These models support the well-accepted software engineering principles of modularity and separation of concerns. Allocating the functions of a given layer to a pre-existing component promotes reuse. Judicious use of the models is expected to reduce total application complexity and lifecycle cost; while there are no empirical studies demonstrating the truth of this conjecture for these models in particular, there is some evidence that it does hold for layered architectures in general [Zweben et al. 1995].

Some groupware development efforts have been guided by the principles enshrined in these models [Coutaz 1997; Singh and Green 1991]. Arch has been used as the basis for a study of single-user interactive systems [Kazman et al. 1993] and Dewan has characterized the architectures of a number of groupware systems using his generic model [Dewan 1999].

Patterson's taxonomy usefully clarifies the patterns of state sharing and replication found in groupware systems but ignores distributed computation aspects. In section 4.1 we introduce a new reference model to permit further exploration of these issues.

All of these reference models propose structures or ways of looking at entire applications in a relatively coarse-grained fashion. In the next section we look at architectural styles, which provide patterns for application development at a much finer level of detail.

3. ARCHITECTURAL STYLES

An architectural style suggests a vocabulary of component and connector types, a topology of interconnection, and a control flow strategy. Ideally, an architectural style will provide the developer with a clear mental model for the system under development, will provoke appropriate questions at an early stage in the development process, and will provide direct operational answers to those questions [Calvary et al. 1997]. For groupware the key question centers around how to build systems that allow multiple users to concurrently interact with each other and with shared data.

In this section we present a selection of architectural styles for groupware which strive to answer this question. There have been a wide range of such styles proposed, practically one for every groupware application, system, toolkit or programming language. Here we include those styles which are influential and widely known as well as a limited number of more recently proposed styles.

Most architectural styles for groupware are based on the same separation of user interface and application seen in the Seeheim model (section 2.1). However, where Seeheim proposes dividing the entire system into a single application and a single user interface, these styles divide the system into fine-grained subsystems, each with its own application and user interface component. Systems sharing this feature include the Presentation-Abstraction-Control (PAC) [Coutaz 1987a], Model-View-Controller (MVC) [Krasner and Pope 1988], Abstraction-Link-View (ALV) [Hill et al. 1994], Suite [Dewan and Choudhary 1992], and Clock [Graham and Urnes 1996] architectural styles. These styles also share a rather surprising feature: they do not directly express inter-user interaction. Instead, interaction is allowed to arise implicitly from some form of state sharing. In a sense this is an effort to reduce the complexities of groupware development to that of single user systems by abstracting away issues relating to component-level distribution, communication, and consistency maintenance. We revisit these issues in section 4.

There are also architectural styles which do directly express user interaction, typically by including other architectural features along with a Seeheim-like separation between user interface and application. Included in this group are architectural styles based on explicit remote procedure calls (GroupKit [Roseman and Greenberg 1996]), inter-client communication channels (the Java Shared Data Toolkit [Burridge 1999]), and inter-component buses (Chiron-2 [Taylor et al. 1995]).

We begin the presentation with PAC*, which can be viewed as inhabiting an intermediate space between the reference models of the previous section and the other styles in this section. PAC* suggests a complete structure for an application, based on the Arch reference model, and is also relatively abstract, whereas the styles presented after it are quite concrete and most are directly supported by programming environments or toolkits.

3.1 PAC*

The PAC* (Presentation-Abstraction-Control-*) architectural style [Calvary et al. 1997; Coutaz 1997] is the most recent member of a family of styles stemming from PAC itself, which was first proposed by Coutaz in 1987 [Coutaz 1987b]. Later, PAC was used as the basis for PAC-Amodeus [Nigay and Coutaz 1991], an application architecture based on the Arch reference model in which the Dialogue component is implemented in the PAC style. PAC* is an extension of PAC-Amodeus to groupware, based on "unzipping" the Arch in the style suggested by Dewan's generic architecture. In this exposition we present first PAC, then PAC-Amodeus, and finally PAC*. PAC itself is described in more detail in [Buschmann et al. 1996].

The PAC architectural style decomposes a system into a hierarchy of PAC *agents*. Each agent includes three *facets*: a *presentation*, which represents the user interface, an *abstraction*, which maintains the underlying data, and a *control*, which mediates all communication between the presentation and abstraction. Each agent is viewed as autonomous, and may execute in an independent process or thread. A PAC agent is illustrated in figure 4(a).

The PAC control facet is similar in spirit to the dialogue component in the Arch model. It provides a single location for dialogue-dependent code (*i.e.*, code which controls the dialogue between the presentation and the abstraction), thus simplifying creation and maintenance of dialogue policy. It also decouples the development of presentation and abstraction objects, since most syntactic or semantic mismatches between them can be accommodated by the control.

The structure of a typical PAC application is shown in figure 4(b). The hierarchy

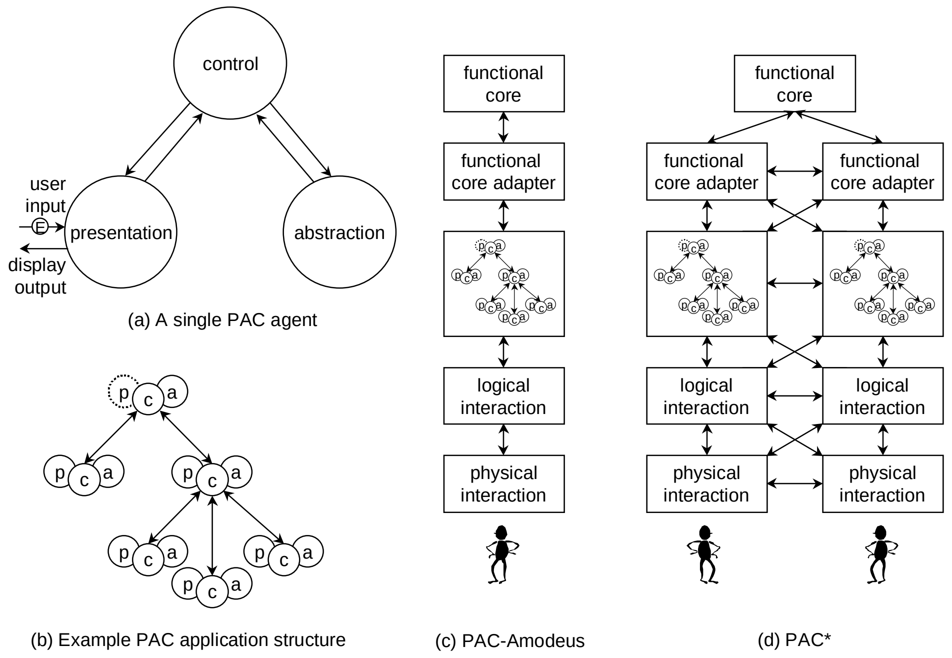


Fig. 4. The PAC family of architectural styles.

of PAC agents represents application composition. The root agent represents the application as a whole, and is decomposed (here) into two subcomponents which might represent different visible windows of the application. These are in turn further decomposed until the level of individual interactive objects is reached. Note that in figure 4(b) the presentation element of the root agent is shown with a dotted line; since the application as a whole has no visual representation, its presentation is effectively null. It is also possible for components to have null abstractions.

PAC controls not only adapt the interfaces of their presentations and abstractions, but also initiate and route communications up and down the hierarchy as required. This keeps the presentations and abstractions simple and localizes all communications-related code in the control facets. The control facet can be seen as implementing the Adaptor and Mediator design patterns described by Gamma *et al.* [Gamma et al. 1995].

PAC is intended to be used at a conceptual level and is not tied to any one programming language or implementation technique. Communication within a single agent and between agents can take any appropriate form. One reasonable approach would be for the control to directly call the presentation and abstraction, but for the presentation and abstraction to communicate with the control via callbacks. This allows presentations and abstractions to be written without regard to their particular employment, making them more reusable. Communication up and down the PAC hierarchy might take place via events. Other strategies are possible.

The original formulation of PAC assumed that an application would be homogeneously structured using only objects interacting in the PAC style. However, in the development of real applications it is often advantageous to reuse existing large-grain components, such as windowing systems or relational databases, which are outside the PAC architectural style.

The PAC developers observed that these large-grained components would typically be seen as mapping onto the interaction or functional core components of the Arch model. However, the Arch dialogue component is normally unique for each application. Since PAC has specific support for dialogue implementation, in the form of its control facet, it seems well suited to this task [Coutaz 1997]. This observation motivated development of the PAC-Amodeus architectural style, in which the dialogue component of the Arch model is implemented using PAC [Nigay and Coutaz 1991].

PAC-Amodeus is illustrated in figure 4(c). Typically, PAC presentations within the dialogue component will be connected to some element of the logical interaction component. Similarly, PAC abstractions will be connected to elements of the functional core adapter, which will perform any necessary translation between PAC and the functional core itself. Nigay has proposed a set of structuring rules to guide designers in creating appropriate PAC hierarchies for dialogue component implementation [Nigay and Coutaz 1991].

The PAC* architectural style for groupware extends PAC-Amodeus by "unzipping" the Arch structure in the style suggested by Dewan's generic architecture, but also allowing multiple forks and joins. One possible example of a PAC* architecture is illustrated in figure 4(d). In [Calvary et al. 1997], Nigay's structuring rules are extended specifically to deal with groupware development issues. In addition, they suggest that multi-user applications must address the three dimensions of *production* (the creation of shared artifacts), direct *communication*, and *coordination* of users' activities across time [Salber 1995], and explain how PAC* can be used to address these issues. The PAC* architectural style has been used successfully in the development of complex multi-user applications with multi-modal input [Coutaz 1997].

As indicated above, the PAC* architectural style is at a relatively abstract level and might reasonably be seen as occupying a middle ground between reference models and architectural styles. In the balance of this section we present a series of architectural styles which are more concrete in nature, many of which have either direct programming language or toolkit support.

3.2 Model-View-Controller

The Model-View-Controller (MVC) architectural style was introduced in Smalltalk-76 [Krasner and Pope 1988] to provide a clean, principled separation between user interfaces and their underlying application semantics. In this section we present MVC as defined in Smalltalk-80. The MVC style has been implemented with minor variations in a wide range of user interface toolkits, most recently in Sun's Swing framework for Java [Eckstein et al. 1998]. MVC is described as a design pattern by Buschmann *et al.* [Buschmann et al. 1996] and is the archetype of the Observer pattern described by Gamma *et al.* [Gamma et al. 1995].

The structure of (single-user) MVC is illustrated in figure 5(a). The basic MVC structure consists of a *model*, which represents the application's data; a *controller*, which interprets user input; and a *view*, which presents output. The controller and

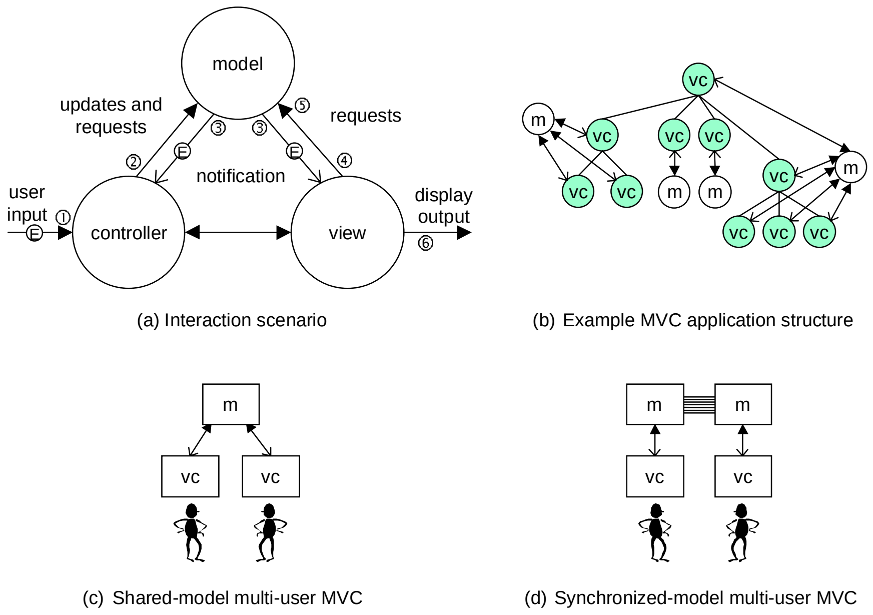


Fig. 5. Single user MVC and extensions for groupware. In (b),(c) and (d) the views and controllers are combined, the call and callback arrows are superimposed, and the event indicators have been omitted. In (b) we have shaded the view-controller pairs to highlight the tree structure.

the view are together similar to the PAC presentation object and the model to the PAC abstraction; however, MVC has no direct analogue of the PAC control. Instead the controller and view can communicate with the model via calls and the model sends the view and controller events via anonymous callbacks.

The view and controller communicate with each other directly by calls. For example, a pop-up menu would normally be implemented within a controller object; the controller would communicate with the view to paint the menu on screen as required. Since the coupling between views and controllers can be quite close, many MVC variants (including Swing) implement view-controller pairs as combined objects.

A typical MVC interaction sequence begins with user input (event 1, at figure 5(a)) which is interpreted by the controller and results in an update to the model (2). The model broadcasts a notification to the view and controller (3) that some aspect of its state has changed. The view queries the model to determine exactly what the change is (4) and on receipt of the details (5, the value returned from the query) updates the display (6). The controller may also react to the notification by changing its mode of interaction.

Each view and controller requires explicit knowledge of its model's calling structure in order to perform updates and correctly interpret change notifications. Conversely, the model requires no information about its views and controllers. Instead, a model provides a mechanism for views and controllers to register anonymous callbacks with it, and uses those callbacks as destinations for event messages. This allows arbitrary views and controllers to attach to a given model without alteration of the model itself.

As shown in figure 5(b) an application in the MVC style may have many models and associated view-controller pairs. The view-controller pairs are typically organized in a subview/superview tree representing visual containment. This allows simple support for the clipping operations and coordinate transformations common in graphical applications. Each view-controller pair could have a separate model; however, in practice a single model is often associated with a subtree of the view hierarchy. There is no default pattern of communication between the models in an application; in fact, the models might not communicate with one another at all.

