Contents

9

Architectures for Collaborative Applications

PRASUN DEWAN
University of North Carolina

ABSTRACT

The architecture of a collaborative application is characterised by the modules, layers, replicas, threads, and processes into which the application is decomposed; the awareness in these components of collaboration functions; and the interaction among these components. It influences the function, fairness, fault tolerance, ease of modification, and performance of the application, the amount of programming effort required to implement the application, and the reuse of existing single-user code. We present here a design space of existing and potential collaboration architectures and discuss the consequences of choosing different points in this space.

9.1 INTRODUCTION

The architecture of a software application characterises the components of the application, the function implemented by each component, and the interaction among these components [SG96, KBAW94]. It is an important issue in the design of the application since it influences the performance, ease of modification, and other properties desired by users and programmers of the application. It is also a difficult issue to resolve since decomposing a large problem into smaller parts is a challenging task: There are a number of different ways in which this decomposition can be done, and the consequences of choosing different decompositions are not always apparent.

For these reasons, a new discipline of computer science has emerged to help programmers choose architectures for software applications [SG96]. The architectural techniques developed so far either apply to general software applications or are tied to specific functionality such

Collaborative Systems, Edited by M. Beaudouin-Lafon

as database management [SG96] and user-interface support [SG96, Bas93]. We address here the domain of collaborative applications by describing the influence of collaboration support on the architecture of an application.

We consider five kinds of components of a collaborative application: modules, layers, threads, processes, and replicas. These components occur in both collaborative and non-collaborative applications but the collaboration domain introduces special techniques for decomposing an application into these components. We do not identify the exact functionality of these components, since it depends on application semantics. Instead, we simply classify them according to whether or not they implement collaboration-specific functionality. Similarly, we do not identify the exact events communicated among these components. Instead, we distinguish among them based only on whether or not they carry collaboration-specific information.

We characterise the design space of collaboration architectures by presenting a generic architecture that captures properties common to the points in this design space, and a set of dimensions that represent the differences among these points. We identify different choices along each of these dimensions and evaluate these choices by discussing their influence on properties desired by programmers/users. We consider several generic properties such as ease-of-modification and performance that have been identified by previous work on software architectures. In addition, we consider the special case of reuse of existing single-user code, an important goal in the design of collaborative applications.

To better explain the scope of this work, it would be useful to identify what we are not addressing here. We are not considering the functionality of a collaborative system, which is covered in [DCS94, OMKO93] and the accompanying discussion on shared editors by Prakash. Moreover, we are not describing tools/infrastructures for implementing collaborative applications, some of which are surveyed in the accompanying discussion on toolkits by Greenberg and infrastructures by Dourish. We envision the process of developing a collaborative application to consist of three main steps: (1) design the functionality, (2) decompose the application into components, and (3) use tools for implementing the components. We are looking at only step (2) of this process. Naturally, these steps are not independent. For instance, the choice of the architecture may depend on the functionality desired, and a tool is typically tied to a particular architecture. We will look at these relationships but will not examine in-depth the functionality and tools issues, *per se*. A preliminary discussion of these concepts was presented at a conference [Dew95].

The remainder of this discussion is organized as follows. We first present a model of collaboration that defines the kind of collaborative applications we consider here. Next we describe the generic collaboration architecture for implementing these applications. We then present the various dimensions along which collaboration architectures differ and discuss the tradeoffs to be made in choosing different points along these dimensions. We use these dimensions to classify architectures supported by several existing collaboration tools, and distill our discussion about the tradeoffs by giving a set of architectural design rules that should be followed when implementing collaborative applications, which are in the spirit of those given in [SG96] for user-interface support. Finally, we present conclusions and directions for future work.

9.2 COLLABORATION MODEL

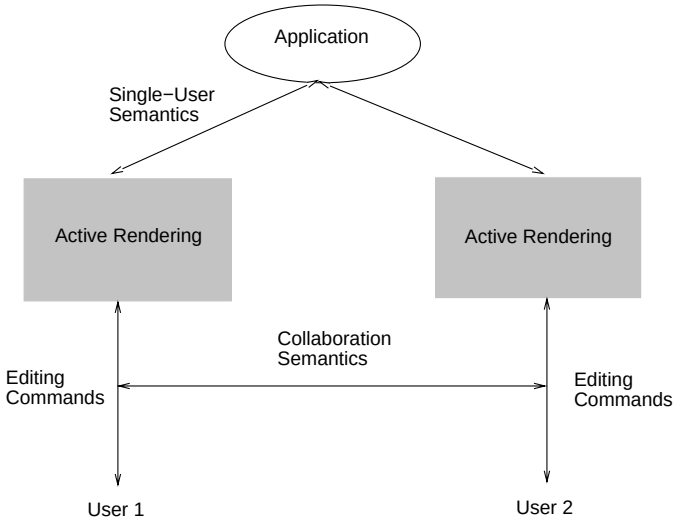

Figure 9.1 Editing-based collaboration model.

To identify architectures of collaborative applications, we first need a model of collaboration that characterises the functionality supported by these applications. We use a collaboration model based on the notion of generalized editing [DS90, DC92, DCS94]. Figure 9.1 illustrates the model.

According to this model, an application can be considered an editor of semantic objects defined by it. A user interacts with the application by editing a rendering [1] of these objects using text/graphics/multimedia editing commands. Thus, interaction with an interactive application is similar to interaction with a text or graphics editor. The difference is that a rendering is "active", that is, changes to it can trigger computations in the application; and conversely, it can be modified in response to computations invoked by the application.

As shown in Figure 9.1, each user perceives a different rendering of the semantic objects. However, the actions of the users are not isolated - they are linked by the application to facilitate and control collaboration among them. For the purposes of this discussion, we will divide the semantics of a collaborative application into *single-user semantics*, which define the feedback users receive in response to commands entered by them or actions taken by the application autonomously (in response to internal state changes or messages from other applications); and *collaboration semantics*, which define the feedback users receive in response to commands entered by others.

This is a simple but general model of collaboration. It models the single-user semantics of a variety of contemporary single-user and collaborative applications. A text/graphics editor can be considered an editor of a text/graphics file; a language-oriented editor can be considered an editor of a program syntax tree; a spreadsheet can be considered an editor of a matrix that responds to an editing of an entry in the matrix by updating related entries; and a debugger can be considered an editor of a debugging history that responds to the insertion of a new command in the history by computing the command and appending the output to the history.

It also models the collaboration semantics of a variety of contemporary collaborative appli-

[1] On the suggestion of one of the referees, we use the term "rendering" here instead of "display" so that we include non- textual/graphical presentations of objects such as audio/video renderings of data.

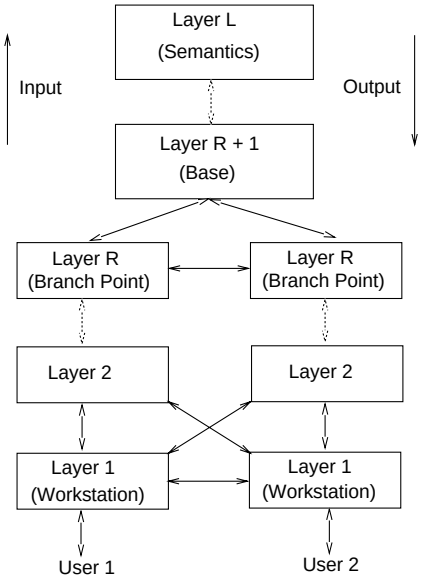

Figure 9.2 Generic architecture.

cations. A "same-time" ("different-time") application is an editor that links (does not link) its renderings in real-time; a "same-place" ("different-place") application is an editor that creates (does not create) all renderings at the same site; a WYSIWIS [2] (non-WYSIWIS) application is an editor that ensures that the renderings are identical (different); and a workflow application is an editor that responds to editing commands by initiating the next step in the workflow.

Thus, at some level of abstraction, any collaborative application can be considered a generalized editor. [3]

9.3 GENERIC ARCHITECTURE

Figure 9.2 shows a generic collaboration architecture for implementing the model described above. It is a generalization of the architecture Patterson proposed at the CSCW'94 workshop on "distributed systems, multimedia and infrastructure support in CSCW" [Dew94] that makes fewer assumptions about collaborative applications. As we shall see later, this architecture can be instantiated to multiple specific architectures.

