CS 290: Collaborative Systems [1]
Section 3: Collaborative Infrastructures
Prasun Dewan

1. COLLABORATIVE INFRASTRUCTURES

In the previous section, we discused collaborative applications, that is, collaboration systems that provide end-users with some functionality to collaborate with each other. In this section, we will discuss collaborative infrastructures, that is, software systems that provide programmers with abstractions to implement collaborative applications.

Because of the diversity in collaborative applications, none of the collaborative infrastructures that have been developed so far is suitable for, or even capable of, implementing all of the collaborative applications we studied in the previous section. As a result, a large variety of infrastructures have been developed for implementing different kinds of collaborative applications. In fact, there may be more diversity in the collaborative infrastructures than in collaborative applications!

Therefore, when we study a collaborative infrastructure, we will look at the features of collaborative applications it can support, thereby evaluating its *flexibility*. Another criterion for evaluating an infrastructure is the *automation* it offers, which measures the amount of effort required to implement functions it "supports". We will distinguish between two basic ways to support some function: *enabling*, which simply allows the implementation of the function, and *automating*, which relieves programmers from implementing how the function is implementing, requiring them to only specify what they want. To understand the difference between these two forms of support, consider an assembly language, Pascal, and C. Pascal automates record management but does not enable interrupt handlers since it does not allow access to hardware registers. An assembly language enables both record management and interrupt handlers, while C automates record management and enables interrupt handlers. It is similarly important, in collaborative infrastructures, to provide a fine balance between automating and enabling. It is also important to offer space and time *efficiency*, another evaluation criterion we will use. Finally, we will look at the amount of *reuse* of existing systems supported by the infrastructure.

Dividing collaborative systems into applications and infrastructures is a simple but coarse classification scheme, since some systems have characteristics of both. Consider ClearBoard-2, which provides a shared whiteboard and video overlays. While the former is a specific application the latter is a general abstraction that can be used to provide face-to-face awareness in arbitrary shared applications. Similarly, the mail system can be considered an application since it directly interacts with the end user, and also an infrastructure since it provides a portable method for communicating information between arbitrary distributed applications. In general, a software system can be divided into may layers and a higher-level layer is an "application" for infrastructure layers below it. It is perhaps not surprising that kernel/OS researchers would regard most of the systems we discuss below as applications. They have been classified as infrastructures since each of them needs at least one layer above to interact with end-users. Later, we will look at their

positions in a layered collaboration architecture.

1.1 Example: True "Hello World" Program

To understand and evaluate these systems, we will discuss how they may be used to implement a simple collaborative program. The program is a variation of the true "hello world" example of [?], which says hello truly to the world. The "world", in this case, is a set of users interacting with the program. We allow a user to change the initial greeting of the program, which is then shared with the remaining users. We leave out many details of the application such as when a change is propagated to other users, or whether there is any concurrency control. How we will handle these issues will depend on the capabilities of the infrastructure - we will discuss solutions that are possible and easy to implement using that infrastructure. When it is really easy to do so, we will look the exact code required to implement the application; otherwise we will simply sketch the solution.

1.2 Shared Distributed Repositories

1.2.1 *File System: Coarse-grained Data.* Given a file system shared by multiple processes, here is a general scheme for developing a collaborative application: For each user interacting with the application, we create one or more processes that interact with the user, and linking among the users is implemented via one or more files shared among these processes. A file may directly store binary representation of shared data or a textual representation of it. It may be created by one of the user processes or by a special process executed before any user accesses it.

Example: A special initializer program is executed with the session name as an argument:

```
session -name ourHelloWorld
```

The program creates a greetings file called `helloWorld` and initialize its contents and access list.

To join the session, another user can execute the `helloWorld` program:

```
helloWorld -join ourHelloWorld
```

This program is an editor that allows its user to view and change the greeting stored in the file. Several variations of it are possible: The simplest approach is for it to provide users with explicit commands to load/store the greeting in the file. Instead of requiring the user to explicitly load a new greeting, it can periodically poll the file. Instead of requiring the user to explicitly store a new greeting, it can automatically store the greeting on every character. Moreover, instead of writing directly to the shared file, it can create a separate version, and always read the latest version of the file. It can check-out a version in the locked mode to prevent conflicts. Otherwise, it can use merge facilities to combine two versions that were concurrently modified.