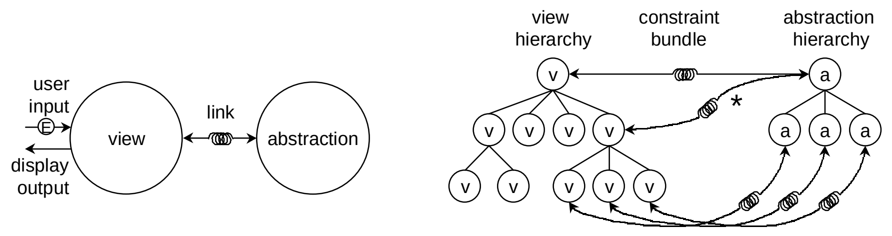
It is possible to develop groupware under MVC by attaching multiple view-controller pairs (one for each user) to each model that needs to be shared. This is illustrated in figures 5(c) and (d) using true sharing and model synchronization as suggested by Patterson's Zipper model. In these figures the model and view-controller components represent the collection of model objects and the view-controller hierarchies, respectively. Despite the simplicity of this approach, MVC (in its original form) is not a particularly satisfactory basis for groupware as it does not support concurrently active controllers [Krasner and Pope 1988]. Some recent systems loosely based on MVC, including Clock (see section 3.4), and the transaction-based DECAF toolkit [Strom et al. 1998] overcome this difficulty.

3.3 Abstraction-Link-View

The Abstraction-Link-View architectural style (ALV, pronounced "al-vee") was developed at Bellcore as the architecture of the Rendezvous system [Hill 1992; Hill et al. 1994], a toolkit designed for the construction of groupware supporting "conversational props" or shared objects. Its overall structure is similar to shared-model MVC, with combined view-controller pairs (called *views*), separate hierarchies of views and *abstractions* (models), and a declarative constraint mechanism called the *link* which connects the two. Abstraction and view components are trees of objects; links are "bundles" of constraints. The ALV architectural style is illustrated in figure 6.

As shown in figure 6(a), ALV treats user inputs as events, which are routed to event handling routines defined in the view. Unlike MVC, the event handlers do not act directly on the abstraction; instead, they modify data stored locally in the view. Keeping local view data reduces the overhead of performing view recomputation, but introduces a requirement that the view data be kept consistent with any corresponding data in the abstraction. Meeting this requirement is the role of the constraints. ALV constraints are one-way, from a source to a target; however, two constraints with inverted sources and targets together constitute a bidirectional constraint. Thus, changes in the view can automatically be reflected in the abstraction and vice versa.

The structure of an ALV application is illustrated in figure 6(b). A user's view component consists of a tree of view objects, which are connected by constraints to corresponding objects in the abstraction. Normally each object in the abstraction will be represented by one object in the view; however, some view objects (*e.g.*, buttons) may not have corresponding abstractions. Since each user has an independent view hierarchy and independent constraints, users can have very different graphical displays of the same underlying abstraction. Any change in the abstraction (typ-



(a) Basic ALV structure

(b) Example ALV application structure; there is one view hierarchy (plus constraints) for each user.

Fig. 6. The ALV architectural style.

ically arising from an action in one user's view) will be automatically reflected in all users' views through the constraint mechanism.

In interactive applications, it is quite common for view and abstraction objects to be created and destroyed during execution of a program. In ALV, the creation of an object in one hierarchy can cause a corresponding object to be created in the other by means of a "tree maintenance" constraint. For example, in figure 6(b), the object at the top of the abstraction hierarchy might dynamically instantiate its three children at run time. A tree maintenance constraint (part of the constraint bundle marked by a "*") could then cause the creation of corresponding objects in the view, and the connection of those view objects to the abstraction hierarchy using appropriate constraints.

The ALV constraint system allows independence of the views and models beyond that provided by MVC. An MVC view requires intimate knowledge of its model's calling structure and data representation; however, an ALV view can (in principle) be programmed independent of any particular abstraction. Since constraints are programmatic objects, they can include code to provide any syntactic or semantic conversion required for the views and abstractions to work together. ALV's separation of interaction issues from representation and presentation issues allows each of the three components to be simpler than it might otherwise be, supporting greater reusability at the object level. [Hill et al. 1994].

3.4 Clock

The Clock architectural style [Graham and Urnes 1996] provides a component-level framework supporting software development in the Clock declarative programming language [Graham 1995]. The architectural style is directly supported by a visual programming environment, ClockWorks [Graham et al. 1996].

As illustrated in figure 7(a), Clock "in the small" is superficially similar to MVC. The Clock *model* is composed of one or more abstract data types (ADTs) whose interfaces include both *requests* (referentially transparent accessors) and *updates* (mutators). The *controller* consists of declarative functions, triggered by user inputs, which may make make requests of, or send updates to, the model. The *view* consists of declarative functions, automatically triggered by changes in model request values, which compute the program's output.

Clock derives some of its ideas from Weasel [Graham and Urnes 1992], an earlier system developed by the same researchers. The principal difference is that Weasel

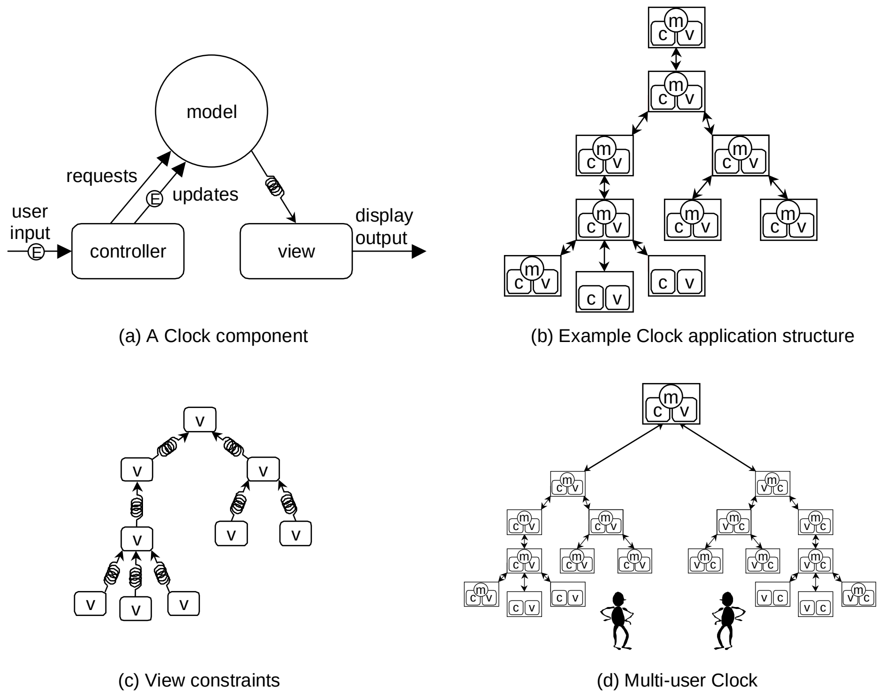


Fig. 7. The Clock architectural style. In (b) and (d) the requests, updates and constraints have been collapsed into a single arrow to reduce clutter.

had no controller; rather the Weasel view was a direct projection (in the functional programming sense) of the underlying model. A user's manipulation of the view therefore resulted in direct and immediate update of the model, and vice versa. While this approach is appealing in its simplicity, it makes modal styles of user interface particularly difficult to construct; Clock introduced its separate controller element to provide the required flexibility.

A Clock application is composed of Clock components, hierarchically arranged as illustrated in figure 7(b). Each component may be missing any one of the model, view or controller elements, but a component typically includes at least a view and a controller. As in MVC and ALV, the hierarchy represents visual view containment. Since containing views are (partially) expressed as functions of their contained subviews, views are constrained by their subviews as indicated in figure 7(c). Conceptually this means that a view will be automatically recomputed if any of its subviews changes. In practice Clock includes optimizations to reduce unnecessary view recomputation.

The component hierarchy extends the communications pattern shown in figure 7(a) from the component level to the application level by allowing both controllers and models to respond to requests or updates originating from lower-level controllers. A request or update is actioned by the local model, or failing this, by the next component upward in the hierarchy which provides the request or update in its interface. Within a component, the controller is considered to be below

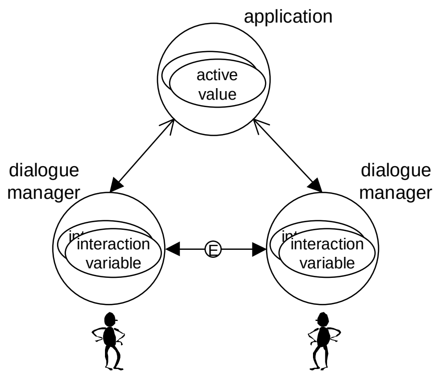


Fig. 8. The Suite architectural style.

the model. Having controllers respond to requests and updates provides for their transformation on the way up the hierarchy and allows resolution of any syntactic or semantic mismatches between components. Since model constraints on views are syntactically identical to requests, they are extended to the hierarchy in an identical fashion. This provides a simple and natural mechanism for data sharing: if a particular ADT is shared by several components, it is normally located in the lowest component in the hierarchy that is visible to them all. Data required by all components in an application is located at the root. The ClockWorks editor provides a simple mechanism for moving ADTs from one component to another.

Groupware is programmed in Clock by providing a view function at some level which allows one subview (sub-tree) to be created for each user. This has the effect of "unzipping" the hierarchy, as shown in figure 7(d), which is a multi-user version of figure 7(b). State represented in models above the unzipped sub-tree is common to all users' views; state in the sub-tree is local to each user. This allows view replication policies to be changed from completely independent views to full WYSIWIS and back, simply by moving ADTs up and down the hierarchy. Clock's formally defined semantics [Graham 1995] guarantee that requests and updates always execute atomically, which eliminates a large class of concurrency control problems.

3.5 Suite

The Suite system was originally developed to experiment with the automatic generation of single user interfaces [Dewan and Choudhary 1991] and was later extended to support groupware [Dewan and Choudhary 1992]. It models groupware as a collection of generalized data-structure editors acting on shared data [Dewan 1993].

The Suite architectural style, shown in figure 8, is superficially similar to shared-model MVC; however, the details are quite different. A Suite program consists of a shared *application* which maintains and manipulates semantic state, and replicated *dialogue managers* which provide the individual user interfaces (*editors*) for the application's data structures. Applications and dialogue managers are coarse-grained, heavy-weight objects. Applications are automatically persistent; dialogue managers are created on request and "attached" to applications.

An application will contain one or more *active values* representing its shared data structures. These are replicated in the dialogue managers as *interaction variables.* Editors are derived semi-automatically from applications based on *attributes* (declarative specifications) which the programmer associates with each active value. Users can manipulate their interaction variables independently and can update the corresponding active values using a "commit" mechanism. Applications communicate with editors using calls; editors communicate with applications using callbacks which are registered through the dialogue managers.

Suite's most novel feature is its support for fine-grained control over the degree of *coupling* between a program's multiple user interfaces [Dewan and Choudhary 1995]. This is implemented as event messages representing constraints on interaction variables and is shown by the horizontal arrow in figure 8. Coupling is similar to synchronization but weaker; where synchronized elements exactly replicate one another, coupled elements may include some replicated data and some local data. In Suite, dialogue coupling can vary from strict WYSIWIS (which is equivalent to view synchronization), through relaxed WYSIWIS (where some aspects of the view are shared, others private), to completely asynchronous. The coupling can be controlled along a number of dimensions including time and semantic completeness. This control can be exercised by either the application programmer or the end user.

3.6 Chiron-2

The Chiron-2 (C2) architectural style [Taylor et al. 1996] is a component- and message-based development approach for graphical user interface software. Although not specifically targeted at groupware, C2 can be used to develop groupware in a straightforward fashion; the original C2 paper [Taylor et al. 1996] includes a distributed, multi-user meeting scheduler as an example. C2 derives partly from the architecture of the Chiron-1 system [Taylor et al. 1995] and is influenced by MVC.

Figure 9(a) illustrates the architecture of a hypothetical single-user application designed in the C2 style. The style consists of *components* and *connectors*[2] with specialized properties. Components are independent computational elements, each of which may have internal state and its own thread of control. They communicate with one another via the connectors, which are active elements responsible for routing, broadcasting and filtering asynchronous inter-component *messages.* The internal structure of components is unconstrained by the C2 style, although Taylor *et al.* suggest a wrapper-based mechanism to allow embedding of "legacy components" (components developed in other styles) in a C2 application [Taylor et al. 1996].

Each component and connector has a defined "top" and "bottom", where "up" normally represents more abstract, as in Dewan's generic architecture. The tops of components may attach only to the bottoms of connectors, and vice versa. Connectors may also be attached directly to other connectors, with the same top-to-bottom restriction. Components may attach only to connectors.

[2]Elsewhere in this paper we use the terms "component" and "connector" in a more general sense, following, *e.g.*, [Perry and Wolf 1992; Shaw and Garlan 1996]. In this section the terms are used specifically to denote components and connectors in the C2 architectural style.

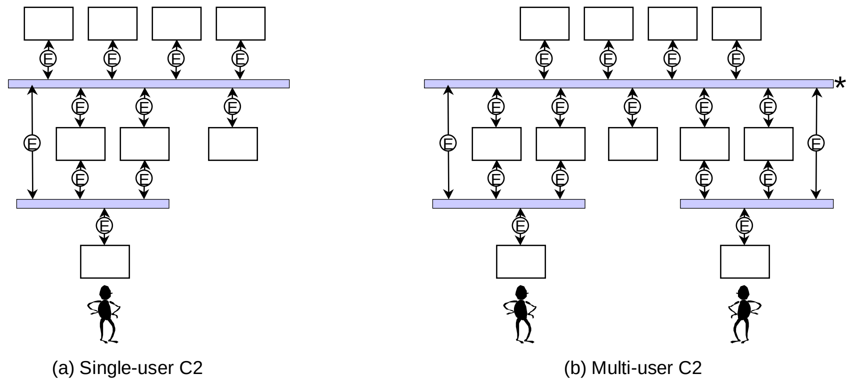


Fig. 9. Examples of the C2 architectural style.