The architecture assumes that a user's input/output is processed by a hierarchy of layers. A lower-level layer (that is, a layer closer to the user) manages objects that are *interactors* of objects in the immediately higher-level layer. We will refer to the latter as *abstractions* of the former. An interactor of an abstraction creates a *presentation* of the abstraction, which contains a transformation of the information in the abstraction (e.g. a text field representing an integer, or a bitmap representing a text field) plus some additional information serving as "syntactic sugar" (e.g. a label field or a window scrollbar). Thus, perceptible renderings of

[2] What You See Is What I See [SFB+ 87]

[3] We refer to generalized editors that perform editing commands without computing additional application-specific side effects as simply *editors*. These applications are addressed in depth in the accompanying discussion on collaborative editors by Prakash.

abstractions are created by applying the presentation operator successively to their interactors, and the interactors of these interactors, and so on. An abstraction can have a variable number of interactors, which may change dynamically as users create or delete renderings of the abstraction.

The layers communicate with each other using *events*. Often, this term implies that the communication is sent asynchronously by the sender to the receiver. However, we will use it here in a more general sense and allow the information to be retrieved synchronously from the sender by the receiver. We divide events of a collaboration application into *interaction events* and *collaboration events* based on whether they support single-user or collaboration semantics. An interaction event may be an *output event* or an *input event* depending on whether it is sent to a lower or upper-level layer.

Abstractions send output events to their interactors and receive input events from the latter. Output events received by objects from their abstractions may be transformed into lower-level events before they are sent to their interactors. Conversely, input events received by objects from their interactors may be transformed into higher-level events before they are sent to their abstractions. Not all input events received by interactors need to be sent to their abstractions - in particular, events that request manipulation of local syntactic sugar. Moreover, not all output events transmitted down by interactors are triggered by output events received from their abstractions. These include not only those events that change local syntactic sugar but also those that generate local echo/feedback in response to requests for changing the higher-level state in the abstraction.

A collaboration event may be a copy or extension of an interaction event or it may be an entirely new kind of event. It may be sent not only to a lower-level and upper-level layer but also a cross layer, that is a layer in an another branch, as shown in the figure.

Some levels in this architecture are *shared* while others are *versioned* or *replicated*. A shared level is associated with a single, shared layer that processes the input/output of multiple users of the application, while a versioned or replicated level is associated with a private layer for each user of the application, which processes the input/output of only that user and collaboration events concerning the user. An object in a private layer is private while an object in a shared layer is shared by multiple users. We refer to the collection of all private objects of a user and the shared objects accessible to the user as the *interaction state* of that user. All levels below a private level are constrained to be private levels and all levels above a shared level are constrained to be shared levels. Thus, the architecture defines a tree of layers rather than a general graph. We refer to this tree as a *protocol tree* in analogy with the related networking concept of a protocol stack. We refer to the lowest shared layer as the *base*, the highest versioned layers as *branch points*, the base and all layers above it as the *stem*, and a branch point and all the layers below it as a *branch* of the architecture. Moreover, we refer to all private layers at a certain level as *peers* or *replicas* of each other.

An abstraction may have interactors in zero or more replicated layers. We refer to the different interactors of an abstraction as replicas, peers, or versions. In general, they can create different logical presentations of the abstraction. However, in most current collaboration architectures, they create different physical replicas (for different users) of the same logical presentation. It is for this reason, we have used the term “replica” for a peer interactor and layer, though strictly speaking, the term “version” is more general. In the rest of the discussion, we will use these terms interchangeably. It is important to note that an interactor in a layer may not have a peer interactor in a peer layer, since not every layer creates an interactor for an abstraction in the layer above.

Abstractions and interactors may not only transform interaction events but also control the interaction by checking access rights, consistency, and other constraints. Unlike the Smalltalk Model-View-Controller paradigm [KP88] but like the abstraction-view paradigms supported by InterViews [LVC89], Rendezvous [HBR+94], PAC [Cou87], and several other frameworks, we do not treat the transformation and control components as separate objects. Similarly, unlike the Clover model [Sal95], we do not differentiate among the different collaboration functions implemented by an abstraction or interactor, clubbing them all in one multi-function object. Furthermore, unlike the PAC model, we do not capture the structure of a hierarchical abstraction or interactor, modelling it as a single unit. We do not assume that an abstraction or interactor is actually implemented as a programming language object. Similarly, we do not assume that an architectural event is actually implemented as a programming event. It may be sent in response to the evaluation of a programming constraint or some other higher-level computation that is not explicitly aware of events. Programming issues are beyond the scope of this discussion since we are focusing here only on architectural issues.

The bottom-most layers in this architecture are the workstation (operating system and hardware) layers managing the screen and input devices attached to a workstation. The workstation layers are usually replicated to allow the collaborators to use different workstations. A notable exception is MMM [BF91], which allows a single workstation layer to be shared by multiple users concurrently manipulating the same screen using different input devices. We refer to the topmost layer in the architecture as the *semantic layer* and the abstractions in this layer as *semantic objects*. Unlike a lower-level object, a semantic object is not itself an interactor for another object. However, like an interactor, a semantic object in a replicated layer may have peers or replicas in peer layers. Peer semantic objects are (the highest-level) computer representations of the same user-level abstract object.

Not all application modules are layered in the protocol tree shown in the figure. We refer to such modules as *external modules*. The layers and modules in a collaboration architecture include both *application components* implemented by the application programmer, and *system components* provided by an infrastructure or tool. When characterizing the "architecture" of a collaboration tool, we will, in fact, be characterizing those aspects of the architectures of clients of the tool that are defined by the tool. An individual client may refine this architecture by adding further layers and modules.

9.4 DESIGN SPACE

The generic architecture given above defines a design space of collaboration architectures that differ in the way they resolve several important issues:

- *Single-User Architecture*: What is the architecture for implementing single-user semantics?
- *Concurrency*: Which components of the application can execute concurrently?
- *Distribution*: Which of these components can execute on separate hosts?
- *Versioning/Replication*: Which of these components are replicated?
- *Collaboration Awareness*: Which of these components are collaboration aware, that is, implement collaboration semantics?

In the following sections, we discuss these issues in-depth. We do not give specific answers to these questions, since they would depend on particulars of the application. Instead, we present

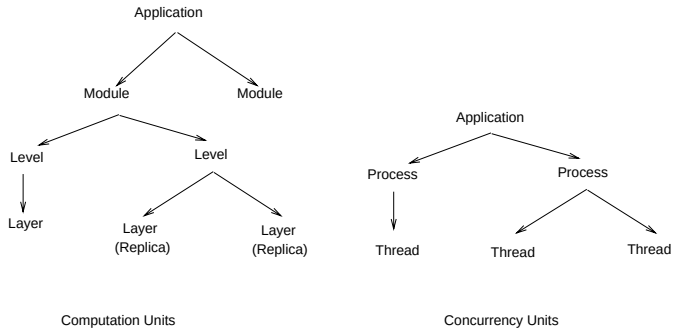

Figure 9.3 Decomposing an application by computation and concurrency units.

general constraints or approaches for resolving these issues and discuss the consequences of using these approaches.

In our discussion, we consider two decompositions of a collaborative application: by computation and concurrency unit. The first one, used in figure 9.2, assumes that an application is divided into one or communicating modules, a module may be composed of one or more levels, and each level consists of one or more replicated layers. In the rest of the discussion, we shall often use the term level and layer interchangeably, especially when a level consists of a single layer. The second one assumes that an application is decomposed into one or more distributable processes, and each process forks one or more concurrent threads. The difference between a process and a thread is that the former is a heavyweight unit of concurrency, associated with its own address space, which can be created on different hosts. In contrast, all threads within a process share a common address space and host, though they may execute on different processors on the host. Figure 9.3 shows the two decompositions of an application. As shown later, these two kinds of decompositions are not independent in the architectures we present below.

9.4.1 SINGLE-USER ARCHITECTURE

The single-user architecture or *basis* of a multiuser architecture describes those aspects of the latter that implement single-user semantics. In this discussion, of course, we are concerned mainly with those aspects that influence/are influenced by collaboration semantics. We consider single-user architectures here because the design of the collaborative aspects is often dependent on the basis.

