A UNIX Toolkit for Distributed Synchronous Collaborative Applications

Dorab Patel and Scott D. Kalter Twin Sun, Inc.

ABSTRACT: There are many low-level problems, such as resource discovery and rendezvous, faced by developers of distributed synchronous collaborative applications. This paper systematically explores these problems and discusses their solutions under UNIX. These solutions are collected into a toolkit that provides a high-level abstract interface to developers for a variety of different application classes. The toolkit supports rendezvous via a file, rather than via a user or application. This allows clients to join a session without additional user specification. An evaluation of the toolkit, and comparison with alternatives, indicates the class of applications most suited to this approach. Experience using the toolkit with various applications demonstrates the usefulness of the provided primitives for rapidly developing collaborative applications.

1. Introduction

"Groupware" means different things to different people. These days, any software not specifically meant for use by a single user seems to be covered under that term. One reasonable definition is provided by Ellis et al. in [1].

> . . . computer-based systems that support groups of people engaged in a *common task* and that provide an interface to a *shared environment*.

Even under this definition, there are many kinds of groupware, ranging from asynchronous systems like email and databases, to highly synchronous systems like shared editors and conferencing systems. This paper concentrates on the class of groupware called synchronous distributed groupware.

Many common low-level problems, like resource discovery and rendezvous, must be solved by groupware implementors on each implementation platform before the collaborative functionality of the application can be developed. Developers must work around the limitations of the operating system to implement solutions to these problems. Each developer usually builds this functionality from scratch and goes through the same learning process. We have abstracted our solutions into a flexible, robust, and application-independent toolkit for building collaborative applications.

The paper starts off by defining synchronous distributed groupware and its tradeoffs. This is followed by a section describing our requirements. The next section describes a series of collaborative issues that must be addressed by developers of synchronous groupware and the challenges faced in implementing solutions under UNIX. The following sections gather these solutions into a framework and describe the high-level abstract interface provided to the toolkit users. A subsequent section describes experiences with the toolkit which highlights the appli-

cability of the COeX primitives to a variety of application domains. A comparison with alternative approaches and a summary conclude the paper.

2. *Distributed Groupware Architectures*

Groupware architectures [2] can be divided into those with centralized state and those with distributed state (Figure 1). Centralized state systems (e.g., XTV [3]) store the shared state at a single location, but allow participants to have multiple views of this shared state. All modifications to the state occur at the central server host. This eases the synchronization and state consistency problems that occur when multiple users modify the state simultaneously. However, the central location becomes a bottleneck for traffic and causes a higher latency for operations because of the round trip time between the central site and the host requesting the operation. Distributed state architectures avoid the latency, robustness, and bottleneck problems of the centralized state approach but introduce other problems. Distributed state architectures (e.g., MMConf [4]) typically consist of replicated application instances that each store a local representation of the shared state. Each instance of the application may be running on different hardware platforms under different operating systems. Changes in state values are distributed to the other instances of the application by user action.

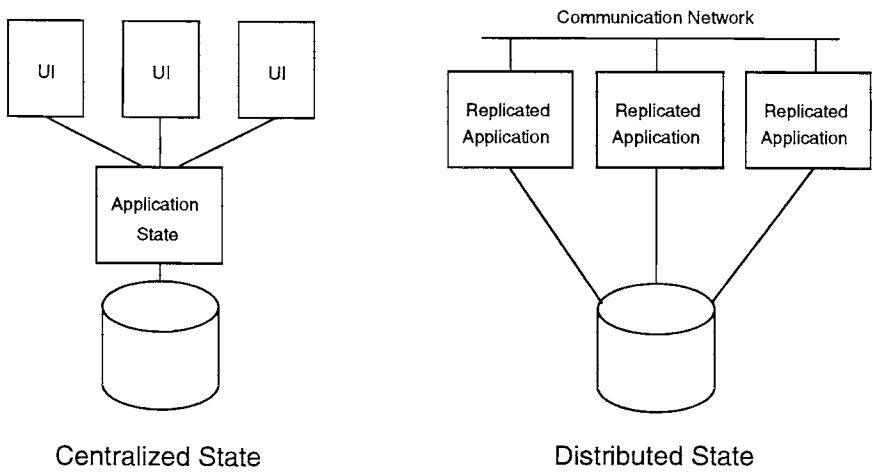

Figure 1: Centralized versus distributed state

Thus, the "state" of the shared data is replicated across the buffers of the application instances. The challenge is to make sure that all these replicated buffers are consistent with each other. This is usually done by serializing updates to the state or using a consistency control algorithm [5, 6].

Just as the state can be either centralized or distributed, so can views. Having all users share the same view results in a WYSIWIS (What You See Is What I See) view architecture [7, 8]. This approach is simple to implement and can provide a focus for group interaction since the whole group shares the same view. This can be especially useful in a teaching or demonstration environment. On the other hand, being forced to share the same view can be rigid and limiting in situations where browsing the data is more important. A strict WYSIWIS system can be relaxed across different dimensions. An extreme relaxing leads to a system in which each user has an independent view of the shared data. These approaches can be combined to form hybrid architectures. Suite [9] is a system which allows a dynamically variable amount of coupling among views and data.

Asynchronous groupware deals with systems facilitating collaboration among users who are interacting over a period of time. Synchronous groupware supports real-time collaboration among users who are operating simultaneously on their shared data. Our paper focuses on synchronous collaboration situations using a replicated state architecture providing independent views.

3. Requirements

In addition to providing a toolkit for building synchronous distributed collaborative applications, we wanted to support the following requirements:

- Provide support for sharing application data, not necessarily the views provided to each client. This makes the toolkit independent of the window system and user interface toolkits used by the application.
- Cover a wide variety of applications, including text and graphics.
- In order to ease the transition from single-user applications to collaborative ones, a collaborative application in single-user mode must behave the same as a single-user application.