The pattern of messages in C2 is quite similar to that of MVC. Messages flowing up the model are *requests*: directives that an action be performed by one or more components higher in the system. Messages flowing down are *notifications*, that is, announcements of state changes. As with MVC, higher level components (models) can be developed in isolation from lower-level components (view-controllers) but lower level components necessarily rely on the semantics of higher level components.

There are several key differences between C2 and MVC. C2 messages are asynchronous and mediated by the connectors, which may broadcast, selectively transmit, prioritize, or silently absorb them, whereas MVC messages are passed synchronously based on direct calls or callbacks. C2 notifications can be of arbitrary type and are normally information bearing; in the original version of MVC [Krasner and Pope 1988] notifications consist of an announcement of change without indication of what has changed. Finally, C2 components have both a top and a bottom, whereas (in C2 terms) MVC models have only bottoms and MVC view-controllers have only tops.

The C2 connector is similar in spirit to the PAC control element, in that it mediates inter-component communication. However, where a PAC control may incorporate arbitrary computation (including, *e.g.*, mappings between the presentation and abstraction interfaces) the C2 connector is responsible only for routing inter-component communication. Any necessary mappings between components' native interfaces is expected to be performed internally to the components themselves, perhaps by providing a component wrapper.

C2 can be used for groupware by "unzipping" the architectural structure to an appropriate level. Figure 9(b) shows the application of figure 9(a) unzipped for multi-user use. Note that the connector marked by a "*" would span multiple machines and encapsulate the required communication topology and services.

3.7 GroupKit

GroupKit [Roseman and Greenberg 1996] is a widely used toolkit for groupware based on the Tcl programming language and the Tk widget set [Ousterhout 1994]. GroupKit's architectural style is at a lower level of abstraction than the others discussed in this section, since it explicitly exposes distribution and communication

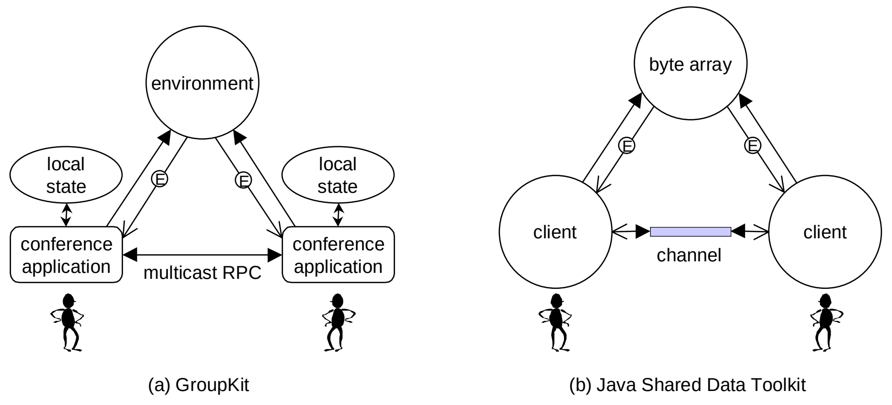


Fig. 10. The architectural styles supported by GroupKit and the Java Shared Data Toolkit.

issues to the GroupKit client programmer. GroupKit is based on the concept of a *conference*, which consists of some number of users who interact via replicated *conference applications*. Users in a conference can interact with one another either directly, via multicast *remote procedure calls* (RPCs), or in an MVC-like indirect style using shared active data structures called *environments*. Users may simultaneously participate in multiple conferences.

The architectural style supported by GroupKit is illustrated in figure 10(a). Each user's conference application is an independent process, running in a Tcl interpreter, with local state. Tcl is a procedural language; GroupKit extends Tcl by allowing a procedure call to be directed simultaneously to multiple applications in a conference. Multicast RPCs can be directed to all applications in a conference (including the sender), to all except the sending application, or to a specified application.

Early versions of GroupKit required that all inter-application communication be via procedure call, which made maintenance of shared data somewhat burdensome [Urnes and Nejabi 1994]. Later versions introduced environments, which are shared active dictionaries (collections of key-value pairs) with keys arranged in a tree structure. Users manipulate environments directly via procedure call; actions on environments automatically generate events which are multicast to all conference members. Applications can bind procedures to events in a highly flexible fashion, specifying the binding based on the type of event (*e.g.* addition or removal of a key) and the affected key's location in the tree [Roseman 1998]. A conference can include an arbitrary number of environments. The environment mechanism has proven extremely useful in the creation of sophisticated large-scale groupware applications [Roseman and Greenberg 1997].

In addition to the architectural features discussed here, GroupKit is also notable for its comprehensive session management facilities [Roseman and Greenberg 1993] (which are borrowed by Clock [Graham 1996]) and its extensive set of group-specific user interface widgets. GroupKit's session management is built on its environment structure. It offers a set of special purpose events indicating that users have joined or left a conference, or that a latecomer to the conference requires updating as to the current session state. Group-specific user interface widgets include multiple cursors, a multi-user scrollbar, and a “radar view” which allows users to see where others are working in a large shared space. GroupKit is the basis of a commercial product, TeamWave Workplace (formerly TeamRooms) which extends the Group-Kit architecture by allowing multiple applications to run in nested interpreters, giving each the illusion of a private variable namespace [Roseman and Greenberg 1997].

3.8 Java Shared Data Toolkit

The Java Shared Data Toolkit (JSDT) [Burridge 1999] is a commercial product available from Sun Microsystems. It is illustrated in figure 10(b). The JSDT is built around a conference model similar to GroupKit’s, where the JSDT conference is called a *session*. *Clients* in a session can interact with one another by sending data over named *channels*, by modifying shared active *byte arrays*, or by manipulating synchronization primitives called *tokens*. The token and JSDT’s built-in session abstraction have been been omitted from figure 10(b). They are accessed via a call/callback event mechanism (called *listeners*) similar to that shown for byte arrays and channels.

Sessions, channels, byte arrays and tokens all support a core set of four events. These advise attached clients when other clients have joined, left, been invited to join, or been expelled from the resource in question. JSDT also provides events specific to each resource type: for instance a client that is a listener to a particular session will be advised whenever a channel, byte array or token has been created or destroyed within the session.

Channels are routing connectors, which can be configured for either “best effort” or guaranteed message delivery. They support four message priorities, and can optionally guarantee in order arrival of all messages of a given priority originating with a given client. A client may attach to any number of channels in a session and may monitor them either via a blocking read or by registering an asynchronous callback. To send data a client first marshals it into a stream of bytes, then invokes one of the channel’s send methods with the byte stream as a parameter. Since most Java objects can be represented as a stream of bytes via Java’s serialization mechanism [Arnold and Gosling 1997], messages can include arbitrarily rich data structures. Messages can be addressed to all attached clients, to all attached clients except the sender, or to a specific client.

The shared byte array provides a service analogous to the MVC model or Group-Kit’s environment. Data in the byte array normally consists of serialized objects, just as messages on channels do. As with channels, any client can attach to any number of shared byte arrays within a session. The shared byte array uses MVC-style notification, in that it only advises its client listeners that something in the array has changed. It is then up to the listeners to reread the array and perform any necessary actions.

Finally, the JSDT token is a synchronization primitive which can be requested by a client, passed from one client to another, grabbed (if free), and released. The semantics of tokens are defined by the client application, and it is expected that higher level synchronization policies would be implemented on top of atomic token operations. There can be any number of tokens in a session.

Each JSDT resource (session, channel, byte array or token) can have an active resource manager associated with it to implement an application-specific management policy. Resource managers provide client authentication and access control; actions normally requiring authentication include joining, creating or destroying a resource. A resource manager can also force a misbehaving client to leave a resource by expelling it. The resource manager mechanism allows JSDT to support applications in which clients are either unreliable or untrustworthy.

3.9 Discussion

In this section we have presented seven distinct architectural styles for groupware, ranging from high-level and abstract to low-level and concrete.

These styles are all intended to provide appropriate abstractions for groupware development. Since groupware has a large user interface component, it is hardly surprising that most of them (PAC*, MVC, Clock, Suite, and Chiron-2) originated as styles for constructing the user interfaces of single user software systems. What is particularly striking is how little change was required to convert them to styles for groupware. In all cases the key was provision of a mechanism for the definition and control of shared state.

PAC*, MVC, ALV, and Clock are all based on components with fixed internal structures, arranged in trees. Recently Tarpin-Bernard *et al.* have proposed a generalization of these styles, expressed as a framework called Multi-Faceted Agents for Collaboration (AMF-C) [Tarpin-Bernard et al. 1998; Tarpin-Bernard and David 1997]. The framework defines each component as an *agent* which consists of a variable number of *facets*. Each agent communicates with others another via *messages* sent to well-defined *ports*. The internal structure of AMF-C agents is guided by a family of design patterns. By constructing an agent with presentation, abstraction, and control facets, the PAC* style may be directly emulated. Other, more complex, patterns are also possible. Since AMF-C does not directly support constraints, it is not clear whether it can directly emulate ALV or Clock.

ALV, Clock, and the Constraint-Imperative programming style [Freeman-Benson and Borning 1992] all share an interaction technique in which user events are handled imperatively but views are updated via declarative constraints. Graham argues that this is the most natural approach: user inputs normally represent demands for action, and therefore have imperative semantics; however, views are defined in relation to their models, and the required relations are most clearly expressed declaratively [Graham 1995].

In the next section we discuss distribution architectures for groupware systems. Here we first mention the distribution approaches supporting the seven architectural styles already discussed. GroupKit provides the programmer with direct control over the distributed design of a groupware system, but requires explicit expression of that design in the application code itself. Chiron-2 encapsulates distribution decisions in its connectors, allowing components to be ignorant of the distributed topology in which they are employed. Applications written in MVC, ALV, Clock, Suite and the JSDT contain no code expressing distribution. Instead, toolkits supporting these styles provide a run-time system that automatically maps applications onto a distributed system. PAC* is relatively abstract, and PAC* designs can be implemented using a number of distribution architectures.

4. DISTRIBUTION ARCHITECTURES

Most of the reference models and architectural styles presented in sections 2 and 3 purposely suppress the distributed system aspects of groupware systems. This allows the designers of groupware applications to focus more directly on problems in the application domain, while largely ignoring distributed implementation concerns. However, groupware systems are ultimately implemented as distributed systems, and the choice of distribution architecture can have a significant impact the system's functional capabilities and performance.

In this section, we examine the range of distribution architectures for groupware systems that have been reported in the literature. The most well known of these are centralized and replicated architectures. In a centralized system the application resides entirely on a shared server, whereas in a replicated architecture each user has a local copy of the application and the applications are somehow synchronized with one another. In addition to these, we discuss a variant of the replicated architecture that includes central coordination, and the semi-replicated architecture in which some aspects of the application are centralized while others are replicated. We conclude with a brief discussion of systems which support flexible distribution, dynamic distribution, and object-based distribution.

To allow precise discussion of the features of these architectures, we begin by presenting Interlace, a new reference model for distributed groupware systems. Interlace focuses on users, processes, and state, their distribution across connected computing platforms, and their patterns of interaction. It provides a high-level framework supporting the precise illustration of a variety of distribution architectures.

4.1 Interlace

There are dozens of groupware applications and toolkits reported in the literature. In almost every description, the authors provide some indication of the distribution architecture of the system described, using widely varying terminologies and techniques to do so. In this section we propose a new descriptive framework, Interlace,[3] to support our presentation of the distribution architectures of groupware systems. Interlace is defined informally below and is illustrated by example in figure 11, which is a verbose version of figure 12(b). See appendix B on page 41 for an explanation of the visual notation used in these figures. All Interlace diagrams in this paper include two users and are drawn mirror-symmetrically, with the users' input processes towards the center.

Interlace represents a groupware system as a collection of users, devices, concurrent processes, and state elements distributed across interconnected computing platforms or sites. Some platforms will have one or more local users while others will operate in a server role. Figure 11 includes two user sites and one server site, indicated by the box-like shaded areas.

In Interlace, each user of a groupware system is supported by one or more input-

[3]Interlace is named for the Celtic interlace style of art, which consists of overlapping, intersecting loops. Celtic interlace designs are topologically similar to Interlace architecture diagrams but are generally much more soothing to look at. They are thought to represent the fundamental interconnectedness of all things.

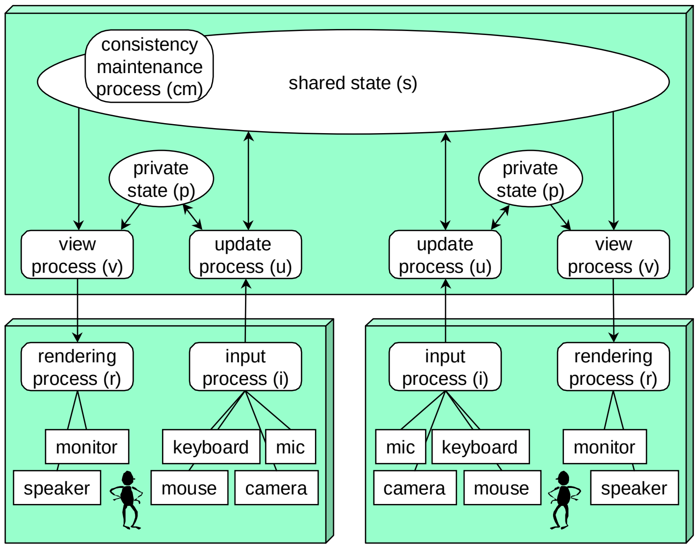


Fig. 11. An example Interlace diagram. The abbreviations shown with each model element are used in the Interlace diagrams in the balance of this section and are repeated for convenience in appendix B.

output *loops* starting and ending with the user and consisting of:

—physical input devices connected to an *input process*, which transforms input into logical interface events;

—a chain of one or more *update processes*, which transform interface events into updates on *state*;

—a chain of one or more *view processes*, which collectively compute an interactive view from the state elements; and

—a *rendering process*, which presents the view to the user on physical output devices.