There are several advantages of using a file system for implementing collaborative applications. A file system automates the implementation of persistence, access control, and concurrency control. Moreover, in artifact-based collaborative applications supporting implicit sessions, the naming scheme provided by a file system can be used as a basis for naming sessions, thereby automating details of hierarchical

names, symbolic links, and other rich concepts provided by modern file systems. Furthermore, file-based version control systems automate the creation of multiple versions of shared data and the diffing and merging of these versions. Finally, file systems also come with programs that provide efficient searches of textual data.

However, in comparison to some of the other infrastructures discused below, a file system has four main disadvantages. First, a processes must poll the file to determine if it has been changed by some other process. Since a high polling frequency (e.g. 3 seconds) can severely degrade the system performance, this approach is not suitable for real-time collaboration. This is a problem in all systems that facilitate sharing through *passive repositories* of data. Later we will look at several examples of *active repositories*, that is, repositories that allow user-defined triggers to be associated with updates to the data.

Second, a file resides primarily on disk, and thus communication between processes can involve potentially costly disk accesses. This problem is reduced but not eliminated by caching, which makes this technology further unsuitable for real-time sharing of rapidly-changing state such as scrollbar and pointer positions.

Third, a file system automates a small subset of the functions of a collaborative application. For instance, while it enables (non real-time) coupling, the programmer is responsible for implementing the details of depositing and fetching data from the file. Similarly, it does not provide automatic support for undo. Later we will look at systems that make application programmers totally unaware of these functions.

Fourth, a file system provides coarse-grained units of data, and functions such as concurrency and access control that operate on these units cannot be used directly by all applications. For instance, applications that need different records of a bibliography to be associated with separate locks must implement their own concurrency control.

Finally, this approach to developing collaborative applications does not work if the collaborators do not share a file system. Thus, it is not suitable for wide-area collaboration.

1.2.2 *Traditional DBMS: Fine-Grained Data.* The last problem is addressed by a traditional (relational) database management system. It also provides a persistent, disk-resident repository of shared data but the persistent data are relations of records instead of files. To create a collaborative application, we simply have to replace the shared file with a shared database, which would now provide the linking among the processes interacting with the user.

Example: Same as the file case except that: We replace the file with a relation of strings in which we store a single record to hold the greeting. An application program uses a query language to insert, modify, and select that record. Since traditional relational databases fix the size of a record, application programs would need to pad/truncate user input.

A DBMS can discriminate among the different records and fields of a database, and can thus provide a finer granularity of sharing, protection, and concurrency control. In particular, it can automate the locking semantics of the bibliography application. In fact, concurrency control schemes in DBMS are more sophisticated than those developed for file systems, providing support for atomically executing a transaction on multiple objects. In contrast, a file system requires users to explicitly

lock/unlock all files that an operation must modify and provides no support for atomicity. Later, we will study the subject of concurrency control in more depth.

Another strength of a relational database system is the powerful predicate-based query language it provides for efficiently searching through large amounts of data. The language is particularly useful for searching for awareness information and the status of a shared artifact or a collaboration task. For instance, it automates queries to find all collaborators at a particular location, all talks scheduled within a particular time range, all workflows that are not complete, and all procedures modified by a user. The cost of manually programming such queries in a collaborative application can be high [?].

On the other hand, it has the problems of passive disk-resident repositories. An additional problem is that a database system and the processes that access it typically use different representations of data. As a result, a DBMS forces its clients to convert between different representations of data, which is referred to as the *impedance mismatch* problem [?]. Impedance mismatch makes the overhead of developing collaborative applications high since programmers must write translation code that converts between the different representations. It also increases the response time of the applications because of the overhead of executing the translation code at runtime. Impedance mismatch is also a problem in single-user applications that use the database to store their persistent data. It is a more severe problem in multiuser applications developed using this approach, since they use the database to also store shared, possibly non-persistent, data such as a scrollbar position or the list of users in a session.