- Provide high-level support for sharing application-level objects.
- Each client must be able to modify parts of the application data concurrently. This implies that floor-passing is not an adequate sharing policy.
- Provide communication and sharing mechanisms, while minimizing policies imposed on the developer.
- The toolkit must be compatible with applications written in different languages and function across various UNIX platforms.

4. *Issues*

There are many issues faced by developers of collaborative applications. Their discussion is ordered chronologically according to their occurrence in the sharing process.

4.1 Resource Discovery Models

The first problem in collaboration is resource discovery—finding out what resources are available for sharing, and how to control that availability. For our purposes, the shared resources are UNIX files that can be edited simultaneously.

Some collaboration systems are based on a teleconferencing model of sharing. They allow users to initiate a new conference or join an existing one. A conference consists of one or more users editing a document. There are two drawbacks to this approach. One is that users have to start or join a conference if they want to share a document. This forces an explicit transition between their individual and shared work. Second, users have to scan a list of conferences before they can decide to join an existing one or start a new one. Again, this process creates a discontinuity. Such systems usually force the user to start an application afresh when a conference is initiated. Thus making it impossible to initiate a conference using an application that has already been started. This leads to a high cognitive overhead for initiating a shared session. Moreover, users typically must decide before starting an application whether they will use it for individual work or for collaboration.

One of our goals was to develop a system in which there is no cognitive overhead to a user who is not collaborating. This requirement

implied that users should be able to start their applications as they would do normally, because they cannot know ahead of time whether they are going to use the application collaboratively. Therefore we decided that all files (remote or local), for which a user had read and write access, were potential candidates for sharing. In this model, a user can open any accessible file just as in a single-user application, and does not have to go through a list of conferences before starting.

This approach is adequate for situations in which the UNIX permissions can be used to control access to the sharing. However, the UNIX permissions model does not provide the flexibility and control over file access that may be necessary in certain environments. Therefore, another solution would be to provide the conference model for resource discovery mentioned before. In this model, users must put the file they want to share into a shared space. They must also specify who is to access the file. From the user's point of view, this solution provides finer grained control at the expense of having to specify sharing and access explicitly. Providers of such a system would have to implement their own security and remote file mechanisms rather than reuse those provided by UNIX.

Our proposed system considers sharable resources to be network accessible files with security and group control available via file permissions. We consider all files that the user can read and write, to be sharable. Therefore, UNIX with a networked file system solves our resource discovery problems.

4.2 Rendezvous

After deciding on a resource (file) to open, a process must now determine whether or not it is already in use. If it is, the process must rendezvous with the other processes using the resource. This is also known as "joining a session".

Owing to the unreliability of NFS [10] and `lockd`, the file system can not be used to rendezvous. Instead, the rendezvous must be handled by a separate manager process. In general, the manager will not have access to the same files as the clients. Hence, each client must obtain a specification of the file and pass that on to the manager. The file specification must be network unique so that the manager need only compare specifications to determine whether clients are accessing

the same file. Unfortunately, it is impossible to determine a unique file specification across all currently available versions of UNIX.

Full path names for files are inadequate unique specifications because paths can be aliased and may be ambiguous across hosts. For instance, on a single host, the path `/tmp/foo` and `/tmp/bar` may refer to the same file via either hard or soft links. Across a network, `/tmp/foo` could refer to a different file on each physical host. Therefore, paths reveal little about the actual identity of a file in a networked environment. An obvious solution would be to use the NFS file handle which is an opaque type. However, the handle for the same file can be different on different NFS clients. An alternative is to use a ⟨*host, device, inode*⟩ triple to identify a file uniquely. Unfortunately, UNIX does not provide enough facilities for a client to determine this triple reliably.

The first problem is that it is impossible to identify a remote physical host unambiguously. Multiple network interfaces can provide a single host with multiple Internet addresses. The physical network to which the client of a file server is connected will determine the Internet address that the client has for the server. The best we can do is to use this Internet address, or host name used in a remote mount, to determine the canonical name of the server host. This, however, depends on the system administrator having set up the NIS or DNS [11] database correctly. Application level software cannot solve this problem completely without the support of the operating system.

The next problem is that of identifying the device on the host that stores the file. Surprisingly, `stat`'ing[1] the file from a remote host is not documented to provide this information over NFS. Some platforms put the storage site's device number in the raw device field, but this is neither documented nor required, and many platforms put other information in its place. By contacting the mount daemon directly on the storage site, it is possible to get this information. However, some mount daemons refuse to divulge this information to non-root processes on the NFS client.

Fortunately, there seems to be no problem finding out the inode of a remote file as the `stat` call returns this information.

1. `stat` is a UNIX system call that returns information about the named file including its size, access rights, and device number.

To summarize, our current solution for obtaining a network-unique file specification is to use the canonical host name of the host that stores the file, coupled with the device number of the device containing the file and the inode number of the file on the storage host. When the device number is unobtainable, the exported mount point is used in its place. This unique file specification is sent by each collaborating client to the manager. The manager compares these specifications to determine which clients should rendezvous with each other.

4.3 Session Management

In addition to joining a session, several other tasks must be performed to manage a session correctly. When new processes join or leave a session, all the other members must be consistently informed of those changes in session membership. A new process must be supplied with the latest shared state that may not have been saved to stable storage. This must be reliable in spite of other processes simultaneously joining or leaving the session. Since UNIX does not provide any standard mechanism to support these requirements, we built our own protocol for reliable session management on top of TCP/IP [12].