Strictly speaking, the basis is a view of a collaboration architecture that may not have an independent existence. In practice, however, collaboration architectures are designed by extending existing single-user architectures. A large variety of single-user architectures have been devised in the context of single-user user-interface software. We will focus here only on those that we know have formed the bases of existing collaboration architectures. Our architectural descriptions are a set of assumptions regarding the nature of a single-user architecture. Thus, they apply to a family of architectures rather than a specific architecture.

The most general architecture is one that makes no assumption about the nature or number of application layers. By making no assumptions about an application, we can cover arbitrary applications, but cannot reason about any of them. The architecture of these applications can be described as a single level of Figure 9.2 that contains arbitrary abstractions/ interactors.

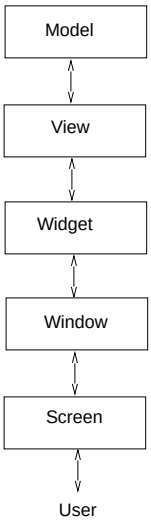

Figure 9.4 Common user interface layers.

It is possible to subclass this architecture in several ways depending on the assumptions we make about the kinds of user-interface layers used. Four main kinds of general layers have been identified so far: window, widget, view, and model [Mye95, KP88], which would have increasing levels in a layered architecture that includes them (Figure 9.4). An architecture that supports one of these levels does not necessarily have all the potential levels below it. For instance, a view layer may be implemented directly on top of the workstation without defining a widget layer. We can distinguish among these architectures by defining a layering degree, L, which gives the number of software layers in the architecture. For instance, Team Workstation [IO90] has a layering degree of 2, since it assumes workstation and application layers; XTV [AWJ94], Rapport [EAHL88], Shared X [GWY94], and MMConf [CF89] have layering degrees of 3, since they assume an additional window layer; GroupKit [RG96] has a layering degree of 4, since it assumes an additional widget layer; and Suite [DC92], Weasel [GU92], and Clock [GUN96] have a layering degree of 5, since they assume an additional view layer. The application layers in all cases may be further subdivided into other layers. The layering and other degrees we associate with a tool (infrastructure) give the minimum degrees of client applications that use the tool. As we shall see later, the layering degree of an architecture bounds its awareness, replication, concurrency, and distribution degrees.

It is possible to further specialize these architectures by classifying them according to the specific instances of the abstract layers used in their implementation. For instance, an early version of GroupKit was based on InterViews widgets [LVC89] while the current one is based on Tk widgets [Ous94]. However, we will not distinguish among these specific instances, since from the architectural point of view, these differences are not important.

9.4.2 COLLABORATION AWARENESS

We discuss now different approaches to transforming a single-user layering to a multiuser one.

One approach is to keep the exact same set of layers and add collaboration functionality to one or more of these layers. This approach is used in many existing architectures including

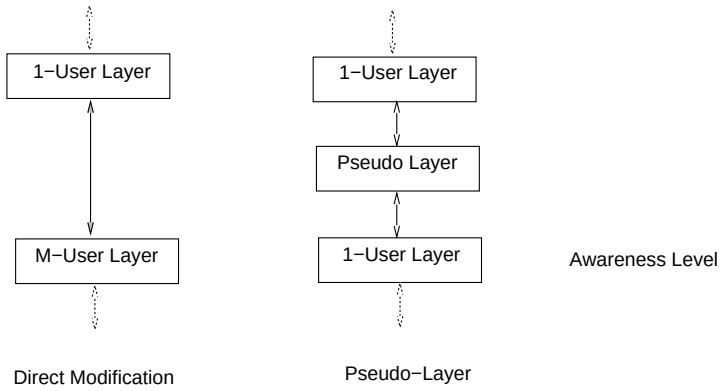

Figure 9.5 Modifying a 1-user layer vs adding a new pseudo-layer.

Shared X, which extends the X window server, and Suite, which extends the view layer. However, it supports limited reuse of existing software since it requires changes to the layers that are made collaboration-aware. Moreover, all implementations of a layer must be changed even if they provide the same interface. It also supports limited modifiability in that a single-layer implements both the single-user and collaboration semantics. (These problems may be reduced, but not eliminated, if these layers are coded in an object-oriented programming language, since the changes may be localized in high-level classes and automatically inherited by unchanged lower-level classes.) Finally, it is not viable if the source code of the layer to be changed is not available.

Another approach is to put a *pseudo-layer* between two existing layers of the single-user architecture. To each of these two layers, the pseudo-layer provides an extension of the interface the other one provided. As a result, it accepts all of the input and output events sent to it by the layers below and above it, respectively. Depending on the nature of the interface between the two existing layers, the addition of the pseudo-layer may require recompiling and/or relinking of the existing layers. However, unlike other approaches, it does not require changes to the original layers. Moreover, it allows the same pseudo-layer to be added between multiple implementations of the two layers, as long as these implementations provide the same interface. It also supports increased modifiability since a pseudo-layer does not have to be changed in response to changes in the implementations of the original single-user layers. This approach is supported in XTV, which inserts a pseudo-layer between an X server and client, and COLA [TRM94] and DistView [PS94], which add pseudo-layers at higher-levels.

The pseudo-layer approach has two main drawbacks: First, all communication between the two layers of the original architecture must now pass through an extra layer, which may reside in a separate address space. For instance, in XTV, all communication between an X server and client must pass through a pseudo X server. Second, a pseudo-layer may need to duplicate the data structures and code of the original layers. For instance, an X pseudo-server that allows only certain windows to be shared must recreate the window tree hierarchy maintained by the X server.

Adding a pseudo-layer does not change the layering degree of the architecture, since the layer is not a “real” layer in that it does not transform its input or output. A pseudo-layer can be considered as logically belonging to the next lower layer, and should be replicated, threaded, or distributed with this layer.

Which levels of the architecture should be made collaboration-aware, that is, at which levels must collaboration awareness be added to existing layers or new pseudo-layers introduced? One approach is to localize these modules at a single level. Assuming this approach is used, we need to choose the collaboration-aware level. There are several advantages of choosing a lower-level. First, a lower-level is typically common to a larger number of applications. [4] For instance, the X window system is used by both Suite and non-Suite applications, while the Suite view layer is used only by the subset of X applications that are Suite applications. As a result, adding collaboration-awareness at a lower-level typically provides collaboration support for a larger number of applications, since it is available not only to direct clients of the layer but also clients of higher-level layers implemented on top of this layer. Second, a lower-level layer can give users earlier feedback than higher-level layers. In general, there is a delay between the time information is received by a lower-level layer and the time it is transmitted to a higher-level layer. For instance, a widget layer may transmit edits to a form item to the higher-level layer only when the user completes the item. Hence, the lower-level a collaboration-aware layer is, the earlier it can distribute a user's edits to others and point out access and concurrency control violations. Earlier feedback allows users to collaborate more synchronously and reduces the amount of work that may have to be undone. Finally, under this approach higher-level layers are not required to process interaction events from lower-level layers (see below), which makes them more modular and portable since they are dependent on handling fewer kinds of events from lower levels.

On the other hand, there are two important, related advantages of adding collaboration support at higher-levels. First, coupling, locking, access control, and other collaboration functions can operate on units that are more meaningful to the user/programmer. For instance, unlike a window layer, a view layer can separately lock the different views displayed in a window. Second, a higher collaboration-aware level can, if it is replicated, typically, provide more degrees of sharing among peers at that level. To explain why, we make the following two observations. The sharing of peer interactor objects implies the sharing of the next-level abstraction objects, assuming that abstractions are kept consistent with their interactors. However, the sharing of an abstraction does not imply sharing of its interactors, since the peer interactors may transform the shared abstraction in different ways and add different kinds of syntactic sugar. Thus, a collaboration-aware layer can allow (a) no sharing between peer abstractions, (b) sharing of peer abstractions without sharing of lower-level interactors, and (c) sharing of lower-level interactors if appropriate input events can be solicited from the lower-level layers. For instance, Suite can allow (a) no sharing between peer views, (b) sharing of peer views without sharing of the windows displaying them, and (c) sharing of peer windows by soliciting all X events. In contrast, a lower collaboration-aware level cannot allow sharing of higher-level abstractions without sharing of their interactors at this level.