In figure 11 there are two loops for each user: one through *private* state and one through *shared* state. State and process elements of the model are private if found only in a single user's loop, shared otherwise. Any element in a diagram can be either shared or private. Loops can interlace (join and split) at any point; in figure 11 the two loops supporting each user split at the update process and rejoin at the view process. The combination of shared elements and interlacing loops allows us to model a wide range of systems. This includes, for example, shared editors on a single screen [Bier and Freeman 1991]; such systems cannot be adequately modeled by Zipper-style approaches such as Patterson's or Dewan's [Dewan 1995].

State sharing may be implemented through true sharing (as in the example figure) or by replication with synchronization, which we indicate using Patterson's notation introduced in section 2.2 [Patterson 1995]. Data streams in Interlace may also be synchronized, by which we mean that they contain identically ordered streams of data in approximate temporal synchrony. We indicate this by applying Patterson's notation to the affected arrows (see *e.g.*, figure 13(a) on page 29).

Groupware systems generally require some form of consistency maintenance to ensure that state remains consistent in the face of possibly conflicting updates from multiple users. In Interlace we model this using a *consistency maintenance process* which can appear at various places in users' loops depending on the approach taken by the modeled system. It can also be integrated with shared state elements as in figure 11, in which case consistency maintenance is applied to all access to, and synchronization of, the shared state.

Some updates to shared state may affect private state and vice versa. For example, consider a shared text editor in which each user's cursor position is private, but the document itself is shared. Here, the editor must gracefully handle the case in which one user deletes a paragraph containing another user's cursor. In Interlace, the update process is implicitly responsible for resolving this sort of inconsistency. Similarly, the view process should avoid displaying a view that reflects mutually inconsistent shared and private state. Consistency maintenance is discussed in more detail in appendix A.

The input and rendering processes of groupware systems are typically implemented using a system-level service such as the X Window System, Microsoft Windows, or Java interface primitives. These processes are tightly coupled with the physical input and output devices and are almost always found on users' local machines. For this reason most Interlace diagrams do not include explicit physical devices; however, some example devices are included in figure 11 for illustration. We refer to the input and rendering processes collectively as the *display services* and define the *application* as the update and view processes plus their associated state.

Processes and state elements in an Interlace diagram will rarely map directly onto the actual processes and data structures of the modeled systems. For example, the five processes and three state elements shown at the server site in figure 11 might be implemented by a single-threaded server (one process) accessing a single flat data structure. However, the model would be considered accurate if that server performs independent update and view computations for each client, and maintains both shared information and information private to each client. The focus of Interlace is on what type of computation is performed where, and on the distribution of state across sites.

The sections that follow use Interlace to present the range of distribution architectures that have been proposed for groupware systems. These include *centralized* systems, in which all elements of the application reside on a single computer; *replicated* systems, in which a separate instance of the application is provided locally for each user; and *semi-replicated* systems, in which some elements of the application are centralized while others are replicated. We also present a variant of the replicated architecture which includes a central coordination element, and briefly discuss systems which support *flexible*, *dynamic*, and *object-based* distribution.

4.2 Centralized

In a fully centralized architecture, the application is located on a single server and only the display services are found at the users' sites. Communication from the users' sites to the application is via interface-level events such as X Window events; communication in the reverse direction is via rendering requests. There are two

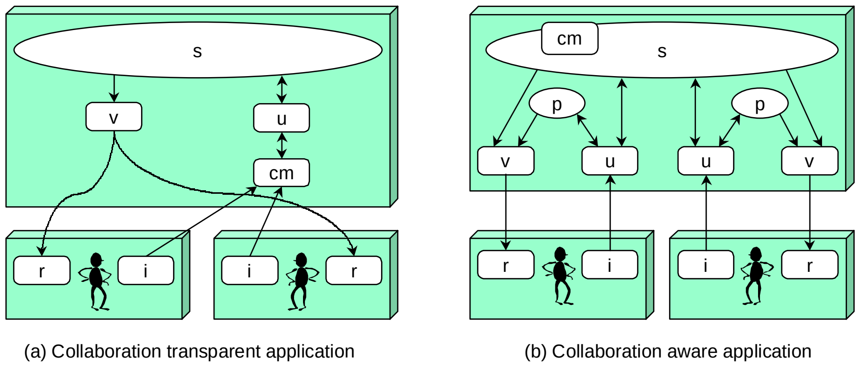

s
v
u
cm
r
i
i
r
(a) Collaboration transparent application
cm
s
p
p
v
u
u
v
r
i
i
r
(b) Collaboration aware application

Fig. 12. Centralized distribution architectures.

distinct variants of the centralized architecture, illustrated in figures 12(a) and (b).

4.2.1 *Variants.* The architecture illustrated in figure 12(a) supports *collaboration transparent* applications, that is, applications which are normally intended for single user use, but are being employed in a multi-user setting. This is achieved either by simply merging the user's input streams or by having the consistency maintenance process enforce a floor control protocol (see appendix A). On the output side, such systems typically rely on modifications to the windowing system or a "pseudo-window-server" to broadcast output events to the user sites. In figure 12(a) this is abstracted by the multiple arrows leaving the view process.

The architecture shown in figure 12(b) is found in *collaboration aware* applications which are specifically designed for use by multiple users. Since each user has private update and view processes and private state, the application can provide relaxed WYSIWIS views of the shared state. Designers adopting this approach can largely ignore distributed system issues since the only distribution is performed by the interface toolkit (*e.g.*, the X Window System).

4.2.2 *Benefits and Liabilities.* The main benefit of the centralized architecture is its simplicity. Since there is only one instance of the application running on a single platform, internal efficiency of the application can be maximized and state consistency can be guaranteed relatively easily. The architecture also provides for accommodation of *latecomers* (users who join a groupware session after it has begun), since it is generally practical to provide them with access to the application's shared state or display [Chung et al. 1993].

Both variants of the centralized architecture tend to be bandwidth intensive and sensitive to network latencies, since communication between the server and the user sites is at the level of interface events in both directions. However, performance is often subjectively acceptable on high-speed local area networks [Abdel-Wahab and Feit 1991; Hill et al. 1994; Urnes 1998].

The collaboration aware variant of this architecture has an additional drawback: poor scalability. If the application's update and view processes are computationally intensive, or if there is a large state storage requirement per user, the resources of the server can quickly become exhausted as the number of users in the groupware session grows [Graham and Urnes 1992]. This problem is compounded by the fact that changes in the shared state will normally require view recomputation to be performed simultaneously for all users [Hill et al. 1994]. Scalability is also a factor for collaboration transparent applications, although to a lesser degree [Lauwers et al. 1990].

4.2.3 *Implementations.* Many "shared window systems" including XTV [Abdel-Wahab and Feit 1991], HP Shared-X [Garfinkel et al. 1994], and Microsoft Netmeeting [Microsoft Corporation 1997] are based on the collaboration transparent centralized architecture. Often in such systems, one of the user sites will double as the server.

The Rendezvous system (section 3.3) was implemented using a centralized, collaboration aware architecture since this avoided the requirement for a distributed constraint implementation. Its designers considered adapting Rendezvous to other distribution architectures [Hill et al. 1994], but implementations were never completed. The Clock run-time system (section 3.4) adopts this distribution architecture by default; however, Clock can also provide a semi-replicated architecture (section 4.5). The centralized architecture is also used in many so-called "multi-user dimensions" (MUDs) such as LambdaMOO.

4.3 Replicated

A fully replicated architecture is the opposite of a fully centralized one: here all data and computation is replicated at all sites. As with the centralized architecture, there are two main variants catering for collaboration transparent and collaboration aware applications.

4.3.1 *Variants.* The collaboration transparent variant of the replicated distribution architecture is illustrated in figure 13(a). Since the internal state of collaboration transparent applications is not externally accessible, direct state synchronization is not generally possible. Instead, synchronization of input streams is the approach most frequently used. As in the centralized case, inputs are routed through a consistency maintenance process which may implement a floor control policy. In the centralized architecture, the consistency maintenance process produces a merged input stream directed at the single update process; in the replicated case it produces a set of synchronized input streams directed at the replicated update processes. As long as each update process is deterministic and receives identical input, the replicated applications can be expected to operate in synchrony [Lauwers et al. 1990].

The collaboration aware variant of the replicated architecture is illustrated in figure 13(b). Collaboration aware replicated applications are typically implemented by synchronizing state rather than inputs. This allows for flexibility in selection of concurrency control protocols and provides for local state and relaxed WYSIWIS views.

4.3.2 *Benefits and Liabilities.* An obvious liability of replicated distribution architectures is the requirement that a separate copy of the application execute at each users' site. This means that replicated applications require more aggregate resources (processing power, memory, software licenses, *etc.*) than equivalent cen-

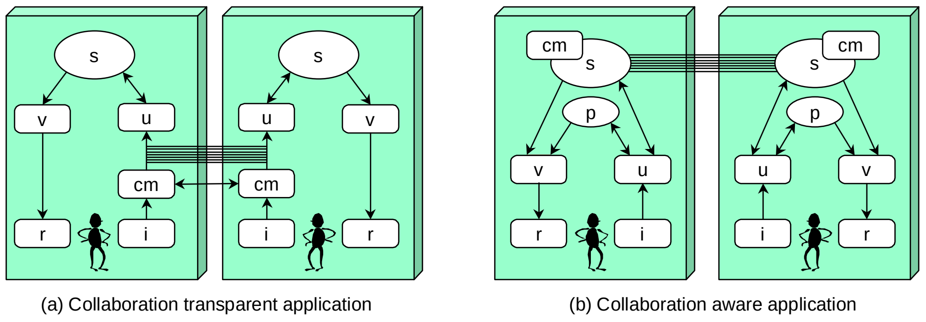

s
v
u
cm
r
i
cm
p
(a) Collaboration transparent application
(b) Collaboration aware application

Fig. 13. Replicated distribution architectures.

tralized applications. This is especially true for collaboration transparent applications. In environments with a mix of machine types and operating systems, the requirement for identical applications at each site can become a significant constraint, although multi-platform systems like Sun's Java [Arnold and Gosling 1997] mitigate this somewhat [Begole et al. 1997].

Collaboration transparent applications with a replicated architecture may use less network bandwidth than equivalent centralized applications. In the centralized architecture, input events are unicast to the server and output events are multicast from the server to all participants. In the replicated architecture, output events are not distributed, since they are computed locally, but input events must be subject to concurrency control and multicast to all sites. While some argue that the replicated case is an improvement [Lauwers et al. 1990], there is little data to support this contention.

For collaboration transparent applications, the fully replicated architecture has a significant number of liabilities, which have been documented in detail elsewhere *et al.* [Lauwers et al. 1990]. These include the difficulties of ensuring input consistency and ordering, ensuring output consistency, and of maintaining single application semantics when multiple copies are running. For example, if a user in a replicated application selects the save operation, should a file be saved by each replica or only by one? Latecomers are particularly problematic for fully replicated, collaboration transparent applications. Since a late joiner cannot simply be given a copy of the current application state, it is necessary to save a (possibly compressed) copy of the entire input stream, which is "played back" to synchronize the latecomer with the other session participants [Chung et al. 1998].

For collaboration aware applications, the main benefit of the replicated architecture is enhanced interface responsiveness. If an optimistic concurrency control algorithm is used (see appendix A), updates to shared state can be performed locally and are unaffected by network latency. A further benefit is that a replicated architecture distributes the computationally expensive view and update processing to the users' computers. This would be expected to lead to better scalability; however, this depends on the overhead incurred in synchronizing the state of the replicated instances, which may be significant.

4.3.3 *Implementations.* Collaboration transparent shared window systems that have been implemented using this distribution architecture include MMConf [Crowley et al. 1990], VConf [Lantz 1986], and Dialogo [Lauwers et al. 1990].

Collaboration aware groupware systems supporting a replicated architecture include DreamTeam [Roth and Unger: 1998], Mushroom [Kindberg et al. 1996], GroupDesign [Karsenty et al. 1993], GINA [Berlage and Genau 1993] and the original version of COAST [Schuckmann et al. 1996].

4.4 Centrally Coordinated

The centrally coordinated distribution architecture is similar to the fully replicated architecture except that the consistency maintenance process is centralized. As with the fully replicated architecture, there is a variant supporting collaboration transparent applications and a variant supporting collaboration aware applications.

4.4.1 *Variants.* The collaboration transparent variant of the centrally coordinated architecture is shown in figure 14(a). It is directly comparable to its fully replicated counterpart, providing synchronized input streams to a replicated application. The principal difference is that where the fully replicated architecture requires execution of a distributed algorithm to ensure input stream synchronization, this architecture allows a much simpler, centralized concurrency control scheme to be used.

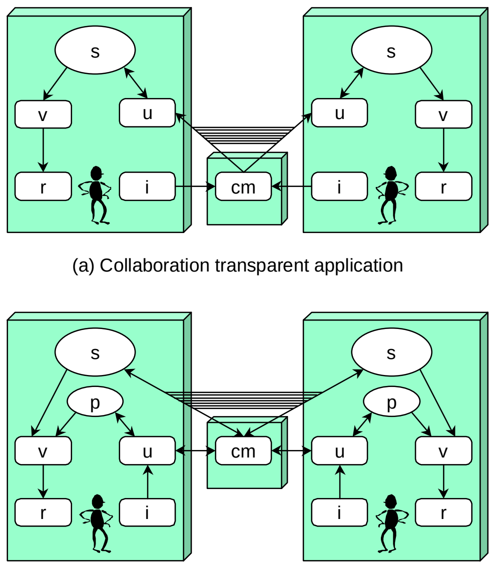
For collaboration aware applications, the architecture shown in figure 14(b) is most frequently used. It is different in principle from its fully replicated counterpart in that the latter uses active state synchronization whereas this architecture uses synchronization of access to the replicated state. While it is possible to imagine a version of this architecture based on state synchronization, we were unable to identify any actual instances of such an architecture's use.

4.4.2 *Benefits and Liabilities.* The centrally coordinated distribution architecture shares most of the benefits and liabilities of the fully replicated architecture.