When one or more new processes try to join a session (see Figure 2), one of the existing processes is chosen and asked to provide the current state. From that time until the state is sent, all messages to existing session members are saved. These saved messages are forwarded

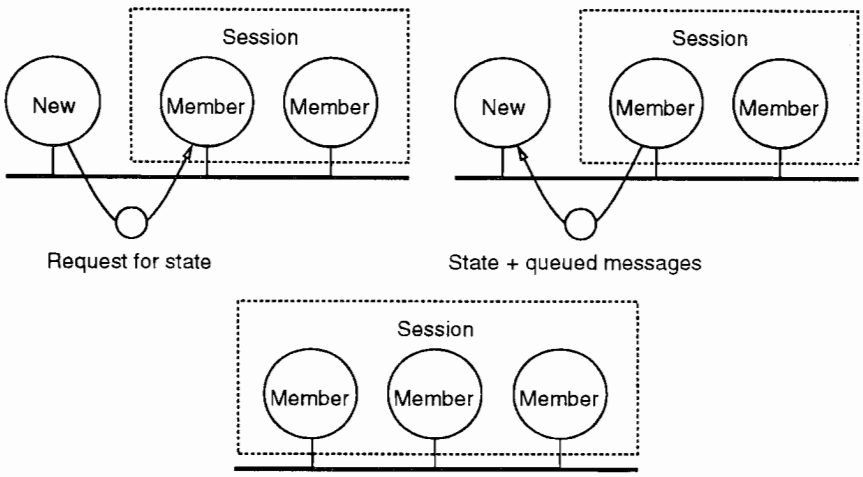

Figure 2: Joining a Session

to the new processes after they are sent the state. The new processes synchronize to the current session state by applying the messages to the provided state.

When clients leave the session voluntarily, they send a message to the other session members indicating that they are leaving the session. Clients may also leave a session on an involuntary basis. If the client process crashes, the other session members will be informed that the crashed client has left the session. However, a client host crash is not easily distinguishable from the network being temporarily unavailable. If the host crashed, the situation can be detected when the host comes back up. A temporary network partitioning will not cause any problems when it becomes reconnected.

Unfortunately, during the time between either a host crash and reboot, or a network disconnect and reconnection, denial of service may be experienced by the other clients. We did not consider the use of `keep-alives` to be an appropriate solution as it increases network traffic and choosing timeouts appropriately is problematical. The problem is fundamental. When the host crashes, we want the connection to break. When there is a temporary network outage, we want the connection to suspend and return when the network resumes normal operation. A library cannot have a single solution to both these situations. An application can provide an appropriate timeout if necessary.

4.4 Communication

Once a session is formed, members have to communicate with each other. Ideally they should address all the session members or any subset of them with a single message. This implies some form of multicasting, which is not widely or consistently available on UNIX platforms. Low level IP multicasting [13] is just becoming available on some UNIX variants, but most routers still do not support it. What is required is a multicast transport protocol [14], which no UNIX vendor currently supports. Therefore, an application desiring this functionality is forced to mimic multicasting with multiple individually addressed messages.

We provide a multicast transport facility by having each client send a message over TCP to a server, which then sends out individual messages over a TCP connection to every member of the session. This is portable across all existing UNIX platforms and is simple to implement.

Another benefit of this scheme for groupware applications, is that the server provides a convenient point for the serialization of messages, so we can easily guarantee that all messages are delivered in the same order to all processes in the same session. The disadvantages are that network traffic is significantly increased, and the fact that the messages go via the server limits the total number of clients that can be communicating at a time. Since delivery order is guaranteed to be the same for all session members, sending clients must wait and execute their message only when it is received. Therefore there is a round-trip latency from the time a client sends out a message to the time it receives the message and can execute it.

Session members must communicate various kinds of data to each other, even though the session may consist of processes running on heterogeneous platforms. Therefore, one client cannot send its raw data to another arbitrary client and expect the data to get through correctly. The solution is to translate the internal representation of the sender to a network data representation on the way out, and translate from that representation to the internal representation of the receiver on the way in. We currently use text as the external representation, but XDR [15] and ASN.1 [16] are suitable alternatives. To assist in these conversions, wrapper functions for marshaling and unmarshaling arguments are provided.

5. *Toolkit Architecture*

The problems and solutions mentioned above are low-level details that every developer of synchronous distributed collaborative applications has to implement. These problems are common to all such groupware applications. Ideally these solutions should be bundled together into a library or toolkit so that they can be re-used by several collaborative applications. This requires the toolkit to be application independent. Moreover, the solutions presented are at the networking and operating system level. It would be useful to provide higher-level capabilities that are closer to the abstractions that developers use. However, care must be taken to allow for the flexibility required by a wide variety of applications.

The next sections describe a toolkit, called COEX, that provides a high-level abstract interface to the communication and sharing mecha-

nisms required of collaborative applications, without unduly restricting developers' policy decisions. It strikes a balance between the need to be application-independent and to provide support for higher-level concurrency control mechanisms suited to the application class.

The COeX toolkit (see Figure 3) consists of a library and a server. The library consists of a kernel and an application-specific module (ASM). The kernel is application-independent and provides basic communication and sharing facilities. The server is also application-independent and provides synchronization and serialization facilities. The ASM provides sharing management functions that are useful for a class of applications like spreadsheets or text editors. The client application interfaces to the ASM.

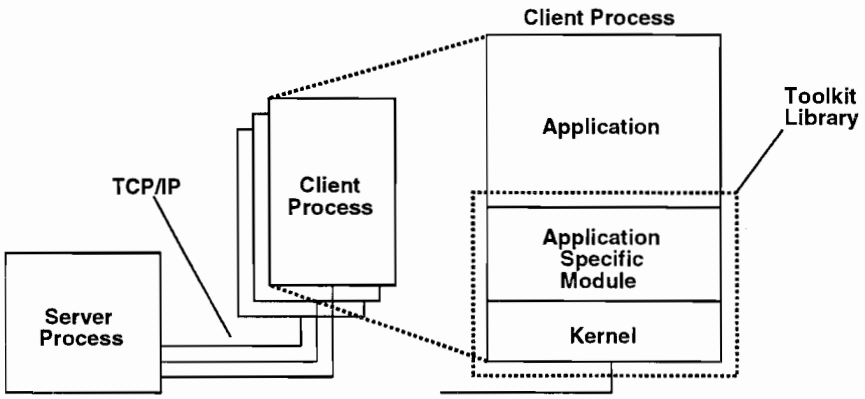

Figure 3: Toolkit Architecture

The next sections divide the COeX toolkit interface into three parts: a system level interface for initializing the toolkit and setting up the connection with the server; a file level interface for accessing file resources and joining sessions; and an application interface for operating on file resources.