In the higher-level case, sharing of lower-level interactors is achieved, at the cost of increasing the *interaction awareness* in the higher-level layer, that is, the awareness of interaction events of lower-level layers. For instance, to allow sharing of lower-level interactors such as widows, multiuser Suite is forced to handle several low-level X events such as window movement and resize events, which single-user Suite was unaware of.

We associate a collaboration architecture with an *awareness degree*, which is the level of the highest layer that is collaboration-aware. The value of this degree ranges from 1 in

[4] This is not always the case since a higher-level layer might be ported to multiple lower-level layers.

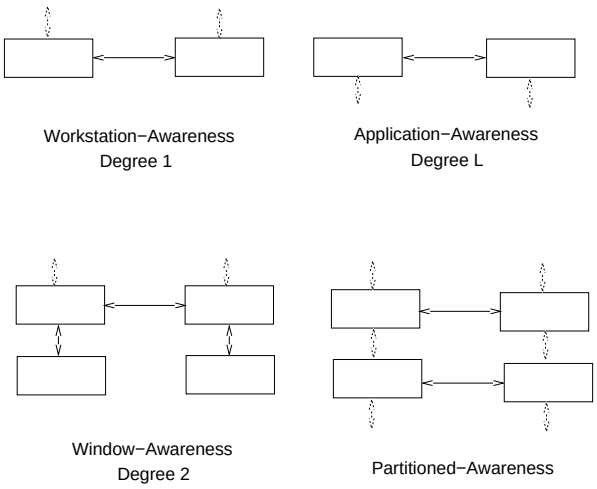

Figure 9.6 Approaches to collaboration awareness.

Team Workstation, which provides all collaboration support at the workstation level; to 5 in DistView, which requires the model layer to be collaboration-aware.

Since there are benefits of adding collaboration-awareness at both lower and upper levels, it is useful to consider an approach that partitions this awareness among multiple layers. Such an approach could offer the benefits of both the lower-level and higher-level approaches. In particular, it can offer logical collaboration units, flexible sharing, and low interaction awareness. However, unlike the localized approach, this approach would require providers of multiple modules to address collaboration, coordinate their activities, and often implement similar functionality (such as remote invocation) multiple times. This approach is offered in MMConf by making both the window and application layers collaboration-aware, and in Suite, by allowing both the view and application layers to be collaboration-aware. Figure 9.6 illustrates the various approaches to collaboration awareness.

9.4.3 VERSIONING/REPLICATION

The versioning/replication architectural dimension determines the base and branch points in the generic architecture of Figure 9.2. As mentioned before, all layers below a base are replicated. We can thus associate an architecture with a *replication degree*, which is the level of the branch point. Two extreme approaches to replication are the *centralized* and *replicated* approaches. The former creates no replicated level while the latter creates no base level. In between these two approaches, several *semi-replicated* approaches are possible, which choose different levels for the base layer. Thus, the replication degree of an architecture with L levels is in the range 0 to L (Figure 9.7).

There are important advantages of choosing a higher replication degree. As we shall see later, the replication degree of an architecture bounds its distribution and concurrency degrees. Thus, a higher replication degrees allows more distribution and concurrency benefits (discussed later). Moreover, a higher replication degree allows more divergence in the interaction states of the users since fewer levels are shared. For instance, if the view level is shared, then all users are constrained to see the same views of models. A higher-degree of replication allows

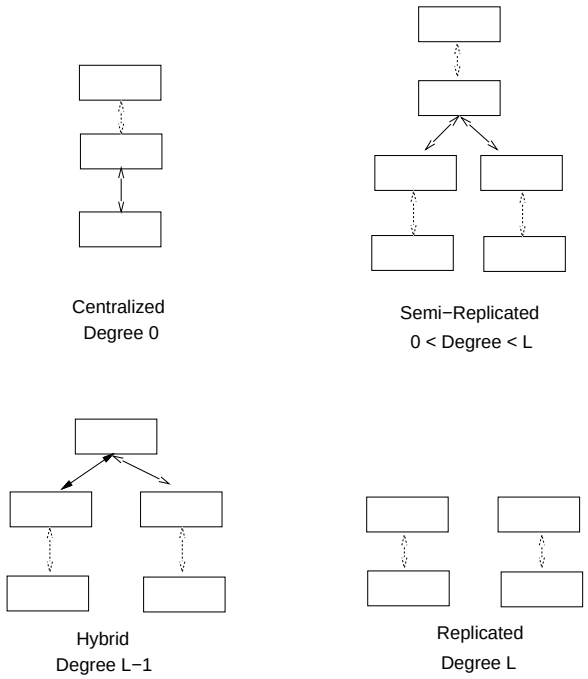

Figure 9.7 Replication approaches.

but does not force more divergence since it is possible for peer objects to share state via collaboration events, as mentioned in the previous section.

On the other hand, replicating a level requires a mechanism for keeping the peer layers at that level consistent. If these layers are meant to be exact replicas, then often a tool can automatically provide this mechanism. Automatic consistency among the objects in exact replicas is typically achieved by executing the same set of operations on these objects. For instance, if a user presses a button widget, then this operation is also invoked on all peers of the widget that are meant to be exact replicas. However, multiple invocations of an operation lead to several problems:

- *Inefficiency*: They can lead to serious efficiency problems if the operation is an expensive one.
- *Access bottleneck*: They may try to, simultaneously, access a central resource (such as a file) at the same time, thereby causing an access bottleneck.
- *Incorrect writes*: They may modify the same central resource, thereby causing the same value to be written multiple times. The access and write problems would be eliminated in a system that replicated all resources.
- *Incorrect side effects*: They may send mail, print documents, and perform other side effects multiple times.

The last two problems can be averted in collaboration-aware peer layers that ensure (based, for instance, on user identities) that only one of these layers performs the write and other side effects.

Not all layers perform operations with one or more of these properties. Typically, it is

the topmost layer - the one containing semantic objects - that performs such operations. Therefore, several systems adopt a special case of the semi-replicated architecture that keeps the semantic layer centralized and the lower-level layers replicated. We refer to this architecture as the *hybrid* architecture. Given an application with L levels, the replication degree of the hybrid architecture for this application is L-1.

Since replication has both important advantages and disadvantages, there is substantial variation in the replication degrees of collaborative applications. Another cause for this variation is the variation in the level of the collaboration tools used for automatically implementing replication. A collaboration tool can either replicate all or none of the layers in its client. Since there are important disadvantages of replicating the topmost layer, typically the tool will replicate its layers but not those of its client. As a result, tools at different levels will offer different replication degrees.

Systems supporting the hybrid architecture include Rendezvous, Suite, Weasel and Clock Systems that offer full replication include GroupKit and GroupDesign [KTBL93], while the only one we know that offers pure centralization is MMM. Team Workstation supports a replication degree of 1. A window-based architecture such as XTV and Rapport that centralizes its client has been traditionally called a *centralized architecture* [LL90]. However, under our terminology, it is a semi-replicated architecture with degree 2, since the workstation and window layers are replicated. Because of the replication degree supported by them, MMM/Team Workstation/XTV/Suite cannot allow screen/windows/views/models to diverge. GroupKit and GroupDesign allow all of these layers to diverge, but require collaboration awareness to solve the problems with invoking the same operation on multiple replicas.

9.4.4 CONCURRENCY

Decomposing an application into multiple threads is important in single-user applications since it allows these threads to execute simultaneously on a multiprocessor system. It is particularly important in multimodal applications where the devices for different I/O modes such as audio, video, mouse, and keyboard can be managed by different threads. The multiuser case offers additional opportunities and reasons for creating multiple threads. Typically, the users of a collaborative application can input and output data concurrently. Thus, the different branches created for these users are potential concurrency units that can be executed simultaneously by different processors of a multiprocessor system. Even in a single-processor system, creating separate threads for these branches is important. It supports fair (preemptive) scheduling among these threads by ensuring that a computation triggered in a branch by the actions of a user does not lock out other users for an unbounded time.

However, there are reasons why a complete branch may not be associated with its own thread. The system support needed to create threads may not be available to programmers. Moreover, programmers may not be willing to put the effort required to create and synchronize threads. A collaboration tool can automate this task for the layers it knows about but not those in its clients. Similarly, it may not be possible to assign a thread to a layer without requiring changes to the layer since the syntax and semantics of an invocation in the same or different thread may be different. Thus, the goal of increasing the concurrency may conflict with the goal of reuse since the former may require changes to source code of an existing layer.