The primary benefit of adding central coordination to a replicated architecture is the relative simplicity of implementing consistency maintenance using a centralized algorithm rather than a distributed one. This has motivated at least one groupware system (COAST) to migrate from a fully replicated architecture to one including central coordination [Schuckmann 1998].

The main additional liability imposed by central coordination is the system's reliance on a single consistency maintenance server: if the server fails, the entire system is rendered useless. As well, user interface response of a centrally coordinated application will generally be poorer than that of a purely replicated application with optimistic concurrency control, since each update to shared state will require a minimum of two network transmissions.

4.4.3 *Implementations.* Collaboration transparent systems which have been implemented using this architecture include the Java Collaboration Environment (JCE) [Abdel-Wahab et al. 1999] and Java Applets Made Multi-user (JAMM) [Begole et al. 1997; Begole et al. 1998]. Collaboration aware systems include NCSA Habanero [Chabert et al. 1998], the Prospero system [Dourish 1996a], Ensemble [Newman-Wolfe et al. 1992], and the most recent version of COAST [Schuck-



(a) Collaboration transparent application

(b) Collaboration aware application

Fig. 14. Centrally coordinated distribution architectures.

mann 1998].

JAMM, JCE and Habanero all rely on the Java Abstract Windowing Toolkit (AWT) event mechanism [Chan and Lee 1998], in which interface objects post events to a queue from which they are dispatched to other objects registered as “event listeners”. In Habanero events are manually routed to a central queue, which forwards them to all applications in the groupware session. JAMM and JCE produce a similar effect for collaboration transparent applications by patching the AWT’s internal event mechanism. JCE works with Java applications, while JAMM works with Java applets (applications based on mobile code, which are typically embedded in web pages) [Chan and Lee 1998].

4.5 Semi-replicated

In a semi-replicated distribution architecture some aspects of computation and state are replicated while others are centralized. The policy for determining what is centralized and what is replicated may vary with the application or system. One approach is to to centralize shared state and processes and replicate private ones. Another strategy is based on a “zipper-style” reference model (see sections 2.2 and 2.3) where the branches are replicated while the stem is centralized.

4.5.1 *Variants.* Figure 15(a) shows a semi-replicated architecture in which shared state and its associated update and view processes are centralized, while private state is maintained locally at each user’s site. The architecture is suitable for both collaboration aware and collaboration transparent applications. In effect, this is a classic client/server architecture.

For collaboration transparent applications, this architecture can be viewed as an

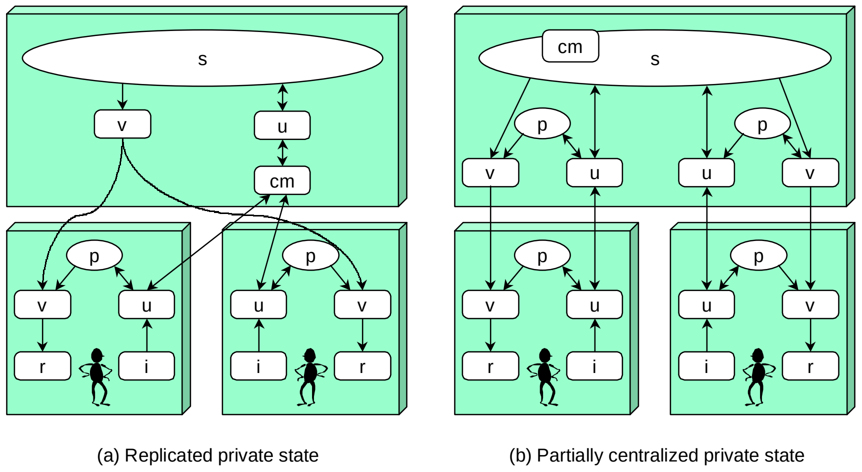

s
v
u
cm
p
v
u
r
i
p
u
v
i
r
(a) Replicated private state
cm
s
p
p
v
u
u
v
p
v
u
r
i
p
u
v
i
r
(b) Partially centralized private state

Fig. 15. Semi-replicated distribution architectures.

extension of that shown in figure 12(a). Here, as in that figure, the single-user application runs on the server, with a consistency maintenance process insulating the application from its multiple users. The difference is that here the window-sharing system managing the shared application provides each user with some private state. This state may allow the user to prepare work privately, then publish it to the shared application, or it may play a role in the management of the groupware session.

For collaboration aware applications, figure 15(a) can be seen as a modification of figure 13(b) in which the shared state has been centralized on a server. In this case the private state and its associated processes form an integral part of the application rather than being an extension to it.

The architecture shown in figure 15(b) is a multi-threaded client/server system in which the server maintains some private state for each client. The architecture shown in figure 16(a) is quite similar, except that rather than centralized private state we have replicated shared state. It is also possible to imagine (though difficult to draw!) an architecture incorporating both strategies. Both of these architectures combine features of figures 12(b) and 13(b). They are suitable only for collaboration aware applications.

4.5.2 *Benefits and Liabilities.* As might be expected, the semi-replicated architecture provides a mix of the benefits and liabilities of the centralized and replicated architectures. There is some evidence that with careful tuning the benefits can outweigh the liabilities [Urnes and Graham 1999].

For collaboration transparent applications, the semi-replicated architecture is more flexible than the centralized architecture and accommodates latecomers better than the fully replicated architecture. Collaboration aware semi-replicated applications generally scale better than centralized ones, since computationally intensive view and update processes can execute at the user sites [Graham et al. 1996]. They are also simpler to develop, since consistency maintenance can be managed centrally rather than via a distributed algorithm. If the protocol between the user sites and the server site is standardized, then a variety of user applications can access the shared data simultaneously [Day 1997].

The principal liability of the semi-replicated architecture is that responsiveness of the user interface may be impacted by network latencies between the user sites and the server. This effect can be mitigated by the introduction of caches at the user sites, as in figure 16(b), at the expense of additional computational and storage overhead [Graham et al. 1996]. Caches can also be added to the architectures shown in figures 15(b) and 16(a).

The semi-replicated architecture also shares the fully replicated architecture's liability of requiring that the replicated applications be available at each of the user sites. Here, as there, multi-platform technologies like Java somewhat reduce the importance of this issue.

4.5.3 *Implementations.* The semi-replicated distribution architecture shown in figure 15(a) underlies the Notification Service Transfer Protocol (NSTP) proposed by Day *et al.* [Day 1997; Day et al. 1996; Patterson et al. 1996]. NSTP has been used as the basis of a number of synchronous groupware applications developed with loosely-coupled client components interconnected via a bus-like structure [Mitchell 1998]. The applications themselves are Java applets, which overcomes the difficulty of distributing them across heterogenous computing environments.

Suite [Dewan and Choudhary 1992] and those GroupKit applications incorporating both shared environments and multicast remote procedure calls [Roseman and Greenberg 1996] have distribution architectures that are a variant of figure 15(a). The key difference is that both these systems support direct communication between user sites, in the form of events in Suite and of remote procedure calls in GroupKit. In Interlace this communication is modeled by adding a bidirectional arrow between the replicated update processes of figure 15(a).

Both Weasel [Graham and Urnes 1992] and Clock [Graham et al. 1996] can implement applications using semi-replicated distribution architectures. Weasel's distribution architecture is that of figure 15(a). Clock's distribution annotations (see section 4.6) allow it to create applications with any of the distribution architectures shown in figures 15 and 16. Experiments suggest that applications exhibit the best responsiveness when implemented using an architecture similar to that of figure 16(a), with the addition of client-side shared state caches and a high-performance, semi-optimistic concurrency control algorithm [Urnes 1998; Urnes and Graham 1999].

Variants of the semi-replicated architecture are also found in the JSDT [Burridge 1999], Jupiter [Nichols et al. 1995], Promondia [Gall and Hauck 1997], the DOLPHIN system [Streitz et al. 1994], Bentley's system for air traffic control [Bentley et al. 1992], and Neil Stephenson's fictional Metaverse [Stephenson 1993].

4.6 Flexible Systems

Systems and toolkits supporting architectural styles like those discussed in section 3 must ultimately map the programmer's design to a distributed architecture. For the most part these toolkits either provide a single distribution architecture for all applications (*e.g.*, centralized, as in Rendezvous) or require the programmer to mix code reflecting distribution decisions with code reflecting application functionality

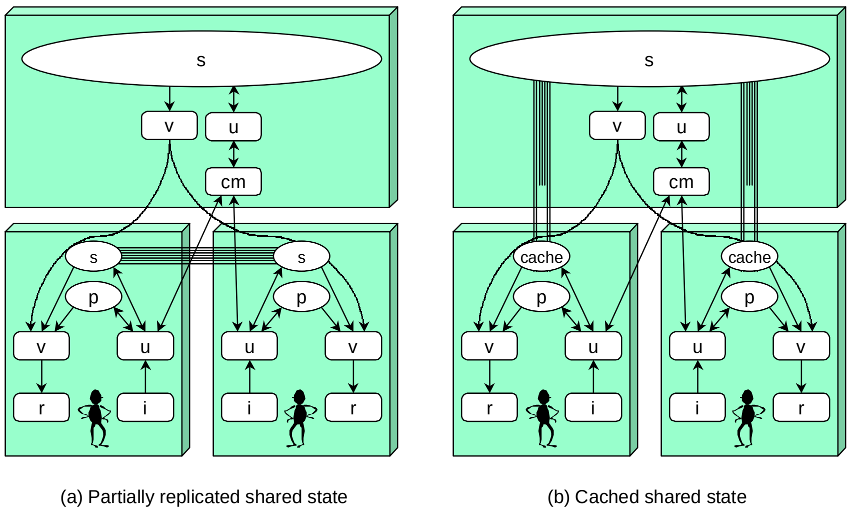


Fig. 16. More semi-replicated distribution architectures.

(*e.g.*, as in GroupKit). Systems supporting *flexible distribution* express application semantics separately from application distribution. This allows the programmer to first implement the required functionality, then adjust the run time distribution architecture to suit application needs. Two systems supporting flexible distribution are the GroupEnvironment (GEN) [O'Grady 1996] and Clock (see section 3.4). Dourish's Prospero system was originally intended to incorporate flexible distribution [Dourish 1995] but this goal was later eliminated in favour of providing a novel, flexible, consistency maintenance mechanism [Dourish 1996a; Dourish 1996b].

GEN provides flexible distribution policies via an *open implementation* strategy [Kiczales et al. 1997]. A toolkit with an open implementation provides the developer with access both to the usual application programmer interface (API) and to a *meta-interface* offering principled control over elements of the toolkit's implementation. This allows the developer to extend or modify the toolkit's API and to alter the internal function of the toolkit in a constrained fashion. In GEN the aspects of the toolkit which are exposed via the meta-interface are those concerning distribution.

The GEN toolkit includes default implementations for replicated and centralized distribution architectures. In [O'Grady 1996], O'Grady demonstrates that the toolkit can be extended via the meta-interface to provide for selective routing of messages, object migration, and optimistic consistency maintenance schemes. GEN was an experimental prototype and is no longer maintained. However, recent releases of GroupKit incorporate a restricted version of GEN's meta-interface for the management of GroupKit environments [Roseman 1998].

In Clock, distribution flexibility is provided by the addition of *annotations* to the ClockWorks architecture diagrams [Urnes 1998; Urnes and Graham 1999]. These annotations act as hints to the Clock run time system and allow the programmer to specify the location of the client/server split, the caching policy to be used by the clients, the concurrency control algorithm to be applied, and which shared ADTs can be safely replicated to user sites. The annotations are guaranteed to preserve Clock's formally defined semantics [Graham 1995], although they necessarily affect application behaviour under different failure scenarios.

A ClockWorks diagram with no annotations will be implemented using Clock's default centralized distribution architecture. Those designs including an indication of the client/server split will be implemented using a variant of the semi-replicated architecture; exactly which one will depend on the other annotations selected. Modifying the annotations provides a safe and simple mechanism for experimenting with different distribution strategies.

The main difference between the GEN and Clock approaches is that GEN allows the application developer program-level control over the application's distribution architecture, while Clock allows the developer to select from distribution options already present in the Clock run time system. Clock's approach is simpler, but GENs is more flexible. Which approach will prove more useful in practice is an open question.

4.7 Dynamic Systems

Most groupware systems, including the flexible systems discussed above, demand that the developer choose the system's distribution architecture at design time. This requires that the developer make assumptions about such characteristics of the run time environment as the number, capabilities, and relative usage costs of the system's user and server sites; the sites' interconnection topology; the network bandwidths and latencies between them; and the variability of any of these factors over time.

It seems unlikely that any one distribution architecture can be appropriate for all circumstances. This has motivated suggestions that the distribution architecture of a groupware system should be chosen at run time and adjusted dynamically during the interactive session [Dourish 1995; Greenberg and Roseman 1999; O'Grady 1996; ter Hofte 1998]. Adjustments could include dynamic replication or centralization of state or processes at either a coarse- or fine-grained level. Research into migratory applications [Black et al. 1987; Johansen et al. 1995; Jul et al. 1988], which allow executing processes to be relocated transparently across a network, suggests that such an approach may be viable. We are unaware of any groupware system that dynamically adjusts its distribution architecture at run time in the full sense suggested here; however, two systems which do provide a restricted sort of dynamic distribution are Chung and Dewan's enhanced XTV [Chung and Dewan 1996] and the Visual Obliq system [Bharat and Brown 1994; Bharat and Cardelli 1995].

Enhanced XTV supports collaboration transparent applications using the centralized architecture shown in figure 12(a), with one of the client machines doubling as the application server. The system allows the server to migrate from one client to another, either to optimize responsiveness for the client that "has the floor", or to account for session changes such as the pending shut down of a server machine. An obvious mechanism for server migration would be to save and ship a binary image of the running application; unfortunately this is not generally possible under Unix. Instead, enhanced XTV relies on a sophisticated event logging and playback mechanism to synchronize the new server with the existing session.

Visual Obliq allows the construction of collaboration aware distributed applications by direct manipulation. It is based on Cardelli's Obliq distributed programming language [Cardelli 1994], in which objects have locations but computation can roam the network. Obliq provides a mechanism for the atomic replication and relocation of objects to new sites. Visual Obliq uses this facility to initialize the user interface of a groupware application at a server site and ship it to the user site where it continues executing [Bharat and Brown 1994]. The actual distribution architecture of an executing Visual Obliq application is roughly the semi-replicated architecture of figure 15(a). Collaboration aware Java applets (*e.g.*, [Mitchell 1998]) operate in an analogous fashion.