Thus, a DBMS is even less suitable than a file system for supporting real-time collaboration. Impedance mismatch would also occur in the latter if a textual rather than directly binary representation of data is stored in a file. However, even in the case of text files, all information about an application data structure such as a parse tree can be stored in a single, variable-length file. In the case of a traditional DBMS, such a data structure would be spread out in multiple fixed-size records and costly joins would be required to fetch and store all of these records [?].

1.2.3 *Lotus Notes: Rarely-Connected Document Replicas.* Lotus Notes [?] provides a storage structure, called a document, that is a mix between database records and files. Like a database record, it can contain fine-grained, fixed-length fields of predefined types, and like a file, it can store variable-length data of predefined and programmer-defined types. In particular, it can contain variable-length strings for storing document text. Thus, Notes documents, like mail messages in Information Lens, are semi-structured. In fact, one of the most popular applications of Notes is Notes Mail, which stores mail messages as semi-structured documents and provides various schemes for sorting them. Notes provides a unified framework for processing mail messages and other documents. For instance, the same program can be used to browse through messages and other documents, and any document can have a "response" linked to it.

Notes is an example of a replicated storage system, that is, a system that creates multiple replicas of a data structure on different sites and ensures consistency of these replicas. Creating a replica at a site promotes fault tolerance, and more importantly, allows read operations to be resolved locally without requiring any

distributed communication.

The write operations are more complicated since they need to propagated to all replicas to ensure consistency among them. Traditional replicated file and database systems ensure a strong *immediate* consistency model, that is, update all (or at least a quorum) of the replicas when a write operation commits at the local site. Notes, like the News application we saw earlier, supports a weaker *eventual consistency* model, that is, ensures that if all update activities stop the replicas will converge after a finite amount of time.

The main motivation for supporting eventual consistency in News was the number of sites involved - news postings would become unbearably slow if we had to commit the posting to all replicas. This is also a motivation in document databases, but perhaps a stronger reason is a *rarely-connected computer*, that is, a computer such as a laptop that is more often than not disconnected from other computers. To allow the user of such a computer to access data in the disconnected state, we would like to create local replicas of shared document databases but cannot ensure immediate consistency of a write operation. Instead, we would like to allow the user to proceed with other operations, and propagate the results of these later when the computer connects to others.

Like News, Notes supports a decentralized consistency scheme wherein pairs of computers exchange changes to their documents. Each computer knows about one or more other computers or peers that store replicas of the local databases. At some point, it can "replicate" with a peer, when it does a 1-way pull of changes from the peer. (The peer can simultaneously pull changes from the local computer.) A user can manually ask for immediate replication with a peer. Moreover, a replication schedule (specified by the system administrator) can trigger automatic replications.

The source and destination computers have autonomy over what parts of their replicas are sent and received, respectively. The source computer sends only those documents parts to which the destination computer has write access. Moreover, it determines if deletions and old changes are shared with the destination. Conversely, the destination receives only those document parts to which the peer has write access. Moreover, it can specify the types of data it should receive from the source. It can also specify also queries dynamically selecting the the database parts it wants replicated. Finally, it determines if changes to access-control lists and replication parameters associated with a database should also be received from the source.

Notes provides an automatic scheme for merging the destination replica with the source replica. It associates objects with timestamps, and replaces an older version of an object with a newer object in the source. Like News, it merges simultaneous additions to the database by performing both operations. Concurrent updates to the same document is hard to resolve and Notes simply creates both alternatives as parallel versions, leaving the user to choose one of them. One of these versions is made the primary document and the other is made a secondary "response" to it. News does not have to address the concurrent update problem since users can simultaneously update a News message, though they can simultaneously add two messages to a bulletin board. As we shall see soon, the merge problem is addressed in several other systems, and we shall address it in-depth later.