As a result, different architectures may take different approaches to concurrency depending on how they tradeoff the benefits of concurrency with its drawbacks. To capture differences among these architectures, we associate them with a *concurrency degree*, which is a measure

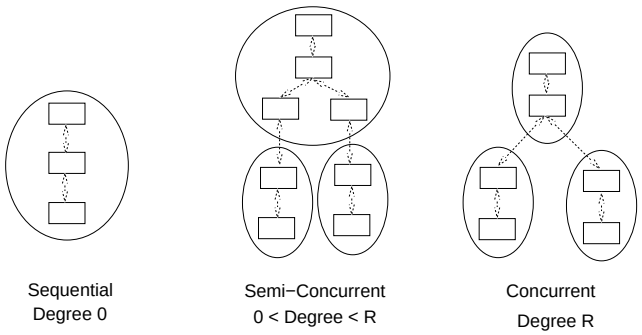

Figure 9.8 Concurrency approaches. The rectangular boxes are layers and the ellipses are threads.

of how many layers in a branch execute in their own thread. An architecture has concurrency degree, C, if no layer at or below level C shares a thread with a stem layer or a layer in a different branch. Different layers in a branch may, and typically do, share a common thread. The concurrency degree of a collaboration architecture ranges from 0 to R, where R is its replication degree. We refer to architectures with concurrency degree 0 and R as *sequential* and *concurrent* architectures, respectively, and the remaining architectures as *semi-concurrent* architectures (Figure 9.8) . A sequential architecture must be a centralized architecture. In a non-centralized architecture, the workstation level is guaranteed to be replicated. A replicated workstation level (but not other levels) must be distributed, by definition, since a level is distributed if it resides on multiple workstations. Furthermore, we assume that distributed layers execute concurrently. Hence no non-centralized architecture is sequential.

All collaboration tools known to us offer the concurrent approach. Of course, the replication degrees in these systems may be different, as mentioned before, which causes variations in the concurrency offered by them. For instance, the concurrency degree in Rendezvous and Suite is 4 and in XTV it is 2. In all existing replicated architectures it is the same as the layering – and hence replication – degree.

The above discussion identifies a simple approach to introducing concurrency in a collaborative application: assign all branch layers below some level C to a separate thread. A concurrent architecture created using this approach does not necessarily offer the maximum possible concurrency, which would require an approach that identifies all portions of the application that could potentially execute concurrently and assigns each of these to a separate thread. We refer to such an approach as the *maximal-concurrent* approach. This approach is highly application dependent and either requires the programmer to identify the threads, which has proven to be a tedious, error-prone, and difficult task in general, or the system to automatically perform this task, which in general is impossible. Unlike the maximal-concurrent approach, our approach does not process concurrently the actions in a branch or stem invoked by a single-user (such as concurrent mouse and key clicks by the same user), or the actions in the stem invoked by different users (such as concurrent key clicks by two independent users handled by a central layer). However, it does allow the computation of the local feedback in the branches of different users to be performed concurrently.

Note that the notion of the concurrency degree applies to all collaboration architectures including those that assign threads based on approaches other than the one we have given above. However, it does not capture all concurrency differences among these architectures.

For instance, as mentioned above, a concurrent architecture may or may not be a maximal concurrent architecture.

9.4.5 DISTRIBUTION

Once the threads of an application have been identified, they must be assigned to process address spaces, which in turn must then be assigned to hosts. Assigning different threads to multiple address spaces increases fault tolerance since fatal errors in one thread do not necessarily cause the whole application to fail. This is particularly important in the multiuser case, since users would like to be protected from the errors of others. If the replicas created for different users are assigned to different address spaces, then a fatal error in one replica would not necessarily cause the other replicas to crash.

Distributing different processes to different hosts also allows an address space to be close to the resources it is accessing the most. Again, this is particularly important in the multiuser case, since the replicas created for different users need to access different and possibly widely separated workstations. By executing replicated layers on a local workstation, no remote communication is required to generate the local feedback computed by these layers. Moreover, events transmitted from these workstations are high-level events generated by the local layers rather than low-level events generated by the workstation. Typically, a higher-level I/O event contains less data and is communicated less frequently than a lower-level one, and thus generates less traffic on the network. For instance, communicating committed changes to an integer value communicates less data than communicating incremental changes to a slider representation of it.

On the other hand, distributing portions of an application on different workstations is not without drawbacks. The distributed parts of the application are not guaranteed to see the same environment, which can cause problems. For instance, problems would occur if the application uses a file name that is not valid at all sites unless the application is site-aware. Moreover, synchronizing distributed replicas is a difficult problem. Often an event received by a layer must also be sent to remote replicas to satisfy consistency constraints among them. To ensure good response times for the local users, such events must be processed immediately by the local layers without trying to ensure a global ordering among them. As a result, the distributed replicas may get inconsistent unless application-specific techniques [EG89] are used to transform or abort received events, or the events are guaranteed to commute.

As a result, different architectures take different approaches to distribution depending on how they tradeoff its communication benefits with its drawbacks. To capture differences among these architectures, we associate an architecture with a *distribution degree*, which is analogous to its concurrency degree. It is a measure of how many layers in a branch can execute on the local host. An architecture has distribution degree, D, if no layer at or below level D shares an address space with a stem layer or a layer in a different branch. Different layers in a branch may, and typically do, share a common address space. The concurrency degree of a system is always higher than its distribution degree since distributed modules execute concurrently. However, it is not the same, since a particular address space can execute multiple threads concurrently. Thus, the distribution degree of a collaboration architecture ranges from 0 to C, where C is its concurrency degree. We refer to architectures with distribution degree 0 and C as *single-site* and *distributed* architectures, respectively, and the remaining architectures as *semi-distributed* architectures (Figure 9.9). A single-site architecture must be a sequential architecture since distributed modules execute concurrently. Like the maximal-concurrent

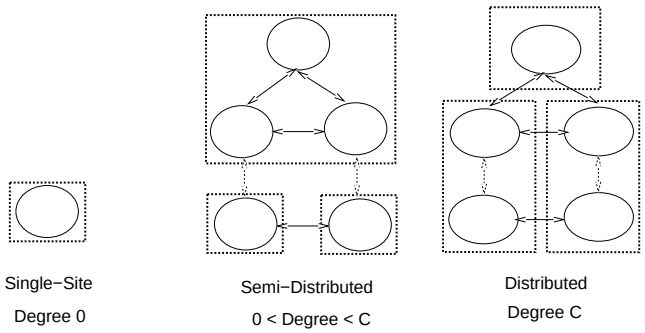

Figure 9.9 Distribution approaches. Ellipses are threads and rectangles are hosts.

approach, it is possible to imagine a *maximal-distributed* approach that dynamically assigns each application module to the workstation accessing it the 'most'. However, such an approach [JLHB88] is still a subject of research and requires application-specific support. Our notion of a distribution degree does not distinguish between those distributed architecture that offer maximal distribution and those that do not.

The distributed approach determines only how the application is decomposed into processes and not how these processes are assigned to hosts. Depending on the workstation and network speed, it may, in fact, be sometimes beneficial to execute branch layers on a fast remote workstation. The higher the distribution degree of an architecture, the more the flexibility in reducing the communication costs.

Not all communication costs go down when a replica is executed on a local host. In particular, the cost of communicating with remote higher-level and peer layers goes up. However, assuming that information gets abstracted as it flows upwards and that a collaboration or input event received by a layer triggers a lower-level output event, the overall communication cost is reduced. To better understand the logic behind this conclusion, consider Figure 9.10, which shows the difference between placing replicas, A and A', on local and central hosts. Consider how an input IA, to layer A, is processed by the various layers in the architecture. Layer A produces some local feedback, OAL, sends a collaboration event, CA, to its peer, and an input event, IB, to the higher-level layer. The higher-level layer, in turn, produces feedback TOB (which is the total feedback consisting of feedback of B and all of the layers above), which, in turn, is transformed to TOA by layer A. On receiving CA, layer A' produces coupling feedback CAO, and sends an input event IB' to B'. Layer B', in turn, produces total feedback TOB', which, in turn, is transformed by A' to TOA'.