6. *System Level Operations*

This section explains how the toolkit communicates with the lowest level of the application. After initialization, the toolkit library opens a connection to a COeX server at a predetermined location. The commu-

nication between the application and the server occurs via a socket file descriptor. The application uses this file descriptor to detect when there is data available from the server and when the library is able to write data to the server. The application may have many different input sources and will have to use its own mechanisms to select among them. Since a message may be split into fragments by the network, a client may require multiple calls to `write` and `read` to communicate complete messages to and from the network.

6.1 Asynchronous Client Input/Output

When data can be read from the server, the application calls the toolkit library to read as much data as it can. If a complete message is received, the library dispatches that message. The execution of that message may involve callbacks into the application.

Similarly, when the toolkit library wants to write data to the server, it queues the data and informs the application that it wants to be notified when it is permissible to write. When the library is notified of that event, it writes out as much data as it can. If this results in all the data being written, the application is informed that the library is temporarily disinterested in writing data on that descriptor. This directly supports the use of `select` or `poll`.[2]

6.2 Termination

The application can shut down the toolkit at any time by calling the appropriate library function. This function informs the server that this client is shutting down and then frees any system resources associated with the toolkit services. During the shutdown phase, the library will callback into the application for each file resource that has been opened via the toolkit. This allows the application to perform concluding operations on state associated with each file resource.

2. The `select` or `poll` UNIX system calls are used to determine which of the supplied file descriptors are ready for reading or writing.

7. File Level Operations

This section describes toolkit operations for accessing file resources, determining the group of clients sharing a file, and generating notifications of membership changes. The toolkit allows an application to access multiple files simultaneously.

7.1 Joining a File's Session

When a client application wants to join a file resource session, it should inform the toolkit library, after making sure that the file exists and is accessible. After making a request to join a file's session, there are two alternatives. Either this is the only application requesting the file, or other applications already have this file open.

In the former case, the application will be notified that it is the only application accessing the file. At that point, the application should create its internal data structures by reading the file from disk.

In the latter case, other applications may have a modified version of the document internally in their buffers. The server chooses one of these applications as the "master" and requests it to send the latest version of the document. The master gets a callback asking it to append the current state of the document to a message. This message is then sent to the new application. When that application receives this message, it realizes that it is not the first to open the file and retrieves the document from the message instead of from the file system.

7.2 Leaving a File's Session

When the application is done with a buffer, it should call the toolkit to leave its session and then clean up its resources. The toolkit then informs the rest of the applications sharing that file that the client is no longer interested in the file.

7.3 Determination of Sharing Group

The application only has to specify the name of the file as it appears on its site—even though the file might be physically resident on another site and accessed via a remote file system. The toolkit figures out

whether two different names, on two different sites, correspond to the same file or not. If two application instances refer to the same file, it is assumed that they are sharing that file. However, sometimes more control is required. If two different applications (not two different application instances) have opened the same file, then they should not share the file. For this reason, the concept of an application identifier is used. COeX assumes that the file is being shared only if the application identifiers of the two application clients match. A similar situation occurs when there are demo versions of applications coexisting with non-demo versions. Usually, demo versions of applications disable the functionality of saving a document to a file. If a demo version and a non-demo version of an application could share a file, the inhibition of the document save feature could be thwarted by having the non-demo version perform the save. To prevent this abuse, the application clients sharing a file must either all be demo versions or all be non-demo versions of the application.

Because the server is a central point of contact for rendezvous, it can be extended to include the functions of a license server. The current COeX protocol allows for the possibility that a client's request to join a session may be denied. The server could check the client's credentials sent with the join request, and could deny the request if the credentials were inadequate. Each client must be able to handle denials gracefully and inform their users of alternatives. Since all connections and messages go via the server, it can implement pay-per-use, pay-per-copy, time-limited, and other licensing schemes.

7.4 Multiple Buffers

The toolkit works with applications that can operate on multiple documents in multiple buffers. If the application supports multiple buffers on the same document, the toolkit treats those buffers as a single buffer.

7.5 Failures and Notifications

All applications sharing a file are notified of any change in the membership of the sharing group.

The COeX protocol is robust in the face of application or server failure. If applications fail while they or others are in the process of join-

ing a sharing session, the protocol handles the situation correctly. It also correctly handles circumstances when more than one application requests to join a session. Race conditions between joining and leaving applications do not affect the operation of the server. If the server should fail, all client applications are notified and their libraries will shut down gracefully. The applications can choose to continue, but they will lack the protection from inconsistencies that COeX provides.

8. Application Level Operations

The operations described in this section fall in the domain of the ASM and hence the details of each operation will be different from one application class to another. However, the general principles are the same.

8.1 Messages

Messages are the unit of communication between application instances. From the application-writer's point of view, messages are opaque types with a limited interface. COeX currently supports the adding and removing of several different kinds of items from a message. These items include integers, floating point numbers, null-terminated strings and blocks of arbitrary data. The order of adding items to a message should be the same as the removal order. The items are not typed. Therefore, the application must be aware of the type and order of the items in a message. In addition to zero or more items, each message has a resource identifier and an operation code. The resource identifier is passed back explicitly to any callback invoked as a result of receiving the message. The application can use this identifier to distinguish messages destined for different buffers. The operation code is used internally by the library. Typically, the first item of an application message will be an application-specified operation code.