Example: Same as the database case except that: We create a document with a variable length text field instead of a database record. Also we can create a local

replica of the document and operate on it in the disconnected mode. In particular, we can modify it in this mode . If we want automatic notification of changes to a greeting, then a system administrator must define a replication schedule - otherwise we explicitly replicate to exchange changes. If both computers have simultaneously changed the greeting, then one of the greetings will be made a response to the other. If we want to receive but not transmit our changes to the greeting, then we give others write access but not read access to our replica.

Notes is useful not only for merging replicas modified by different users but also those owned by the same user as he moves among workstations (e.g office and home computers). In fact, the merge problem is much easier in this case, since concurrent changes are never made to the replica.

1.2.4 *Coda: Multiple Computer Couplings.* So far, we have seen systems designed for for two forms of networks - the traditional strongly-connected networks in which disconnection of a computer is an exception, and rarely-connected networks in which the reverse is true. In the former, the exception occurs when a computer fails while in the latter it occurs when it replicates or merges its changes. Coda [**?**; **?**] is designed for hybrid networks in which a computer may be strongly-connected, rarely-connected, or *weakly-connected*, that is, connected to the network through an expensive low-bandwidth connection such as a cellular connection. Instead of creating local replicas, Coda creates a local cache of data. The semantics of file operations issued by the computer depends on its connection to the network.

When a computer is strongly-connected, it caches data accessed by it and immediately propagates the results of local operations to all other strongly-connected computers, which can refresh their local caches. Thus, if all computers are strongly-connected, strong immediate consistency is maintained among the caches. When a computer is disconnected in the rarely-connected mode, it allows the user to perform local operations on the cached data and propagates the results to other computers when it is next connects to the network. It cannot, however, provide access to data that are not cached. When it is weakly-connected, it loads the cache with data not stored in it, but does not immediately propagate changes to other computers to save on communication cost. It logs (and compresses) local changes and uses "trickle re-integration" to asynchronously send them in batch to the server. Moreover, it does not receive updates made by strongly-connected computers, thereby not impeding their progress. It receives remote changes when it is next strongly connected. (Why not receive them in trickle reintegration coming the other way?)

Coda has experimented with several approaches to merging cached and server copies of the file system. In an early version, it automatically merged certain kinds of changes. For instance, like Locus, it automatically merged concurrent insertions of new objects in a directory. For other changes, it required manual merging. For instance, if two users concurrently change the same file, it required them to manually merge the changes. However, this approach was too conservative when the users had changed independent parts of the file. some changes were automatically res Therefore, Coda now allows a programmer to provide an application-specific program for merging versions of a file. For instance, a programmer of a calendar application can provide a program for merging concurrent operations to a calendar file based on the slots that were modified - if the slots are different then it

combines the two changes, otherwise it makes an arbitrary choice or asks for user intervention. Thus, unlike Notes, Coda gives the application programmer control over how merging is done. Because of the large granularity of the data it manages (files/directories), there is more need for this feature in Coda.

Another difference is that it uses a more centralized approach for maintaining consistency. Data are stored in a (possibly replicated) server and caches of these data are stored in local clients. When a strongly-connected client invokes a file operation, it makes a call in the server to update the server copy of the file. The server, in turn, propagates the results in all strongly-connected clients by invoking callbacks in them. When a disconnected or weakly-connected wishes to integrate, its cache is merged with the server copy, and any updates to the server copy are transmitted to other strongly-connected clients through callbacks. Thus, clients do not do pairwise merging as in Notes - instead they merge with each other through the server(s). Replicated servers maintain consistency using another algorithm that assumes strong-connection among them.

Example: It is programmed as in the file case except that the programmer writes an application-specific merge procedure, which can, for instance, concatenate the conflicting strings, prefixing each string with the name of the user who entered it. The behavior of the application would be different in that it users would be able to use it in the disconnected and weakly connected modes. Of course, in the disconnected mode, they would not receive or transmit updates; while in the weakly-connected mode they would not receive updates, and local changes would be trickled asynchronously in batches to the strongly-connected users.

1.3 Distributed Communication

1.3.1 *Interprocess Communication.* Instead of communicating through a repository, user processes can directly exchange information with each other using some form of interprocess communication (IPC). Several forms of interprocess communication have been developed ranging from simple message passing to remote procedure call. We do not distinguish among them here unless necessary. We assume that they enable a process to communicate data to another process, allow a process to declare its intention to get data from multiple processes, and notify a receiving process when the data arrives, either by unblocking the receiver if it is waiting or invoking an asynchronous callback, if it is not.