Consider the local and central placement schemes shown in Figure 9.10. The difference between them is in the placement of the replicas - under local placement, replicas A and A' are placed on the local workstations, while under central placement, they are placed on the central site. In the local case, events IA, TOA, CAO, TOA', OAL are transmitted locally, and events IB, TOB, CA, TOB', and IB' are transmitted across the network, while in the central case, the converse is true. If we assume that information gets condensed when it is processed by a higher-level layer, then the following relationships hold among the size of these events: IA > IB, TOA > TOB, CAO > CA, TOA' > TOB'. Also, if we assume that a higher-level event triggered by an input event is smaller than any lower-level event also triggered by the same input, then OAL > IB'. These relationships imply that more information is transmitted locally in the first case.

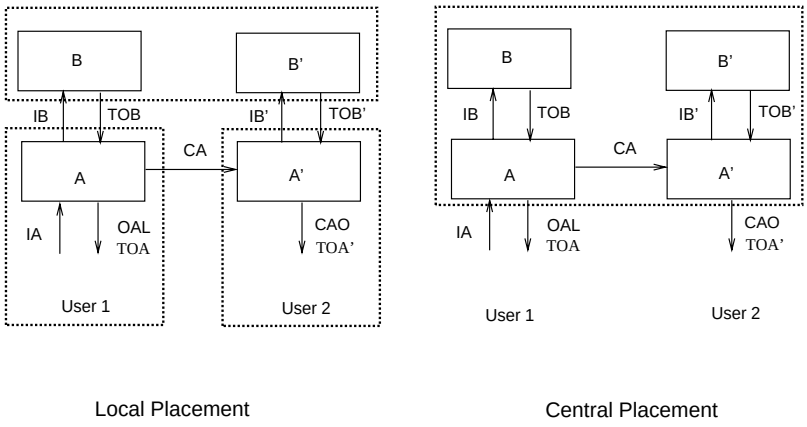

Figure 9.10 Local and central placement of replicated layers.

We have ignored, above, peer collaboration events sent to B' and other layers above A'. In both cases, such events will be communicated locally. However, under local placement, the resulting output sent to the remote user will be higher level - the output of B' rather than A' - thereby further reducing the communication cost.

We have also ignored collaboration events sent to cross layers. For similar reasons, these events also favour local placement of modules.

Most existing architectures offer the distributed approach, that is, distribute all of their concurrent threads. A notable exception is the Rendezvous architecture, which offers a distribution of degree of 2 but a concurrency degree of 4. In this architecture, all layers except the X window layers execute at a central site. However, at the central site, the layers in different branches execute in separate threads. The Clock system provides a hybrid approach, allowing the same application program to have degrees 2 to 4, depending on whether it centralizes the replicated widget and view layers.

In our discussion above, we have assumed that every collaboration event sent to a peer layer results in an output event. This may not hold true for constraint-based systems such as Rendezvous, Weasel, and Clock, which may need several collaboration events to be exchanged before the constraint evaluation can fire the output events. It is perhaps for this reason that Rendezvous does not use a distributed architecture, though preliminary performance results from Weasel and Clock show advantages of using such an architecture even in a constraint-based environment.

9.5 EXTERNAL MODULES

Not all collaboration modules can be added to existing single-user layers or new pseudo-layers. It may also be necessary to create new external modules that do not belong in the protocol tree, for several reasons (Figure 9.11):

- *Session Management*: In a collaborative system, session management modules are needed to create/delete the protocol tree of an application when a session with an application is started/terminated. In the single-user case, the operating system is responsible for creating/deleting interactive sessions, but in the multiuser case, special, possibly application-

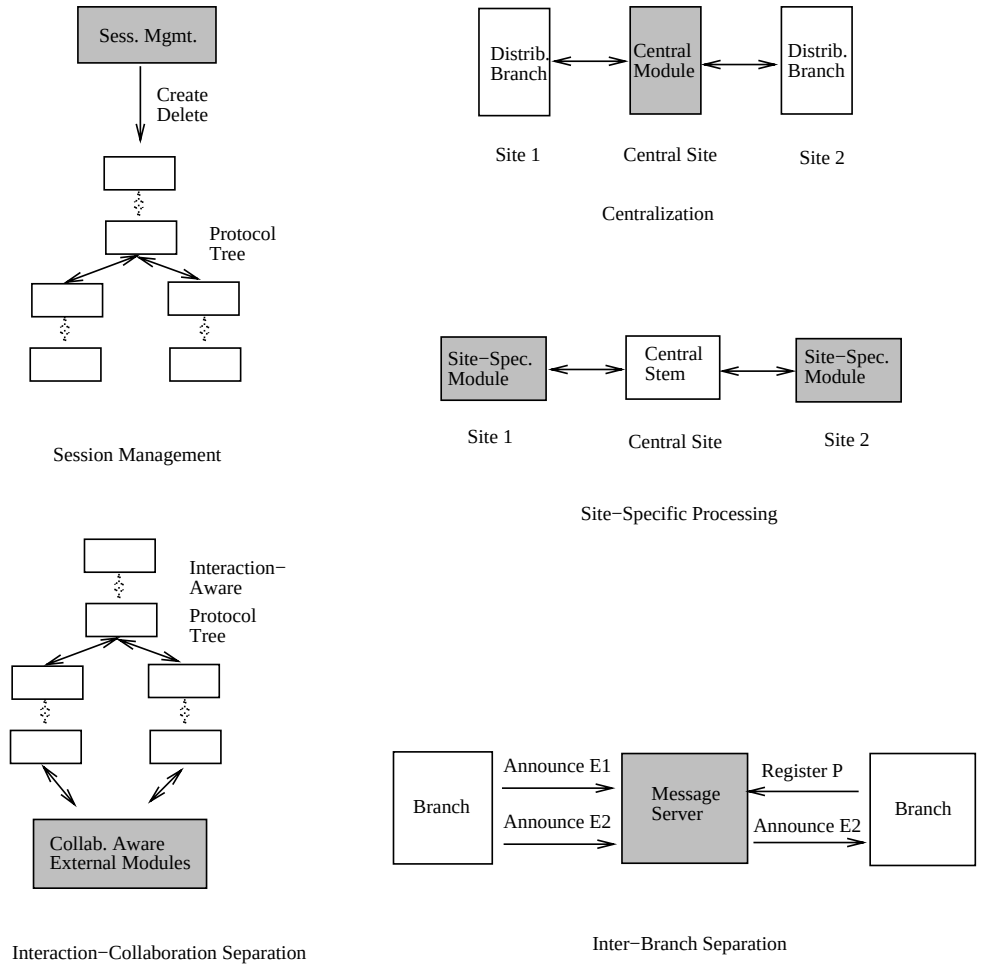

Figure 9.11 Reasons for adding external modules.

specific, protocols are necessary for session management [RG96]. Since these protocols create/delete protocol trees, they cannot be implemented within the tree itself, and thus must be provided by external modules.

- *Centralization*: Replicated collaboration-aware layers may need to communicate with central modules to keep central resources such as locks or ensure global ordering of messages communicated among these layers. These modules can be implemented in the stem of the protocol tree, which is the approach taken in Suite. However, this approach cannot be taken if the architecture is fully replicated or if it cannot have any collaboration-aware stem layers. In these cases, the central modules must be external to the protocol tree.
- *Site-Specific Processing*: Centralized collaboration-aware layers may need to communicate with modules that must be located at a particular site for efficiency or other reasons. Examples of such modules are those that access files, devices, or processes at a particular site or keep information about the active users or sessions at a site. These modules can be implemented in collaboration-aware branch layers at that site if such layers exist; otherwise

they must be modules external to the protocol tree. Systems that distribute replicas create a site-specific server for creating and terminating processes at that site. Similarly, Suite creates an audio server at each site to access the audio devices at that site [RMS+93].