8.2 Maintaining a Consistent Distributed State

The supported model of sharing is that of a consistent replicated distributed state. The server is not a database and does not store any

application data. Each application instance stores a copy of the current state of the document in its buffers. COeX provides mechanisms by which the application instances can coordinate accesses such that each instance has a consistent copy of the shared state. The data model can be thought of as a replicated distributed cache.

At the lowest level, the kernel can send messages to all instances of an application sharing a document. The system guarantees that such messages will be received by the applications in the same order. Hence, if each application instance carries out the requisite operation on the receipt of a message, each of the instances will have a consistent state. Such messages are called *operations*, or *Ops* for short.

In practice, there can be problems. For example, certain operations may be allowable only if some preconditions are satisfied. For example, a user might be restricted to locking regions that have not already been locked by someone else. The sending application can check to see if the preconditions are satisfied before initiating the operation. However, in the time between the sending of the operation and its receipt, applications may receive other operations that might cause the preconditions to be invalidated. For example, another application might have locked a subrange of the requested region. This results in two operations that might conflict with each other.

There are two approaches to solving this problem: optimistic and pessimistic. The optimistic approach involves sending out the operation, but having each application instance independently check the action's precondition on receipt before performing the action. The pessimistic approach involves the initiating application sending the operation only when it is sure that there will be no other operations between the sending and the receipt of the operation. On receipt of the operation all application instances can immediately execute the operation without any additional checking. Since the system guarantees that there are no intervening operations, pessimistic approaches can reduce concurrency. The library provides the mechanisms to support both the optimistic and pessimistic solutions. Application designers can choose the approach most suitable to their environment.

Our toolkit provides a special operation, called a *clashing operation*, or *ClashOp* for short. A ClashOp will either succeed and be sent, or it will fail. If it succeeds, the system guarantees that no other ClashOp intervenes between the sending and receipt of that ClashOp. If it fails, the application is notified and can retry the ClashOp later.

The application can use ClashOps to distribute operations that might potentially conflict with each other and thus guarantee that only one ClashOp can occur at a time. ClashOps can be used to implement a pessimistic solution to the consistency problem. However, since ClashOps inhibit concurrency, they should be used sparingly and judiciously.

8.3 Network Traffic

A simplistic approach for maintaining a consistent state would have applications send an Op or ClashOp every time the state was modified. If individual views of the data have to be shared as well, then operations will have to be sent whenever any individual view is changed. This approach has the benefit that all changes made to any part of the state will be reflected immediately in the state of all the collaborating applications. However, if every change is propagated, it may be distracting to the other collaborators, and will cause excessive network traffic.

An alternate approach is to have users reserve parts of the document for their exclusive use. Owners of the reserved parts can make any local changes that they want without sending any messages. When they are satisfied with the results, they can release their reservations causing the changes to be propagated to the others. This scheme has the advantage of reducing network traffic significantly, at the expense of not showing the changes to others immediately.

COeX provides the mechanisms for both these approaches. Application designers can choose the most appropriate one for their applications. Shared graphical editors may prefer to use the first approach since it is very useful to immediately notice changes made by others in this environment. Spreadsheet users, on the other hand, often experiment with cell values and hence would benefit from using the second method.

8.4 Regions

The smallest object in the document that can be reserved, or locked, is called a *granule*. A collection of granules locked by the same user is called a *region*. A user may lock multiple regions in a single document. One way of characterizing regions is by the dimensions of their extent. For example:

0-dimensional: Each object in a structured graphic editor could be a granule. In this case, each region consists of exactly one granule. The extent of the region can be represented by a point and hence is without a dimension.

1-dimensional: In a text editor, a single character could be a granule. Regions consist of a linear string of contiguous characters. The extent of a region is represented by a one dimensional line.

2-dimensional: Each cell in a spreadsheet could be a granule. A two-dimensional area of contiguous cells forms a region.

The ASM provides operations, appropriate to the application class, for manipulating regions. There are three basic operations: lock, update and unlock. A *lock* operation reserves the specified region for the exclusive use of the application requesting the lock. *Update* transmits the current contents of the specified region to all the sharers of the document. This makes the latest changes available to the others. In addition to updating the contents of the locked region, an *unlock* removes the reservation on the region, making it lockable by others. A pessimistic (Section 8.2) implementation will typically use ClashOps to implement the lock and unlock operations, and use Ops to implement updates.

The update and unlock operations denote a region by a unique identifier returned by a lock operation. The requested region in a lock operation is specified by offsets from the preceding region or the beginning of the buffer. This allows the specification of the requested region to be independent of any concurrent update operations. Therefore, the update operation does not conflict with a lock or unlock operation.

In addition to these basic operations, the ASM provides operations to keep the region information current when changes are made to locked regions. For example, adding characters to a locked text region will change the boundaries of the region and the location of subsequent regions. The toolkit has to be notified of this change so that it can keep its internal data structures consistent with the application's structures. The library also provides predicates that the application can call to decide whether a particular operation is allowed or not. When any lock information changes, the application is called with information on the changes. This should be used by the application to update the views of locked regions provided to the user.

8.5 Other Application-Specific Operations

The operations relating to regions and locking are provided by most ASMs. There are other operations, like "save the document", that may also be provided. The application should use this operation to make sure that only one of the application instances performs a save at any given time. This will avoid inconsistent copies being stored on stable storage. Each application class may have other application-dependent operations (e.g., the *insert row* operation in a spreadsheet).

9. Operating System Dependencies

This paper has described some of the impediments to the construction of robust collaborative applications, and demonstrated solutions to these problems under UNIX. However, the ideas behind these solutions are applicable to other systems, though implementation details will be different.