Interprocess communication allows programmers to implement collaborative applications in arbitrary ways. There are two main approaches we can use when trying to code such applications:

—*Replication*: Implement as much of the application functionality as possible at the local workstation of each user, thereby creating an application *replica* at each site. It is not possible to implement the complete functionality at the local workstation since linkage between users implies that there has to be some communication among their replicas. Moreover, these are not true "replicas" as as the linkage between users may not be WYSIWIS.

—*Centralization*: Implement as much of the application functionality as possible in a single, *master*, process executing at some central site. It is not possible to implement all of the application functionality in a central site, since some code

must be executed at the local site to display output on the local workstation and receive input from it. In general, the extent of this code depends on the I/O package used by the application programmer to (e.g virtual terminal package, curses library, window-system, toolkit) interact with a user. Assuming that the I/O package allows a process to interact with a single user, we would have to implement a separate *I/O manager*, for each user, which would interact with the user through the I/O package. It would receive input from I/O package and transmit it to the master; and, conversely, receive output from the master, and give it to the I/O package.

In both cases, we would need a *session manager* responsible for allowing users to dynamically join and leave the conference, and a *name server* responsible for associating session managers with session names. Moreover, we would user IPC to communicate among the various application processes.

We will compare these two approaches in depth later - for now it is enough to assume that the centralized application is easier to implement while the replicated application gives better interactive performance by localizing the processing of operations. Here we present them mainly to show some of the ways in which IPC can be used in the construction of a collaborative applications.

Example-Replicated Case: As in the previous cases, a process or replica is created at the local workstation of each user, which displays the greeting string and allows the user to edit it. Also we create a session manager, which creates the session and keeps track of all the replicas and the current value of the greeting.

A user creates a new session by invoking the session manager, giving it the session name:

```
session -name ourHelloWorld
```

The session manager registers under the session name with a name server. When a user joins the session:

```
helloWorld -join ourHelloWorld
```

the new replica uses the name server to lookup the session manager, registers itself with the session manager, looks up the current greeting and replicas registered with the session manager, and informs the replicas that it has joined the session. On each keystroke, a replica uses the IPC mechanism to send the input character to all other replicas and the session manager, which immediately modify their local strings and update the screens. (Could we have done without the session manager process?)

Example-Centralized Case: A single master process is started at some central site:

```
helloWorld -name ourHelloWorld
```

which creates and initializes a single copy of the string and registers itself with a name server. To join the session, a user invokes an I/O manager:

```
join -name ourHelloWorld
```

which uses the name server to contact the master and registers itself with the master. The master processes input messages received from different I/O agents in the order in which it receives them, updates the string, and broadcasts the new value to all of the agents.

An IPC system does not have the drawback of disk-resident data. Since we assume that it notifies processes when messages arrive, w it does not have the problem of polling. If it supports communication of only simple types such as strings, then it does suffer from the impedance mismatch problem, since structured types must be converted to/from the simple types. Some IPC systems do allow processes to exchange values of arbitrary types [?], doing automatic marshalling and unmarshalling of data, and thus, do not suffer from this problem.

On the other hand, these benefits come at the cost of several drawbacks: An IPC mechanism does not know which parts of the communicating processes' state are shared or persistent data structures and simply provides a high-level scheme for communicating information. As a result, it cannot automatically offer persistence, concurrency control, access control, versioning, merging, or querying, which have to be implemented by the application programmer. It also does not automate session naming, which must be implemented by a separate name server. Another important drawback of this mechanism is that it require a process interacting with a user to be *agent aware*, that is, aware of the identities of processes (agents) interacting with other users. In the centralized case, the master process is aware of all the I/O agents, while in the replicated process, each replica is aware of other replicas. As a result, the programmer of a process must write code that reacts to a new user joining and leaving a session. The repository-based approaches have none of these drawbacks.