In [Bharat and Cardelli 1995], Visual Obliq is extended to allow complete graphical applications to migrate from one computer to another. However, the implementation specifically excludes migratory multi-user applications, for which connectivity would need to be maintained during migration. How best to address this issue remains an open research question.

4.8 Object-based Distribution

The four distribution architectures presented in sections 4.2 through 4.5 are all rather large-grained and based on a classic client/server computing model. Several recent systems are based instead on fine-grained, distributed object models. These include DECAF [Strom et al. 1998], AMF-C [Tarpin-Bernard et al. 1998], and TeleComputing Developer (TCD) [Anderson et al. 2000].

DECAF is a variant of MVC which includes combined view-controllers (*views*), *models*, and *transactions*. Transactions are created by views whenever updates are to be performed. DECAF views and transactions are always found at user sites, but the models they interact with may be arbitrarily distributed. An application will consist of many views, and a single view may be attached to many models at many locations. Further, models may be arbitrarily replicated and connected to one another via synchronizing *replica relationships*. DECAF provides atomic transactions and replica synchronization using a distributed consistency maintenance algorithm, which is briefly discussed in appendix A.

Tarpin-Bernard *et al.* argue that AMF-C (briefly described in section 3.9) is an ideal candidate for distributed groupware implementations since its facet boundaries provide natural "fragmentation points". These allow individual agents to be split across network boundaries. Each AMF-C agent may have one of the four distribution architectures already discussed, and a complete agent system may have an arbitrarily complex mix of these.

The TCD system provides a distribution architecture (called Dragonfly) for systems designed in the Clock architectural style. It extends Clock's flexible distribution mechanism by allowing each Clock component to incorporate a separate caching and consistency maintenance strategy [Wright 1999]. Dragonfly also allows each component in a Clock architecture to be implemented at a separate site, although this is expected to be rare in practice.

All three of these systems provide the developer with great degree of flexibility in choosing an actual distribution architecture for an application. Guidelines for effectively employing this flexibility have yet to be developed.

4.9 Discussion

The general trend in distribution architectures has been from simple to complex. Earlier systems tended to be either fully centralized or fully replicated. More recent systems have tended to use more complex distribution architectures including centrally coordinated, variants of semi-replicated, or object-based strategies. Additionally, there has been some recent experimentation with flexible specification of distribution architectures and with systems which dynamically reconfigure their distribution architectures at run time.

The distribution architecture of an application must satisfy a number of conflicting concerns. Depending on the application, these may include responsiveness, flowthrough, and consistency (discussed in appendix A); effective use of bandwidth and machine resources; scalability; fault tolerance; provision of persistent storage; accommodation of external resources (*e.g.*, large databases or non-groupware client/server systems); and support for temporarily disconnected users. No one distribution architecture will satisfy all concerns for all applications. It therefore appears likely that flexible and dynamic distribution will become necessary as groupware systems move out of the research lab and into the mainstream.

5. CONCLUSION

Software architectures represent codified solutions to commonly occurring problems. In this paper we have presented three classes of such solutions for the problem of developing groupware: reference models, architectural styles, and distribution architectures. Each addresses a different problem which must be solved. Reference models suggest overall structures for groupware systems. Architectural styles provide operational answers to questions of detailed design. Distribution architectures express the distribution of computation and state across multiple sites.

Here, we have offered only glimpses into the groupware problem domain itself. Indeed, groupware researchers are still very much in the process of identifying the key requirements for effective groupware systems [Graham and Grundy 1998]. As the the problems posed by these requirements become better understood, the architectural solutions presented here will necessarily be adapted, extended, and replaced.

Ultimately, we expect that support for groupware applications will be incorporated directly into mainstream operating systems, in much the same way that support for graphical applications has been gradually added over the past fifteen years. The current challenge is to determine the architectural abstractions and infrastructure that such support will require.

ACKNOWLEDGMENTS

This paper has benefited immensely from discussions with Gary Anderson, Tore Urnes, Tim Wright, and the members of IFIP Working Group 2.7/13.4. Germinal Boloix, Stefan Bruda, David Lamb, Laurie Ricker, Jörg Roth, Terry Shepard, and David Skillicorn provided insightful comments on various drafts. Our thanks to all.

APPENDIX

A. CONSISTENCY MAINTENANCE

The design of a groupware system typically involves tradeoffs between the desirable properties of *responsiveness*, *predictability*, and *consistency*. Meeting any two of these goals is typically straightforward, but it appears impossible to meet all three without compromise [Karsenty and Beaudouin-Lafon 1993]. In this appendix we first define these properties, then present some consistency maintenance strategies for groupware and their effects on responsiveness and predictability.

A.1 Definitions

Responsiveness. Responsiveness refers to the system's observable reaction to user input, which must be rapid enough to avoid degrading user performance. For common tasks such as typing or mouse motion, a total delay of less than 50–150 milliseconds from input to observation is required [Shneiderman 1998]. For other tasks a slower response time may be acceptable.

Predictability. A system is predictable if it faithfully executes actions requested by the user and provides a reasonable degree of explanation for any other events occurring in the interface. For example, in a shared editor the disappearance of a paragraph from one user's display might be "explained" by the fact that another user was editing in that region. This kind of explanation requires mutual *awareness* among users [Dourish and Bellotti 1992], and *flowthrough* of each user's actions to the other users [Dix 1996] .

Consistency. There are several types of consistency requirements for distributed systems [Coulouris et al. 1994]. For the purposes of this paper, the most important of these are update and replica consistency. Update consistency requires that all multi-valued updates appear as atomic. Replica consistency requires that if a single data element is replicated, all replicas of it are maintained in an identical state. A weaker version of replica consistency requires only that all replicas reach identical states at quiescence. Replica consistency can be seen as a special form of update consistency, and cache consistency can be seen as a special form of replica consistency.

Consistency maintenance approaches for groupware can be classed as either *pessimistic* or *optimistic*. Pessimistic approaches emphasize predictability over responsiveness and disallow any action which could potentially violate consistency. Optimistic approaches are motivated by the observation that in groupware systems with good awareness and flowthrough, semantic inconsistencies between user actions are relatively rare [Stefik et al. 1987]. The optimistic approach therefore emphasizes responsiveness over predictability and allows inconsistencies to arise, possibly repairing them later.

A.2 Pessimistic Approaches

Most pessimistic approaches to consistency maintenance are based on *serialization* and *locking*. Strict serialization involves globally ordering all updates and requiring that they be executed in order at all sites. In single user applications, serialization is often sufficient to ensure reasonable application behaviour. However, in multi-user applications the interleaving of conflicting updates can cause inconsistency or unpredictability. For example, consider two users simultaneously attempting to move an object in a graphical editor. If one moves the object left while the other moves it right, serialization alone might result in the object bouncing back and forth between the two users' cursors. To prevent this, many systems require that an update process acquire an exclusive lock on shared state (in this case the object's position) before modifying it. Update consistency may also demand that an update process acquire a lock on all state that is to be read from or written to during an update, before calculating and applying the modifications.

Locking is transaction-based. Each transaction is typically broken down into three phases: *lock acquisition*, where the update process *requests* exclusive access to a set of objects and the system (eventually) *grants* or *denies* it; manipulation of the objects themselves; and *release*, where control of the objects is returned to the system. Locks can be requested explicitly by the user or implicitly by the update process based on user actions [Newman-Wolfe et al. 1992].

Lock *granularity* can significantly impact the user interface behaviour of a groupware application. At one extreme, the lock can be imposed at the level of the entire application. This approach is called *floor control* by analogy to the "social protocol" often used in large meetings [Boyd 1993; Shen and Dewan 1992]. Changes in possession of the floor are mediated either by a user with special privileges (who "chairs" the session), or by the current floor holder, or in an ad hoc fashion. At the other extreme, locking can be performed on the most primitive objects available in the application. This can allow a high degree of concurrent user interaction, at the expense of possibly-considerable lock maintenance overhead. Still more concurrency can be provided by separating locks for reading data from locks for writing to it.

Waiting for locks introduces delay and degrades responsiveness. If the lock is unobtainable, or if the lock request must traverse a slow or intermittent network link, or if the lock granting process is computationally expensive, the observed degradation can be significant. Several optimistic and semi-optimistic consistency maintenance strategies have been proposed to overcome this problem [Cormack 1995a; Ellis and Gibbs 1989; Karsenty and Beaudouin-Lafon 1993; Strom et al. 1998; Urnes and Graham 1999]. In general they allow the user to update local state replicas immediately (optimizing responsiveness) and then propagate the updates to remote replicas. The challenge is to ensure that an acceptable degree of consistency and predictability is maintained in the process.

A.3 Optimistic Approaches

The most optimistic approach to consistency maintenance is not to provide it at all. Greenberg and Marwood argue that inconsistencies are acceptable in applications like pixel-oriented shared whiteboards that are intended for informal communication rather than the production of a common artifact [Greenberg and Marwood 1994]. If two users draw intersecting lines (say, one red and one blue), that appear in opposite orders on their two whiteboards, then the pixel at the lines' intersection will be red on one board and blue on the other. This kind of inconsistency is unlikely to cause the users any confusion and can therefore be tolerated. Inconsistencies can also be tolerated where they can easily be detected and resolved by social protocols acted out by the system's users [Stefik et al. 1987].

Locking at one semantic level can remove the requirement for consistency maintenance at lower levels. In our example of the shared graphical editor, once a lock on the object's position is obtained the application can move the object without applying consistency maintenance to each individual motion. If the node's position data is locally replicated, this can provide a dramatic improvement in interface responsiveness [Urnes and Graham 1999].

Where some degree of consistency maintenance is required and locking provides unacceptable performance, a fully or semi-optimistic algorithm may be employed. Fully optimistic algorithms include the distributed OPerational Transform (dOPT) [Ellis and Gibbs 1989], the Calculus for Concurrent Update (CCU) [Cormack 1995a], and the Optimal RESponse TimE (ORESTE) algorithm [Karsenty and Beaudouin-Lafon 1993]. These allow updates to be made to local state immediately (providing optimal responsiveness), then propagated to remote replicas for flowthrough. The algorithms sacrifice short term consistency, but guarantee that all replicas will reach a consistent state at quiescence.

dOPT defines a partial order of all operations in the system (roughly, Lamport's *happened before* relation [Lamport 1978]) and guarantees that all events in the partial order are executed in order at all sites. Where two operations o and p are not in the partial order, dOPT has all sites execute either $o' \circ p$ (p followed by o') or $p' \circ o$, where o' and p' are transformations of o and p chosen such that $o' \circ p$ and $p' \circ o$ have the same effect given the same initial state. CCU, which corrects a subtle flaw in dOPT [Cormack 1995b], works in essentially the same way. Both CCU and dOPT require the programmer to provide a transformation function for every ordered pair of operations possible in the system. This gives n^2 transformation functions for a system with n distinct operations. Choosing transformations which are both correct and predictable is a challenging and apparently sometimes unachievable task [Nichols et al. 1995].

ORESTE is based on a total order of all events in the system. Where operations are determined to have been executed out of order at a site, and the order is determined to have caused a state inconsistency, that site will "undo" the offending operations and "redo" them in the correct sequence. This requires that all operations in the system include a mechanism to undo them and that all ordered pairs of operations be categorized as to whether they conflict, safely commute, or mask one another. The rollback and reapplication of operations in ORESTE can be reflected as unpredictable behaviour in the user interface.

Semi-optimistic algorithms include Clock's eager concurrency scheme [Urnes 1998; Urnes and Graham 1999] and that of the Distributed, Extensible Collaborative Application Framework (DECAF) [Strom et al. 1998]. In both of these approaches, update calculations are allowed to proceed using possibly inconsistent data. Then, before updates are applied, the consistency of the source data is verified either centrally (in Clock) or by execution of a distributed algorithm (in DECAF). If an inconsistency is detected, the update may be restarted or aborted. However, since inconsistencies are rare in practice, these mechanisms provide considerably higher responsiveness than lock based systems, even allowing for the overhead of multiple transaction retries [Urnes and Graham 1999].

In Clock all views are pessimistic; that is, a view only displays the results of successfully committed updates. In DECAF, views can be either pessimistic or

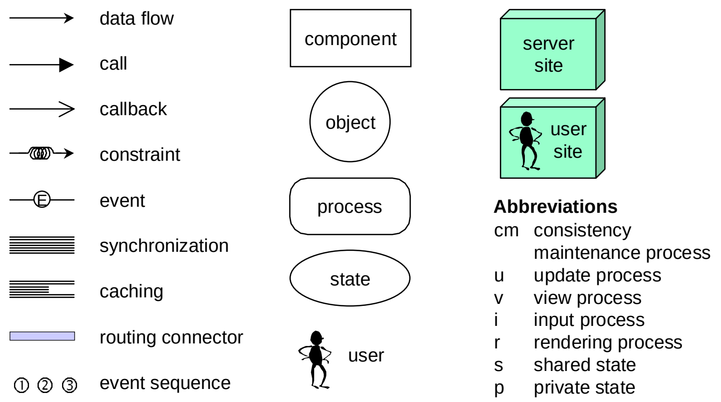


Fig. 17. Key to the visual notation and abbreviations.

optimistic. Optimistic views sacrifice predictability, but provide interface responsiveness directly comparable to that of fully optimistic algorithms. DECAF allows the view display policy to be switched between optimistic and pessimistic at run time to compensate for changing network conditions [Banavar et al. 1998].

A.4 Conclusion

This appendix has presented a brief overview of consistency maintenance approaches for groupware. Good starting points for further reading include [Greenberg and Marwood 1994; Kindberg 1996; Munson and Dewan 1996; Wright 1999].

B. VISUAL NOTATION

The visual notation used in this paper is presented in figure 17 and described informally below. Abbreviations used in the Interlace diagrams of section 4 are also listed in the figure.