Our toolkit is mostly independent of the operating system. There are three aspects that are system specific and have been isolated to a few files. They are: access to remote files, generating unique identifiers for remote files, and a reliable network transport protocol.

UNIX provides NFS for access to remote files. It should be possible to support other remote file systems like DFS, RFS, and Netware with minimal changes. The current method for generating unique file identifiers depends on NFS. Appropriate algorithms for generating unique file identifiers would have to be created for other remote file systems. TCP/IP provides a reliable network transport protocol and is widely available. Any communication protocol that supports reliable sequenced messages is adequate for implementing COeX.

10. Evaluation

The COeX system is relatively small, weighing in at approximately 10,000 lines of C. The stripped server (Sparc under SunOS 4.1.2) occupies about 50k on disk. The library for a text-based application (Sparc under SunOS 4.1.2) is approximately 40k (text + data + bss). The amount of memory used in practice is application-dependent.

	Application data (in bytes)						
	1	1k	2k	3k	4000	4k	8k
Multisite Op	5	9	12	15	17	na	na
Multisite ClashOp	9	12	16	19	21	na	na
Singlesite Op	5	7	10	12	13	17	30
Singlesite ClashOp	8	11	14	16	16	21	34

Table 1: Round-trip latency in ms

Table 1 gives an idea of the round-trip latency of messages using SparcStation 1s running SunOS 4.1.2 connected by a 10Mb UTP Ethernet. The timings include the overhead in the COEX library, and for message creation, destruction, and dispatch. The server and two clients were each run on different hosts for the "Multisite" data collection. For the "Singlesite" results, all three processes were run on the same site, thus not sending any data across the network. We measured both Ops and ClashOps, each with different amounts of application data. For application data above 4096 bytes in the "Multisite" case, our timing results contained a lot of variation and so are not included. We rounded the timings to the nearest milli-second.

From the table, the overhead of a ClashOp over an Op is approximately 4ms, and each extra 1024 bytes of application data increases the latency by approximately 3ms.

11. Experiences with the Toolkit

Our original prototype was built in 1990. We enhanced GNU Emacs [17] to share text files among multiple editor instances. All modifications were made entirely in Emacs Lisp. The architecture used a centralized server that maintained the shared state. We realized that the communication and sharing issues we dealt with were common to all groupware applications and our solutions could be abstracted into a toolkit. In early 1991 we started work on the first version of the toolkit. It supported a distributed state architecture and was written in C. The toolkit was integrated into an existing Motif-based commercial spreadsheet. This enhanced multi-user spreadsheet has been on the market since late 1991. In early 1992 we started work on the second version of our toolkit. This version added support for multiple buffer

applications and involved a total rewrite of the library code as a result of our experience with the first version. We also built ASMs for text editors and structured graphics editors. This version was completed in mid 1992.

In addition to the commercial spreadsheet, we used our toolkit in four other applications: a text editor, a structured graphics editor, a puzzle game, and a brainstorming tool. The group text and graphics editors were modifications of the `sted` and `idraw` applications that are a part of the `InterViews` [18] distribution. In addition to making `sted` into a group editor, we added facilities for one-shot synchronization of views and selections. We also added history visualization features that provide a graphical representation of the collaborative events on the file over time. Each user can lock regions of the buffer and can modify the data. Their changes are made available to the other members of their sharing session on update or unlock. Since `idraw` follows a select-then-operate metaphor, we lock the selected objects implicitly. Therefore, there is no change in user gestures between the single-user and group-aware versions. To demonstrate the ability of our toolkit to support multiple policies, we provided for two update modes. In the first mode, the other session members see changes only on deselection. In the second mode, the changes are reflected immediately to all session members. Users can dynamically select between these modes via a menu item. In single-user mode (i.e., when there is only one user sharing a file), there are no user-observable differences between the original and enhanced applications.

The puzzle game is a modified version of the `puzzle` demo from the X distribution. We added a file argument to the program to provide a rendezvous point. All the previous examples were modifications of already existing applications. The brainstorming tool was written explicitly as a multi-user application. Each user has a private area where they can edit a string. On typing a carriage return, their private string is sent to all the session members and appears in a global, read-only, scrolling list of items. The last two applications did not require any high-level application-specific support and hence use a null ASM.

Our experience with these applications indicates that the primitives supplied by the toolkit are appropriate for collaborative applications and are adequate for a variety of different application classes. Use of the toolkit greatly reduces the amount of time required to build a collaborative application. For example, it took us approximately 2 hours

to build a simple single-user brainstorming application. It took another 2 hours to integrate it with our toolkit and have it work in multi-user mode. It then took another 4 hours to clean up the user interface and add user interface features. In the case of modifying already existing applications, our experience suggests that most of the time is spent in understanding the code structure and determining where the modifications are to be made. For the case of `idraw`, which is a fairly complex application, the total modification process took a week. It is our experience that the basic integration takes on the order of a few days. However, to ensure perfect coverage, and to make sure the changes are robust under all situations, can take considerably more time.

Our integration architecture and philosophy is similar to that of DistEdit [19]. The application's original modification functions are replaced by glue function stubs that call our primitives. When messages are received, our toolkit calls back to glue functions that then call the application's original functions. Table 2 gives an idea of the amount of modifications required to existing applications. The glue code is more or less the same for all applications, except for function names and application-specific checks.

Application	Original	Modified	Glue
Spreadsheet	119K	1K	2K
Text editor	500	100	300
Graphics editor	19K	700	800

Table 2: Lines of code for modified applications

Our experience shows that applications that have a clean separation between data and views, are event-driven, and validate all operations before execution, are easier to make collaborative. Not surprisingly, applications that are well-structured and those that follow good software engineering practices are easier to convert.