- *Collaboration and Interaction Independence*: For modularity reasons, it may be desirable to separate the processing of interaction and collaboration events. As mentioned before, pseudo-layers can be used to increase this separation since such layers are responsible only for transmitting interaction events and not for transforming them. However, these layers have the performance disadvantages mentioned before and support limited separation since they must process both kinds of events. Similarly, within a layer, encapsulation may be used to separate the interaction-aware and collaboration-aware objects. An approach providing more separation is to process collaboration events in external modules, which can be shared by multiple layers and branches.
- *Inter Branch Independence:* It is also useful to reduce the awareness a branch has about branches created for other users. Reduced inter-branch awareness increases the modularity of the system, and more important, reduces the cost of connecting a branch to other branches. If every branch kept track of every other branch it may need to communicate with, then branch awareness and interaction awareness would be implemented by the same layers, and more important, a branch would need to be informed each time a new branch is created that may need to communicate with it. It may be more attractive to implement one or more (possibly replicated) external message servers [Rei90], responsible for linking the replicated branches. A message server receives message *patterns* from information clients indicating the kind of messages they are interested in receiving, and *announcements* from information servers announcing events in which information clients may be interested. The message server forwards an announcement from an information server to all information clients who have registered an interest in the announcement. This is essentially the approach taken in [BMT89]. A message server leads to more modularity and reduced connection cost, but increases the "hop count" of inter-branch messages, that is, increases the number of modules responsible for processing inter-branch messages. The increased hop count is a serious problem if the message server is centralized and the branches are distributed, since the message server can become a central bottleneck. On the other hand, as mentioned before, a central agent such may be necessary in any case to implement global ordering of distributed operations.

In many of the cases above, we have not defined the specifics of how the external modules are connected to each other, threaded, distributed, or replicated. These issues can be resolved in the same way they were resolved for the original modules. In fact, it is possible to create a hierarchy of replicated, distributed, concurrent external modules. For instance, GroupKit creates a central registrar that acts as a connection point and name server, with replicated local session managers at all sites deciding the policy for how people enter into groupware sessions.

9.6 RULES

Ideally, we would like to identify universal principles that should be followed in the design of all collaboration architectures. However, as explained in the sections above, there are no absolute rules in the design of these architectures. Therefore, what we offer, instead, is a set of qualified rules summarizing the advantages/disadvantages of different architectural approaches. These

can be used by developers of an application/tool to optimize the set of properties that are important for that application/tool.

Layering: A higher degree of layering can support higher degrees of awareness and replication.

Awareness: A higher degree of awareness leads to more flexible sharing and higher-level units of collaboration, but supports less reuse, delays feedback, and increases interaction awareness (if the partitioned approach is not taken).

Replication: A higher degree of replication supports more divergence and a higher degree of concurrency; but requires more layers to be kept consistent, and results in inefficiency, access bottlenecks, and incorrect writes and other side effects.

Concurrency: A higher degree of concurrency increases fairness, performance, and the maximum degree of distribution; but reduces reuse, requires special system support, and increases programming overhead.

Distribution: A higher degree of distribution increases fault tolerance and reduces communication costs, but introduces problems of synchronization and heterogeneity.

Partitioning: Partitioned collaboration awareness reduces interaction awareness; but requires more programming effort and supports less reuse.

Pseudo-Layer: The pseudo-layer approach supports more reuse and modularity; but offers less performance and can result in duplication of effort.

External Modules: External modules are necessary for supporting session management, centralization, site-specific processing, collaboration and interaction independence, and inter branch independence; but increase the complexity of the system and can reduce performance.

9.7 CLASSIFYING EXISTING SYSTEMS

Table 9.1 Layering and associated degrees of existing architectures.

System	Layers	Lyr. D.	Awr. D.	Rep. D.	Conc. D.	Dist. D.
MMM [BF91]	app/workst	2	2	0	0	0
Team Workstation [IO90]	app/workst	2	1	1	1	1
XTV [AWJ94]	app/win/workst	3	2	2	2	2
Shared X [GWY94]	app/win/workst	3	2	2	2	2
MMConf [CF89]	app/win/workst	3	3	3	3	3
GroupKit [RG96]	app/wid/win/workst	4	4	4	4	4
Rendezvous [HBR+94]	app/view/wid/win/workst	5	5	4	4	2
Suite [DC92]	app/view/wid/win/workst	5	5	4	4	4
Weasel [GU92]	app/view/wid/win/workst	5	5	4	4	4
Clock [GUN96]	app/view/wid/win/workst	5	5	4	4	2-4
DistView [PS94]	app/view/wid/win/workst	5	5	5	5	5

Table 9.1 and 9.2 describe architectures of several existing collaboration systems. Table 9.1 gives the layering and associated degrees supported by them. Since all of these systems are collaboration tools, these values refer to the minimum values of these degrees, since some clients may create additional layers, replicas, processes, and threads in the application. We have assumed above that all view layers are build on top of widget layers so that a comparison

of the various degrees is more meaningful. Table 9.2 indicates the other properties supported by them: pseudo-modules, partitioned awareness, and external modules to support session management, centralization, site-specific computing, collaboration awareness, and message servers. These tables show the similarities and differences among these tools. All systems except MMM support multi-workstation collaboration. Among these systems, Team Workstation is workstation-based; XTV, Shared X, and MMConf are window-based; and Rendezvous, Suite, Weasel, and DistView are view-based. MMM offers the pure centralized architecture; MMConf, GroupKit, and DistView the replicated architecture; and Team Workstation, XTV, Rendezvous, Weasel, and DistView the semi-replicated architecture. In all systems except Rendezvous, the distribution degree is the same as concurrency degree. From an architectural point of view, there are no differences between Suite and Weasel.

9.8 CONCLUSIONS AND FUTURE WORK

This work makes several contributions. It motivates the need for studying software architectures of collaborative systems, describes a generic architecture that encapsulates architectural properties common to a wide range of collaborative systems, identifies a set of issues that a designer of a specific architecture must face, discusses and evaluates competing approaches to addressing these issues, classifies existing systems according to the approaches they have taken, and gives a set of architectural rules.

This work is related to the SAAM model for describing architectures of software systems [KBAW94]. This model advocates (i) a canonical decomposition of the functionality of the system, (ii) identification of the structure of the system, that is, the set of components of the system and the communication among these components, (iii) identification of the functions performed by each component, (iv) selection of a set of abstract properties for evaluating the architecture, (v) selection of a set of concrete tasks that have these properties, and (vi) evaluation of the extent to which the architecture supports the abstract properties and concrete tasks. Our work has applied several of these steps. In particular, it has applied step (i) by decomposing the functionality of a collaborative application into interaction functions and collaboration functions, (ii) by identifying the layers, threads, and processes of a collaborative system, (iii) by distinguishing between collaboration-aware and unaware layers, (iv) by selecting functionality, performance, programming effort, reusability, and modularity as evaluation properties, and (vi) by evaluating how well each of these abstract properties are satisfied by an architecture. It would be useful to extend this work by (a) identifying concrete tasks that have the evaluation properties and evaluating how well the architecture supports these tasks, (b) doing a finer structural decomposition that identifies the components of the layers and the external modules of the architecture, and (c) doing a finer task assignment that distinguishes among layers based not only on whether they perform interaction or collaboration functions but also on the set of collaboration functions they perform.

The framework and associated terminology can be used for understanding, comparing, and classifying existing collaboration systems. It can also be used to varying degrees to design new systems. One method would be to take the set of approaches supported in an existing system to develop a new system that addresses details not covered here differently. For instance, the set of approaches used in XTV can be used to develop a shared window system based on a different network single-user window system such as the Plan 9 window system [PPTT90]. A more novel use of the framework would be to choose a new combination of the set of

the approaches described here. For instance, a new version of Suite can be developed that supports a fully replicated architecture. This framework makes these tasks easier by telling the designers which questions they have to answer what choices are available, and what the consequences of these choices are.

This work can be extended in many other ways. It would be useful to decompose a layer by structure, as in the PAC model, and function, as in the Clover model. A first-cut at combining this architecture with PAC and Clover has been published recently [CCN97]. It is also necessary to identify other assumptions, issues, approaches, and criteria for comparing architectures. In particular, it useful to relax the assumption that all levels above a central level are also centralized. In a single-workstation collaborative system such as MMM, it may be useful to create different branches for different users. Moreover, in such a system, it would be useful to capture, in the concurrency degree, the notion of assigning different devices to different threads. This architecture was developed based on experiences with implementing multiuser textual/graphical user-interfaces. It would be useful to test its applicability for multiuser audio/video and 3-D virtual reality user-interfaces. It may also be useful to relax the assumption that a layer is replicated/threaded/distributed as a whole, which does not apply to Shastra [AB93]. In Shastra, the semantic layer consists of two parts: one performs expensive computations while the other performs relatively inexpensive ones. The expensive part is centralized but the inexpensive one is replicated and distributed since computation costs dominate in one case and communication costs in the other. It would also be useful to consider migration and caching of centralized components of collaborative applications [GUN96, CD96] and their impact on performance.