Data flow. Data flow connectors represent the directed movement of information between components. The actual type of connection (call, callback, constraint, *etc.*) is unspecified. The arrowhead(s) represent the direction(s) of flow.

Call. A call is an explicit procedure call or method invocation, which may have a returned value. The caller requires knowledge of the calling structure of the callee. The arrowhead indicates the callee.

Callback. A callback is an anonymous call, registered by the callee with the caller using some predefined mechanism. Callbacks rarely have return values. The arrowhead indicates the callee.

Constraint. A constraint expresses an automatically maintained relationship between a *source* and a *target*. Changes in the source are propagated to the target by an underlying constraint maintenance system. Constraints are commonly found in functional and constraint programming systems. The arrowhead indicates the constraint's target; a double-headed arrow indicates a bi-directional constraint.

Event. An event is a notification that something of significance has occurred. It may be transmitted via data flow, call, or callback, and is indicated by decorating an arrow with the event symbol. Whereas a call or callback normally requires action on the part of the callee, an event may be ignored by the recipient.

Synchronization. Synchronization is an identity relation on replicated state elements or data streams. Synchronized state elements contain exactly the same information at quiescence but may diverge when updates are pending. Synchronized data streams transmit identical sequences of information in approximate temporal synchrony. Adapted from [Patterson 1995].

Caching. Caching is a restricted form of synchronization in which only some portion of the shared state is replicated in the cache. Caches serving different users will typically contain different subsets of the shared state, their contents will vary over time, and active maintenance of cache coherence is necessary to support semantically correct application behaviour.

Routing connector. A routing connector encapsulates information about the interconnection topology of the elements it connects and actively routes information between those elements. Adapted from [Taylor et al. 1996].

Event sequence. Dynamic behaviour of systems is illustrated using event sequence numbers to indicate ordering. Adapted from the Unified Modeling Language collaboration diagram notation [Booch et al. 1999].

Component. A component may contain processes, state, and objects, and may have an arbitrarily rich internal structure. Physical devices are also represented as components.

Object. Objects include encapsulated state, operations which can be invoked on the state, and a well-defined interface to those operations. This includes abstract data types as well as "objects" in the sense normally meant in discussions of object-oriented programming languages. Objects may also encapsulate a concurrent process element, in which case they are *agents*.

Process. A process element represents some computation performed by the system based on data represented separately, in either in a state or object element. Process elements are normally concurrent.

State. State elements store information which is read and updated by separate process elements.

User. A human (or possibly human-like agent) user of a groupware system.

Server site. A computer which does not interact directly with a user. Adapted from the Unified Modeling Language "node" notation [Booch et al. 1999].

User site. A computer, its input and output devices, and at least one user.

REFERENCES

ABDEL-WAHAB, H. AND FEIT, M. 1991. XTV: A framework for sharing X window clients in remote synchronous collaboration. In *Proceedings of the IEEE Conference on Communication Software: Communications for Distributed Applications and Systems* (Tricomm '91, Chapel Hill, NC, USA, April) (1991), pp. 159–167.

Abdel-Wahab, H., Kim, O., Kabore, P., and Favreau, J. 1999. Java-based multimedia collaboration and application sharing environment. In *Colloque Francophone sur L'Ingenierie des Protocoles (CFIP'99), Nancy, France* (April 26–29 1999).

Anderson, G., Graham, T., and Wright, T. 2000. Dragonfly: Linking conceptual and implementation architectures of multiuser interactive systems. In *Proceedings of the 22nd International Conference on Software Engineering* (ICSE '00, Limerick, Ireland, June 4–9) (2000). To appear.

Arnold, K. and Gosling, J. 1997. *The Java Programming Language* (second edition ed.). Addison-Wesley. ISBN 0-201-31006-6.

Baecker, R. 1993. *Readings in Groupware and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration.* Morgan Kaufmann Publishers. ISBN 1-55860-241-0.

Banavar, G., Miller, K., and Ward, M. 1998. Adaptive views: Adapting to changing network conditions in optimistic groupware. In *Proc. Euro-PDS '98* (1998).

Bass, L., Clements, P., and Kazman, R. 1998. *Software Architecture in Practice.* SEI Series in Software Engineering. Addison-Wesley. ISBN 0-201-19930-0.

Begole, J., Rosson, M., and Shaffer, C. 1998. Supporting worker independence in collaboration transparency. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST '98) (1998). ACM Press.

Begole, J., Struble, C., Shaffer, C., and Smith, R. 1997. Transparent sharing of Java applets: A replicated approach. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST '97, Banff, Alberta, Canada, Oct. 14–17) (1997), pp. 55–64. ACM Press.

Bentley, R., Rodden, T., Sawyer, P., and Sommerville, I. 1992. An architecture for tailoring cooperative multi-user displays. In J. Turner and R. Kraut Eds., *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '92, Toronto, Canada, Oct. 31–Nov. 4) (1992), pp. 187–194. ACM Press.

Berlage, T. and Genau, A. 1993. A framework for shared applications with replicated architecture. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST '93, Atlanta, GA, USA, Nov. 3–5) (1993). ACM Press.

Bharat, K. and Brown, M. 1994. Building distributed multi-user applications by direct manipulation. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST '94, Marina delRey, CA, USA) (1994), pp. 71–82. ACM Press.

Bharat, K. and Cardelli, L. 1995. Migratory applications. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST '95, Pittsburgh, PA, USA, Nov. 14–17) (1995), pp. 133–142. ACM Press.

Bier, E. and Freeman, S. 1991. MMM: A user interface architecture for shared editors on a single screen. In *Proceedings of the Fourth Annual Symposium on User Interface Software and Technology* (UIST '91, Hilton Head, SC, USA, Nov. 11–13) (1991), pp. 79–86. ACM Press.

Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L. 1987. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering SE-13*, 1 (Jan.), 65–76.

Booch, G., Rumbaugh, J., and Jacobson, I. 1999. *The Unified Modeling Language User Guide.* Object Technology Series. ACM Press/Addison-Wesley. ISBN 0-201-57168-4.

Boyd, J. 1993. Floor control policies in multi-user applications. In *Human Factors in Computing Systems: INTERCHI '93 Conference Proceedings* (Amsterdam, The Netherlands, Apr. 24–29) (1993), pp. 107–108. ACM Press/Addison-Wesley.

Burridge, R. 1999. *Java Shared Data Toolkit User Guide version 2.0.* Sun Microsystems, JavaSoft Division. Available from `http://java.sun.com`.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. 1996. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons Ltd. ISBN 0-471-95869-7.

CALVARY, G., COUTAZ, J., AND NIGAY, L. 1997. From single-user architectural design to PAC*: A generic software architecture model for CSCW. In *Human Factors in Computing Systems: CHI '97 Conference Proceedings* (USA) (1997), pp. 242–249. ACM Press/Addison-Wesley.

CARDELLI, L. 1994. Obliq: A language with distributed scope. Technical Report 122 (March), Digital Equipment Corporation, System Research Center, Palo Alto, CA.

CHABERT, A., GROSSMAN, E., JACKSON, L., PIETROWICZ, S., AND SEGUIN, C. 1998. Java object sharing in Habanero. *Communications of the ACM 41*, 6 (June), 69–76.

CHAN, P. AND LEE, R. 1998. *The Java Class Libraries, Volume 2: java.applet, java.awt, java.beans* (second ed.). The Java Series. Addison-Wesley. ISBN 0-201-31003-1.

CHUNG, G. AND DEWAN, P. 1996. A mechanism for supporting client migration in a shared window system. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST '96, Seattle, WA, USA, Nov. 6–8) (1996). ACM Press.

CHUNG, G., DEWAN, P., AND RAJARAM, S. 1998. Generic and composable latecomer accommodation service for centralized shared systems. In *Proceedings of the IFIP 2.7 Working Conference on Engineering for Human-Computer Interaction (EHCI '98,* Herkalion, Crete, September 14–18) (1998). To appear.

CHUNG, G., JEFFAY, K., AND ABDEL-WAHAB, H. 1993. Accomodating latecomers in shared window systems. *Project Overviews, IEEE Computer 26*, 1 (January), 72–74.

CORMACK, G. 1995a. A calculus for concurrent update. Research report CS-95-06, University of Waterloo. Available from `ftp://cs-archive.uwaterloo.ca`.

CORMACK, G. 1995b. A counterexample to the distributed operational transform and a corrected algorithm for point-to-point communication. Research report CS-95-08, University of Waterloo. Available from `ftp://cs-archive.uwaterloo.ca`.

COULOURIS, G., DOLLIMORE, J., AND KINDBERG, T. 1994. *Distributed Systems: Concepts and Design* (second ed.). Addison-Wesley. ISBN 0-201-62433-8.

COUTAZ, J. 1987a. The construction of user interfaces and the object paradigm. In *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP '87, Paris, France), Published as Lecture Notes in Computer Science, No. 276, Springer-Verlag (1987), pp. 121–130. Springer-Verlag.

COUTAZ, J. 1987b. PAC, an object oriented model for dialog design. In *Proc. INTERACT'87* (1987), pp. 431–436. Elsevier Science Publishers B. V. (North-Holland).

COUTAZ, J. 1997. PAC-ing the architecture of your user interface. In *Proceedings of the DSV-IS'97, Fourth Eurographics Workshop on Design, Specification and Verification of Interactive Systems* (1997), pp. 15–32. Springer Verlag.

COUTAZ, J., BÉRARD, F., CARRAUX, E., AND CROWLEY, J. 1998. Early experience with the mediaspace CoMedi. In *Proceedings of the IFIP 2.7 Working Conference on Engineering for Human-Computer Interaction (EHCI '98,* Herkalion, Crete, September 14–18) (1998). Kluwer Academic Publishers.

CROWLEY, T., MILAZZO, P., BAKER, E., FORSDICK, H., AND TOMLINSON, R. 1990. MMConf: An infrastructure for building shared multimedia applications. In F. HALASZ Ed., *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '90, Los Angeles, CA, USA, Oct. 7–10), (also in [Baecker 1993]) (1990), pp. 329–342. ACM Press.

DAY, M. 1997. What synchronous groupware needs: Notification services. Position paper for the 6th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VI). Also available as Lotus Workgroup Technologies Technical Report 97-02 from `http://research.lotus.com`.

DAY, M., PATTERSON, J., KUCAN, J., CHEE, W., AND MITCHELL, D. 1996. Notification service transfer protocol version 1.0. Technical Report 96-08 (November 15), Lotus Workgroup Technologies. Available from `http://research.lotus.com`.

DEWAN, P. 1993. An editing-based characterization of the design space of collaborative applicaitons. In *Proceedings of the 4th Conference on Organizational Computing, Coordination, and Collaboration* (March 1993).

DEWAN, P. 1995. Multiuser architectures. In *Proceedings of the EHCI '95, IFIP Working Conference on Engineering for Human-Computer Interaction* (1995).

DEWAN, P. 1999. Architectures for collaborative applications. In M. BEAUDOUIN-LAFON Ed., *Computer Supported Co-operative Work*. John Wiley & Sons Ltd. ISBN 0-471-96736-X.

DEWAN, P. AND CHOUDHARY, R. 1991. Primitives for programming multi-user interfaces. In *Proceedings of the Fourth Annual Symposium on User Interface Software and Technology* (UIST '91, Hilton Head, SC, USA, Nov. 11–13), (also in [Baecker 1993]) (1991), pp. 41–48. ACM Press.

DEWAN, P. AND CHOUDHARY, R. 1992. A high-level and flexible framework for implementing multiuser user interfaces. *ACM Transactions on Information Systems 10*, 4 (October), 345–380.

DEWAN, P. AND CHOUDHARY, R. 1995. Coupling the user interfaces of a multiuser program. *ACM Transactions on Computer-Human Interaction 2*, 1 (March), 1–39.

DIX, A. 1996. Challenges and perspectives for cooperative work on the Web. In *Proc. ERCIM Workshop on CSCW and the Web* (1996).

DOURISH, P. 1995. Developing a reflective model of collaborative systems. *ACM Transactions on Computer-Human Interaction 2*, 1 (March), 40–63.

DOURISH, P. 1996a. Consistency guarantees: Exploiting application semantics for consistency management in a collaboration toolkit. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '96, Boston, MA, USA, Nov. 16–20) (1996). ACM Press.

DOURISH, P. 1996b. *Open Implementation and Flexibility in CSCW Toolkits*. Ph. D. thesis, University of London.

DOURISH, P. AND BELLOTTI, V. 1992. Awareness and coordination in shared workspaces. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '92, Toronto, Canada, Oct. 31–Nov. 4) (1992), pp. 107–114. ACM Press.

DUCE, D., GALLOP, J., JOHNSON, I., ROBINSON, K., SEELIG, C., AND COOPER, C. 1998. Distributed collaborative visualization – the MANICORAL approach. In *Proc. Eurographics UK Conference* (March 1998), pp. 69–85.

ECKSTEIN, R., LOY, M., AND WOOD, D. 1998. *Java Swing*. O'Reilly and Associates. ISBN 1-56952-455-X.

ELLIS, C. AND GIBBS, S. 1989. Concurrency control in groupware systems. In *Proceedings of the ACM Conference on the Management of Data* (SIGMOD '89, Seattle, WA, USA, May 2–4) (1989), pp. 399–407. ACM Press.

FREEMAN-BENSON, B. AND BORNING, A. 1992. Constraint imperative programming languages for building interactive systems. In B. MYERS Ed., *Languages for Developing User Interfaces*, Chapter 11, pp. 161–181. Jones & Bartlett Publishers.

GALL, U. AND HAUCK, F. 1997. Promondia: A Java-based framework for real-time group communication on the Web. In *Proceedings of the 6th World Wide Web Conference,* (Santa Clara, CA. April 7–11) (1997). Elsevier Science Publishers B. V. (North-Holland).

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.

GARFINKEL, D., WELTI, B., AND YIP, T. 1994. HP SharedX: A tool for real-time collaboration. *Hewlett-Packard Journal 45*, 2 (April), 23–36.