12. Related Work

Like other toolkits, COEX abstracts common features from a variety of potential applications and provides a general interface to that functionality. This general interface insulates the application from platform variations. Performance and reliability enhancements to the toolkit are

transferred to the applications simply by relinking. Thus, code reuse is a major benefit of toolkits. A good toolkit implements flexible mechanisms on which application programmers can impose their own policies.

COeX provides communication and concurrency control to share data among replicated instances of an application. It is independent of any particular user interface toolkit. It uses a file as the rendezvous point for the session. High-level support for various application classes is provided along with the ability to modify parts of the shared data concurrently.

Other toolkits, like Sun's ToolTalk [20] and HP's SoftBench [21], also offer communication among applications by providing a "software backplane" to which applications can connect. However, both are oriented towards dissimilar applications using loosely-coupled communications, and hence tend to rendezvous via the common user of the applications rather than a common file. ToolTalk does provide for file-based rendezvous, but requires a server on every NFS site to support this feature. Since these architectures expect the communication between applications to be infrequent, they are not suitable for real-time data exchange.

The ISIS toolkit [22] provides communication mechanisms for fault-tolerant distributed computing with large numbers of processes. It requires a central server for session management, though not for normal communications. On the other hand, COeX is tailored towards supporting a small number of tightly-coupled distributed applications working together on a shared state. Additionally, the ASMs provide high-level sharing and consistency support for a wide range of application classes. While ISIS solves similar low level communication and session management problems as our toolkit, it is over an order of magnitude larger, involves greater complexity, and provides comparable latencies (see Table III in [23]) for small sized groups. However, it scales better than COeX—in terms of large messages, large groups, and a large number of groups. Since our application domain contains sharing groups that are small, size and complexity outweigh scalability.

DistEdit [19] is an independently developed toolkit that provides high-level support for building collaborative text editors and uses ISIS for its low level communications. It allows sharing among different text editors and provides integral support for undoing actions in a

collaborative environment [24]. Like COeX, it employs a distributed architecture and is independent of the user interface. Unlike COeX it only supports text editors, but provides undo support.

Groupware toolkits like Rendezvous [25], LIZA [26], and Suite [9] use a hybrid architecture with a central site for data storage, but replicate the user interface which may contain replicated cached copies of the central data objects. The central storage site provides a convenient synchronization point for access to the shared state. The replicated user interfaces allow user interface tasks to be performed locally, thereby increasing responsiveness. Each of these systems provides a framework for building collaborative applications, but developers are restricted to using the provided user interface components. All these frameworks rendezvous via the central application. Suite additionally provides extremely flexible, dynamically modifiable coupling among the user interface objects, including the ability to share uncommitted values or partial results. The default concurrency control scheme used by Suite is to implicitly lock the data object when a user modifies the corresponding user interface object, and to unlock when its value is committed.

GroupKit [27] is a toolkit for building collaborative applications using the `InterViews` user interface toolkit. It provides `InterViews` users with extremely convenient mechanisms to add collaborative user interface features such as telepointers and annotations using overlays. GroupKit uses a conferencing model for distributed state collaboration and provides flexible support for implementing various conference and floor passing policies. It does not provide any concurrency control mechanisms, but supports application-specific functionality via subclassing.

MMConf [4] is a framework for constructing collaborative applications with distributed state. Applications built with MMConf have to use the provided user interface objects. MMConf provides a variety of floor-control mechanisms for regulating access to the application. No concurrency control is provided in the free-for-all mode.

13. Summary and Future Work

This paper highlights some of the issues faced by developers of synchronous distributed collaborative applications. We show how these issues can be addressed in the context of a UNIX implementation. These

solutions are collected into a toolkit that provides a higher-level abstraction for the collaborative application developer.

The toolkit provides support for sharing application-level objects rather than user interface objects. This makes it possible to use COeX with applications written for any windowing system or with any user interface toolkit. Since individual views are not necessarily shared, each user can have immediate feedback for user interface operations like scrolling. The replicated architecture also provides performance benefits since each client has a full copy of the application data and can operate on it locally. Each client can make concurrent changes to parts of the shared state. Developers can use Ops, ClashOps, and the locking mechanisms provided by ASMs to implement a concurrency control policy that is appropriate for their application.

COeX does not interpret application data and hence can be used for any type of application, including text or graphics. The ASMs provide high-level support for sharing application-level objects within various application classes. Our experience building five different applications with the toolkit suggests that the functionality provided by the ASMs is appropriate. Four of the five were existing single-user applications. We found that using COeX allowed us to quickly convert these applications for collaborative use. Most of the conversion time was spent in understanding the application's structure, deciding where to make modifications, and making sure that we had complete coverage of the application. Integrating COeX into a new application was even quicker—usually on the order of a few hours to obtain a minimally functioning system.

Users are attached to their comfortable everyday single-user tools and are loath to make any new changes. This poses a significant obstacle for the acceptance of multi-user collaborative tools. One method of surmounting this obstacle is to provide enough extra functionality to make it beneficial for users to change. Another method is to make the transition from single-user tools to multi-user tools as smooth as possible. COeX supports this transition in two ways. First, by providing packaged solutions to common communication and sharing problems, our toolkit facilitates the conversion of popular single-user tools to group-aware ones. This allows users to continue utilizing their favorite familiar tools. Second, by providing file-based rendezvous, COeX supports the construction of collaborative applications that behave the same in single-user mode as their single-user counterparts. This is in

contrast to other frameworks which use a user name, application name, or conference name to rendezvous. Since most single-user tools operate on files, our approach is appropriate.

COeX uses a central server for all communication, which may be a bottleneck when scaling up the system. In our application domain, the number of clients per session typically is small. For example, it is unlikely that hundreds of users want to edit the same memo. However, it is likely that there are a large number of sessions. In this situation, the single server can be a significant bottleneck for traffic. A simple load balancing scheme can alleviate the situation. Assuming a fixed number of independent servers, the unique file identifier can be hashed in such a way that each file is deterministically associated with a particular server. Thus, each server serves a particular group of files. While this scheme works for uniform file accesses, it is a static solution, and could result in unbalanced server loads for certain file usage patterns. A more dynamic, though significantly more complex, scheme would be to use a multiple server architecture. Each client can connect to any server, and the servers exchange data among themselves. This is a topic for future work.