ACKNOWLEDGMENTS

I am grateful for the in-depth comments of the referees. This research was supported in part by National Science Foundation Grants IRI-9408708, IRI-9508514, IRI-9627619, and CDA-9624662, and DARPA/ONR Grant N 66001-96-C-8507.

Table 9.2 Pseudo layers, partioned awareness, and different kinds of external modules.

System	Pseudo	Part.	Sess. M.	Central	Site-Spec.	Colab. Awr.	Msg. Serv.
MMM	N	Y	N	N	N	N	N
Team Workstation	N	N	Y	N	N	N	N
XTV	Y	N	Y	N	Y	N	N
Shared X	N	N	Y	N	N	N	N
MMConf	N	Y	Y	N	Y	N	N
GroupKit	N	Y	Y	N	Y	N	N
Rendezvous	N	Y	Y	N	N	N	N
Suite	N	Y	Y	N	Y	N	N
Weasel	N	Y	Y	N	N	N	N
Clock	N	Y	Y	N	N	N	N
DistView	Y	Y	Y	N	N	N	N

References

[AB93] V. Anupam and C. Bajaj. Collaborative multimedia scientific design in. *Proceedings of ACM Conference on Multimedia*, pages 447–456, 1993.

[AWJ94] Hussein Abdel-Wahab and Kevin Jeffay. Issues, problems and solutions in sharing clients on multiple displays. *Internetworking: Research and Experience*, 5:1–15, 94.

[Bas93] Len Bass. Architectures for interactive software system: Rationale and design. *Trends in Software: Issue on User Interface Software*, 1:31–44, 1993.

[BF91] Eric A. Bier and Steve Freeman. Mmm: A user interface architecture for shared editors on a single screen. *Proceedings of the 4th ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 79–87, November 1991.

[BMT89] A. Bonfiglio, G. Malatesta, and F. Tisato. Conference toolkit: Framework for real-time conferencing. *ECSCW'89*, pages 303–316, 89.

[CCN97] Gaelle Calvary, Joelle Coutaz, and Laurence Nigay. From single-user architectural design to pac*: a generic software architecture model for cscw. In *CHI Proceedings'97*, pages 242–249, March 1997.

[CD96] Goopeel Chung and Prasun Dewan. A mechanism for supporting client migration in a shared window system. In *Proceedings of the Ninth Conference on User Interface Software and Technology*, pages 11–20, October 1996.

[CF89] T. Crowley and H. Forsdick. Mmconf: The diamond multimedia conferencing system. *Proceedings of the IFIP WG8.4 Groupware Technology Workshop*, August 1989.

[Cou87] J. Coutaz. Pac, an object oriented model for dialog design. In *Proceedings Interact'87*, pages 431–436. North Holland, 1987.

[DC92] Prasun Dewan and Rajiv Choudhary. A high-level and flexible framework for implementing multi-user user interfaces. *ACM Transactions on Information Systems*, 10(4):345–380, October 1992.

[DCS94] Prasun Dewan, Rajiv Choudhary, and HongHai Shen. An editing-based characterization of the design space of collaborative applications. *Journal of Organizational Computing*, 4(3):219–240, 1994.

[Dew94] Prasun Dewan. Cscw'94 workshops. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 1–4, October 1994.

[Dew95] Prasun Dewan. Multiuser architectures. *Proc. of IFIP WG2.7 Working Conference on Engineering for Human-Computer Communication*, pages 43–70, August 1995.

[DS90] Prasun Dewan and Marvin Solomon. An approach to support automatic generation of user interfaces. *ACM Transactions on Programming Languages and Systems*, 12(4):566–609, October 1990.

[EAHL88] J.R. Ensor, S.R. Ahuja, D.N. Horn, and S.E. Lucco. The rapport multimedia conferencing system: A software overview. *Proceedings of the 2nd IEEE Conference on Computer Workstations*, pages 52–58, March 1988.

[EG89] C.A. Ellis and S.J. Gibbs. Concurrency control in groupware systems. *Proceedings of the ACM SIGMOD '89 Conference in Groupware Systems*, May, 1989.

[GU92] T. C. Nicholas Graham and Tore Urnes. Relational views as a model for automatic distributed implementation of multi-user applications. *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 59–66, November 1992.

[GUN96] T.C. Nicholas Graham, Tore Urnes, and Roy Nejabi. Efficient distributed implementation

of semi-replicated synchronous groupware. In *Proceedings of the Ninth Conference on User Interface Software and Technology*, pages 1–10, October 1996.

[GWY94] Daniel Garfinkel, Bruce Welti, and Thomas Yip. Shared x: A tool for real-time collaboration. *Hewlett-Packard Journal*, pages 23–24, April 1994.

[HBR+94] Ralph Hill, Tom Brinck, Steven Rohall, John Patterson, and Wayne Wilner. The rendezvous architecture and language for constructing multiuser applications. *ACM Transactions on Computer Human Interaction*, 1(2), June 1994.

[IO90] Hirsoshi Ishii and Masaaki Ohkubo. Design of a team workstation. *Multi-User Interfaces and Applications*, pages 131–142, 1990.

[JLHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM TOCS*, 1988.

[KBAW94] Rick Kazman, Len Bass, Gregory Abowd, and Mike Webb. Saam: A method for analyzing the properties of software architectures. *Proc. of International Conference on Software Engineering*, pages 81–90, May 1994.

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.

[KTBL93] Alain Karsenty, Christophe Tronche, and Michel Beaudouin-Lafon. Groupdesign: Shared editing in a heterogeneous environment. *Usenix Computing Systems*, 6(2):167–195, Spring 1993.

[LL90] J.C. Lauwers and K.A. Lantz. Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems. *Proceedings of ACM CHI'90*, pages 303–312, April 1990.

[LVC89] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with interviews. *IEEE Computer*, pages 8–24, February 1989.

[Mye95] Brad Myers. User interface software tools. *ACM Transactions on Computer-Human Interaction*, 2(1):64–103, March 1995.

[OMKO93] Gary M. Olson, Lola J. McGuffin, Eiji Kuwana, and Judith S. Olson. Designing software for a group's needs: A functional analysis of synchronous groupware. *Trends in Software: Special Issue on User Interface Software*, 1:129–148, 1993.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Mass., 1994.

[PPTT90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from bell labs. *Proc. of the Summer UKUUG Conf.*, pages 1–9, July 1990.

[PS94] Atul Prakash and Hyong Sop Shim. Distview: Support for building efficient collaborative applications using replicated active objects. *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 153–162, October 1994.

[Rei90] Steven P. Reiss. Connecting tools using message passing in the field environment. *IEEE Software*, 7(4):57–66, July 1990.

[RG96] Mark Roseman and Saul Greenberg. Building real-time groupware with groupkit, groupware toolkit. *ACM Transactions on Computer-Human Interaction*, 3(1):66–106, 1996.

[RMS+93] J. Riedl, V. Mashayekhi, J. Schnepf, M. Claypool, and D. Frankowski. Suitesound: System for distributed collaborative multimedia. *IEEE Transactions on Knowledge and Data Engineering*, 5(4):600–609, August 1993.

[Sal95] D. Salber. De'interaction individuelle aux systemes multi-utilisateurs.'exemple de la communication homme-homme-mediatisee. September 1995.

[SFB+87] Mark Stefik, Gregg Foster, Daniel G. Bobrow, Kenneth Kahn, Stan Lanning, and Lucy Suchman. Beyond the chalkboard: Computer support for collaboration and problem solving in meetings. *CACM*, 30(1):32–47, January 1987.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, New Jersey, 1996.

[TRM94] Jonathon Trevor, Tom Rodden, and John Mariani. The use of adapters to support cooperative sharing. *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 219–230, October 1994.