GRAHAM, T. 1995. *Declarative Development of Interactive Systems*, Volume 243 of *Berichte der GMD*. R. Oldenbourg Verlag.

GRAHAM, T. 1996. The Clock language: Preliminary reference manual. Internal Report.

GRAHAM, T. AND GRUNDY, J. 1998. External requirements of groupware development tools. In *Proceedings of the IFIP 2.7 Working Conference on Engineering for Human-Computer Interaction (EHCI '98,* Herkalion, Crete, September 14–18) (1998). Kluwer Academic Publishers. To appear.

GRAHAM, T., MORTON, C., AND URNES, T. 1996. ClockWorks: Visual programming of component-based software architectures. *Journal of Visual Languages & Computing 7*, 2 (June), 175–196.

Graham, T. and Urnes, T. 1992. Relational views as a model for automatic distributed implementation of multi-user applications. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '92, Toronto, Canada, Oct. 31–Nov. 4) (1992), pp. 59–66. ACM Press.

Graham, T. and Urnes, T. 1996. Linguistic support for the evolutionary design of software architectures. In *Proceedings of the 18th International Conference on Software Engineering* (ICSE 18, Berlin, Germany, Mar. 25–29) (1996), pp. 418–427. IEEE Computer Society Press.

Graham, T., Urnes, T., and Nejabi, R. 1996. Efficient distributed implementation of semi-replicated synchronous groupware. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST '96, Seattle, WA, USA, Nov. 6–8) (1996), pp. 1–10. ACM Press.

Greenberg, S. and Marwood, D. 1994. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '94, Chapel Hill, NC, USA, Oct. 22–26) (1994), pp. 207–217. ACM Press.

Greenberg, S. and Roseman, M. 1999. Groupware toolkits for synchronous work. In M. Beaudouin-Lafon Ed., *Computer Supported Co-operative Work*. John Wiley & Sons Ltd. ISBN 0-471-96736-X.

Greif, I. 1988. *Computer-Supported Cooperative Work: A Book of Readings*. Morgan Kaufmann Publishers. ISBN 0-934613-57-5.

Grundy, J. 1998. Engineering component-based, user-configurable collaborative editing systems. In *Proceedings of the IFIP 2.7 Working Conference on Engineering for Human-Computer Interaction (EHCI '98,* Herkalion, Crete, September 14–18) (Sept. 1998).

Hill, R. 1992. The Abstraction-Link-View paradigm: Using constraints to connect user interfaces to applications. In *Human Factors in Computing Systems: CHI '92 Conference Proceedings* (Monterey, CA, USA, May 3–7) (1992), pp. 335–342. ACM Press.

Hill, R., Brinck, T., Rohall, S., Patterson, J., and Wilner, W. 1994. The *Rendezvous* language and architecture for constructing multi-user applications. *ACM Transactions on Computer-Human Interaction 1*, 2 (June), 81–125.

Johansen, D., van Renesse, R., and Schneider, F. 1995. Operating system support for mobile agents. In *Proceedings of the 5th. IEEE Workshop on Hot Topics in Operating Systems, Orcas Island, WA, USA (4th-5th May, 1995)* (May 1995). IEEE Computer Society Press.

Jul, E., Levy, H., Hutchinson, N., and Black, A. 1988. Fine-grained mobility in the Emerald system. *ACM Transactions on Computers 6*, 1 (Feb.), 109–133.

Karsenty, A. and Beaudouin-Lafon, M. 1993. An algorithm for distributed groupware applications. In *Proc. 13th International Conference on Distributed Computing Systems (ICDCS)* (1993), pp. 195–202.

Karsenty, A., Tronche, C., and Beaudouin-Lafon, M. 1993. GroupDesign: Shared editing in a heterogeneous environment. *Usenix Journal of Computing Systems 6*, 2, 167–195.

Kazman, R., Bass, L., Abowd, G., and Webb, M. 1993. Analyzing the properties of user interface software. Technical Report CMU-CS093-201 (Oct.), Carnegie Mellon University.

Kiczales, G., Lamping, J., Lopes, C., Maeda, C., Mendhekar, A., and Murphy, G. 1997. Open implementation design guilelines. In *Proceedings of the 19th International Conference on Software Engineering* (ICSE '97, Boston, MA, USA, May 19–23) (May 1997). ACM Press.

Kindberg, T. 1996. Notes on concurrency control in groupware. Unpublished note, available from the Project Mushroom pages at `http://www.dcs.qmw.ac.uk`.

Kindberg, T., Coulouris, G., Dollimore, J., and Heikkinen, J. 1996. Sharing objects over the Internet: The Mushroom approach. In *Proceedings of IEEE Global Internet '96* (Mini-conference at GLOBECOM '96, London, England, Nov. 20–21) (1996). IEEE ComSoc.

Kobayashi, M., Shinozaki, M., Sakairi, T., Touma, M., and Daijavad, S. 1998. Collaborative customer services using synchronous Web browser sharing. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '98, Seattle, WA, USA) (1998), pp. 99–108.

Kolon, M. and Goralski, W. 1999. *IP Telephony.* McGraw Hill.

Krasner, G. and Pope, S. 1988. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming 1*, 3 (August/September), 26–49.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (July), 558–565.

Lantz, K. 1986. An experiment in integrated multimedia conferencing. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '86), (also in [Greif 1988], pages 533-552) (December 1986), pp. 267–275.

Lauwers, J., Lantz, K., and Romanow, A. 1990. Replicated architectures for shared window systems: A critique. In *Proceedings of the Conference on Office Information Systems* (ACM COIS '90, Boston, MA, USA, Apr. 25–27)(also in [Baecker 1993]) (1990), pp. 249–260. ACM Press.

Microsoft Corporation. 1997. *Microsoft Netmeeting 2.1 Resource Kit.* Microsoft Corporation. Available from `http://www.microsoft.com/netmeeting`.

Mitchell, A., Posner, I., and Baecker, R. 1995. Learning to write together using groupware. In *Human Factors in Computing Systems: CHI '95 Conference Proceedings* (Denver, CO, USA, May 7–11) (1995), pp. 288–295.

Mitchell, D. 1998. A component approach to embedding awareness and conversation. In *Proc. WETICE'98* (1998). IEEE Computer Society Press. Also available as Lotus Research technical report 98-08 from `http://research.lotus.com`.

Munson, J. and Dewan, P. 1996. A concurrency control framework for collaborative systems. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '96, Boston, MA, USA, Nov. 16–20) (1996), pp. 278–287. ACM Press.

Newman-Wolfe, R., Webb, M., and Montes, M. 1992. Implicit locking in the Ensemble concurrent object-oriented graphics editor. In J. Turner and R. Kraut Eds., *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '92, Toronto, Canada, Oct. 31–Nov. 4) (1992), pp. 265–272. ACM Press.

Nichols, D., Curtis, P., Dixon, M., and Lamping, J. 1995. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST '95, Pittsburgh, PA, USA, Nov. 14–17) (1995), pp. 111–120. ACM Press.

Nigay, L. and Coutaz, J. 1991. Building user interfaces: Organizing software agents. In *Proc. ESPRIT'91 Conference* (1991), pp. 707–719.

O'Grady, T. 1996. Flexible data sharing in a groupware toolkit. Master's thesis, University of Calgary, Calgary, Alberta, Canada.

Ousterhout, J. 1994. *Tcl and the Tk Toolkit.* Addison-Wesley. ISBN 0-201-63337-X.

Patterson, J. 1995. A taxonomy of architectures for synchronous groupware applications. *ACM SIGOIS Bulletin Special Issue: Papers of the CSCW'94 Workshops 15*, 3 (April), 27–29.

Patterson, J., Day, M., and Kucan, J. 1996. Notification servers for synchronous groupware. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '96, Boston, MA, USA, Nov. 16–20) (1996), pp. 122–129. ACM Press.

Perry, D. and Wolf, A. 1992. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes 17*, 4 (October), 40–52.

Pfaff, G. et al. 1985. *User Interface Management Systems.* Eurographics Seminars. Springer Verlag.

Roseman, M. 1998. *GroupKit 5.0 Documentation.* University of Calgary GroupLab. Available from `http://www.cpsc.ucalgary.ca`.

Roseman, M. and Greenberg, S. 1993. Building flexible groupware through open protocols. In *Proceedings of the Conference on Organizational Computing Systems* (ACM COOCS '93, Milpitas, CA, USA, November) (1993). ACM Press.

Roseman, M. and Greenberg, S. 1996. Building real time groupware with GroupKit, a groupware toolkit. *ACM Transactions on Computer-Human Interaction 3*, 1 (March), 66–106.

Roseman, M. and Greenberg, S. 1997. Simplifying component development in an integrated groupware environment. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST '97, Banff, Alberta, Canada, Oct. 14–17) (1997), pp. 65–72. ACM Press.

Roth, J. and Unger:, C. 1998. Dreamteam - a platform for synchronous collaborative applications. In T. Herrmann and K. Just-Hahn Eds., *Groupware und organisatorische Innovation (D-CSCW'98)* (1998), pp. 153–165. B.G. Teubner Stuttgart, Leipzig.

Salber, D. 1995. *De l'interaction individuelle aux systèmes multi-utilisateurs. L'exemple de la Communcation Homme-Homme-Médiatisée.* Ph. D. thesis, l'Universitè Joseph Fournier.

Scheirer, J., Fernandez, R., and Picard, R. 1999. Expression Glasses: a wearable device for facial expression recognition. In *CHI 99 Extended Abstracts* (1999), pp. 262–263.

Schuckmann, C. 1998. Private electronic mail message.

Schuckmann, C., Kirchner, L., Schummer, J., and Haake, J. 1996. Designing object-oriented synchronous groupware with COAST. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '96, Boston, MA, USA, Nov. 16–20) (1996). ACM Press.

Shaw, M. and Garlan, D. 1996. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall. ISBN 0-13-182957-2.

Shen, H. and Dewan, P. 1992. Access control for collaborative environments. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '92, Toronto, Canada, Oct. 31–Nov. 4) (Nov. 1992), pp. 51–58. ACM Press.

Shneiderman, B. 1998. *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (Third ed.). Addison-Wesley. ISBN 0-201-69497-2.

Singh, G. and Green, M. 1991. Automating the lexical and syntactic design of graphical user interfaces: The UofA* UIMS. *ACM Transactions on Graphics 10*, 3 (July), 213–254.

Stefik, M., Bobrow, D., Foster, G., Lanning, S., and Tatar, D. 1987. WYSIWIS revised: Early experiences with multiuser interfaces. *ACM Transactions on Office Information Systems* (also in [Baecker 1993]) *5*, 2, 147–167.

Stephenson, N. 1993. *Snow Crash.* Bantam Spectra. ISBN 0-553-56261-4.

Streitz, N., Geissler, J., Haake, J., and Hol, J. 1994. DOLPHIN: Integrated meeting support across liveboards, local and remote desktop environments. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '94, Chapel Hill, NC, USA, Oct. 22–26) (1994), pp. 345–358. ACM Press/Addison-Wesley.

Strom, R., Banavar, G., Miller, K., Prakash, A., and Ward, M. 1998. Concurrency control and view notification algorithms for collaborative replicated objects. *IEEE Transactions on Computers 47*, 4 (April), 458–471.

Tarpin-Bernard, F. and David, B. 1997. AMF: A new design pattern for complex interactive software. In *Proceedings of the International Conference on Human-Computer Interaction* (HCI '97), San Franciso. Published as *Design of Computing Systems 21 B* (1997). Kluwer Academic Publishers.

Tarpin-Bernard, F., David, B., and Primet, P. 1998. Frameworks and patterns for synchronous groupware: AMF-C approach. In *Proceedings of the IFIP 2.7 Working Conference on Engineering for Human-Computer Interaction (EHCI '98,* Herkalion, Crete, September 14–18) (Sept. 1998).

Taylor, R., Medvidovic, N., Anderson, K., Whitehead Jr., E., Robbins, J., Nies, K., Oreizy, P., and Dubrow, D. 1996. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering 22*, 6 (June), 390–406.

Taylor, R., Nies, K., Bolcer, G., MacFarlane, C., Anderson, K., and Johnson, G. 1995. Chiron-1: A software architecture for user interface development, maintenance, and run-time support. *ACM Transactions on Computer-Human Interaction 2*, 2 (June), 105–144.

ter Hofte, G. 1998. *Working Apart Together: Foundations for Component Groupware.* Number 001 in Telematica Instituut Fundamental Research Series. Telematica Instituut, Enschede, the Netherlands. ISBN 90-75176-14-7. Also available from `http://www.telin.nl`.

UIMS Tool Developers' Workshop. 1992. A metamodel for the runtime architecture of an interactive system. *ACM SIGCHI Bulletin 24*, 1, 32–37.

Urnes, T. 1998. *Efficiently Implementing Synchronous Groupware.* Ph. D. thesis, York University, Toronto, Ontario, Canada.

Urnes, T. and Graham, T. 1999. Flexibly mapping synchronous groupware architectures to distributed implementations. In *Proceedings of the DSV-IS'99, Sixth Eurographics Workshop on Design, Specification and Verification of Interactive Systems* (1999), pp. 133–148.

Urnes, T. and Nejabi, R. 1994. Tools for implementing groupware: Survey and evaluation. Technical Report CS-94-03 (May), York University, Canada.

Vertegaal, R. 1999. The GAZE groupware system: Mediating joint attention in multiparty communication and collaboration. In *Human Factors in Computing Systems: CHI '99 Conference Proceedings* (Pittsburgh, PA, USA, May 15–20) (1999), pp. 294–301.

Viegas, F. and Donath, J. 1999. Chat circles. In *Human Factors in Computing Systems: CHI '99 Conference Proceedings* (Pittsburgh, PA, USA, May 15–20) (1999), pp. 9–16.

Wright, T. 1999. Hierarchical adaptive concurrency control for synchronous groupware applications. Master's thesis, Queen's University, Kingston, Ontario, Canada.

Zweben, S., Edwards, S., Weide, B., and Hollingsworth, J. 1995. The effects of layering and encapsulation on software development cost and quality. *IEEE Transactions on Software Engineering 21*, 3 (March), 200–208.