A related issue is that of latency. So far, our experience has been limited to low latency local area networks. In a high latency situation, over a wide area network or over dialup lines, the round trip delay to the server could be prohibitive. It is likely that different protocols and different notions of consistency will be required.

COeX is appropriate for static media. Its applicability to continuous media like audio and video are open questions.

In conclusion, our work provides a first-cut at solving many common problems faced by developers of collaborative applications. In particular, we provide the ability to rendezvous via files. However, there is still more work to be done in handling scaling, latency and continuous media.

References

[1] Clarence A. Ellis, Simon J. Gibbs, and Gail L. Rein. Groupware: Some issues and experiences. *Communications of the ACM,* 34(1):38–58, January 1991.

[2] Keith A. Lantz. An experiment in integrated multimedia conferencing. In *CSCW'86: Proceedings of the Conference on Computer-Supported Cooperative Work,* pages 267–275, Austin, Texas, December 3–5 1986. Also published in [28], pages 533–552.

[3] Hussein M. Abdel-Wahab and Mark A. Feit. XTV: a framework for sharing X window clients in remote synchronous collaboration. In *Proceedings of the IEEE TriComm'91: Communications for Distributed Applications and Systems,* pages 159–167, Chapel Hill, North Carolina, April 1991.

[4] Terrence Crowley, Paul Milazzo, Ellie Baker, Harry Forsdick, and Raymond Tomlinson. MMConf: An infrastructure for building shared multimedia applications. In *CSCW90: Proceedings of the Conference on Computer-Supported Cooperative Work,* pages 329–342, Los Angeles, California, October 7–10 1990.

[5] Clarence A. Ellis and Simon J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data,* pages 399–407, Portland, Oregon, May 31–June 2 1989.

[6] Alain Karsenty and Michel Beaudouin-Lafon. An Algorithm for distributed groupware applications. Rapport LRI No. 785, Laboratoire de Recherche en Informatique, Université de Paris-Sud, Orsay Cedex, France, October 1992.

[7] D. Garfinkel, P. Gust, M. Lemon, and S. Lowder. The SharedX multiuser interface user's guide, version 2.0. Technical Report STL-TM-89-07, Hewlett Packard Laboratories, Palo Alto, California, March 1989.

[8] 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. *Communcations of the ACM,* 30(1):32–47, January 1987. Also published in [28], pages 335–366.

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

[10] Sun Microsystems, Inc. NFS: Network file system protocol specification. RFC 1094, DARPA, March 1989.

[11] P.V. Mockapetris. Domain names—implementation and specification. RFC 1035, DARPA, November 1987.

[12] J.B. Postel. Transmission control protocol. RFC 793, DARPA, September 1981.

[13] S.E. Deering. Host extensions for IP multicasting. RFC 1112, DARPA, August 1989.

[14] S.M. Armstrong, A.O. Freier, and K.A. Marzullo. Multicast transport protocol. RFC 1301, DARPA, February 1992.

[15] Sun Microsystems, Inc. XDR: External data representation standard. RFC 1014, DARPA, June 1987.

[16] CCITT. Specification of Abstract Syntax Notation One (ASN.1). Recommendation X.208, CCITT, 1988.

[17] Richard Stallman. *GNU Emacs Manual*. Free Software Foundation, Cambridge, Massachusetts, sixth edition, March 1987.

[18] Mark A. Linton and Paul R. Calder. The design and implementation of InterViews. In *Proceedings of the C++ Workshop,* pages 256–267, Santa Fe, New Mexico, September 9–10 1987. Usenix Association.

[19] Michael J. Knister and Atul Prakash. DistEdit: A distributed toolkit for supporting multiple group editors. In *CSCW90: Proceedings of the Conference on Computer-Supported Cooperative Work,* pages 343–355, Los Angeles, California, October 7–10 1990.

[20] SunSoft, Mountain View, California. *ToolTalk 1.0 Programmer's Guide,* September 1991.

[21] Martin R. Cagan. The HP SoftBench environment: An architecture for a new generation of software. *Hewlett-Packard Journal,* 41(3):36–47, June 1990.

[22] Kenneth P. Birman. ISIS: A system for fault-tolerant distributed computing. Technical Report TR-86-744, Computer Science Department, Cornell University, Ithaca, NY, April 1986.

[23] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems,* 9(3):272–314, August 1991.

[24] Atul Prakash and Michael J. Knister. Undoing actions in collaborative work. In *CSCW92: Proceedings of the Conference on Computer-Supported Cooperative Work,* pages 273–280, Toronto, Canada, October 31–November 4 1992.

[25] John F. Patterson, Ralph D. Hill, Steven L. Rohall, and W. Scott Meeks. Rendezvous: An architecture for synchronous multi-user applications. In *CSCW90: Proceedings of the Conference on Computer-Supported Cooperative Work,* pages 317–328, Los Angeles, California, October 7–10 1990.

[26] S.J. Gibbs. LIZA: An extensible groupware toolkit. In *CHI'89: Proceedings of the Conference on Human Factors in Computing Systems,* pages 29–35, Austin, Texas, April 30–May 4 1989. ACM, Addison Wesley.

[27] Mark Roseman and Saul Greenberg. GroupKit: A groupware toolkit for building real-time conferencing applications. In *CSCW92: Proceedings of the Conference on Computer-Supported Cooperative Work,* pages 43–50, Toronto, Canada, October 31–November 4 1992.

[28] Irene Greif, editor. *Computer-Supported Cooperative Work: A Book of Readings*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1988.