1.3.2 *ISIS: Process Groups and Causal Multicast.* ISIS [?] illustrates how the peer awareness problem can be addressed. The system can provide the abstraction of a *process group*, which processes can join or leave; and allows a process to multicast a message to a whole process group. The tasks of managing memberships of process groups and multicasting to them, thus, is the task of the system, thereby making individual processes peer unaware.

Example-Replicated Case: As in the previous cases, we create a central session manager and replica on the workstation of each user. Instead of expliciting sending a new string value to each of the other user processes, a replica sends the message to a process group including the session manager and all replicas. The session manager creates the process group and initializes the string, each replica contacts it for the process group id and the current value of the string.

Example-Centralized Case: The single master process now creates a process group including all I/O agents and multicasts it output to this group.

Consider a tricky issue in message delivery that we have so far ignored. Suppose a process receives two messages, m1 and m2, that are *causally related*, that is, one of these messages, m2, would not have been sent had the other message, m1, not been sent. In our example, that may happen if m1 initializes an erroneous string (`insertAt: 1 str: "helo world""`) and `m2` contains an edit to the string (`insertAt: 3 str: "l"`). To preserve the semantics of the interaction, we would want the IPC mechanism to ensure that the cause (m1) is received

at each process before the effect (m2).

Of course, a general purpose IPC mechanism does not know what the messages are about, so it can never know the causality relationship among the messages. But it can take a conservative approach to ordering messages by making two assumptions: (1) If a process sends two messages, m1 and m2, in succession, then the first message, m1, is a cause of the second, m2. (2) If a process sends a message, m2, after receiving a message m1, from another message, then message received, m1, is the cause of the message sent, m2. Stream-based protocols such as TCP/IP provide support for the first, intra-process, ordering. They are sufficient to support applications such as the centralized example above, where only one process is multicasting to the group. ISIS takes this idea a step further by also supporting the second, inter-process, ordering of messages, thereby supporting applications such as the replicated example above, where more than one process multicasts messages. For this reason, the multicast it supports is termed as *causal multicast*.

Causal multicast ensures that each process in a multicast group receives a cause before an effect. However, it does not ensure that each process in the group receives the set of multicast messages in the same order. In particular, two messages are multicast concurrently (that is, one message is not a cause of the other), then members of the multicast group may receive them in different orders. Continuing with the repl, if two users concurrently change the string to "hello world" and "goodbye world" respectively, then some users would see the first one as the final value and some the second one. Thus, causal multicast suffices as long the application ensures (by providing an appropriate concurrency control protocol) that concurrent messages never conflict with each other. For applications that cannot provide this guarantee, ISIS also provides a stronger version of causal multicast, called atomic multicast, which ensures processes in a process group see multicast messages in the same order. In general, an atomic multicast may not be causal, though in the case of ISIS it is.

ISIS was designed mainly to support replication of applications for fault tolerance. However, as we have seen above, it can also support replication for good interactive performance in a collaborative applications. For this reason, some collaborative applications such as the MASSIVE [?] VR system have used it for multicasting messages. However, this multicast is not ideal for all collaborative applications, for two reasons.

First, causal multicast may be too conservative for messages that commute with each other. Consider a replicated implementation of a GROVE-like structured outline in which the replicas exchange fine-grain updates with each other, that is, describe an edit in terms of the smallest structure that changed rather than the whole buffer. In this situation, changes to different parts of the structure would commute with each other. For instance, the edit, `section: 1 insertAt: 1 str: "hello world"` would commute with `section: 1 insertAt: 1 str: "goodbye world"` . Even if the first change caused the other, there is no harm done in processing it after the second one. Thus there is no advantage in using ISIS's implementation of causal multicast. On the other hand, there is a disadvantage in using it, since it delays a message until all of its predecessors are received, thereby giving poorer response.

More important, causal multicast is too liberal when concurrent messages conflict

with each other, that is, do not commute with each other. Such conflicts would occur in an application such as Grove that does not provide concurrency control, and even in applications that provide fine-grained concurrency control but exchange large-grained updates. Consider again the replicated implementation of a Grove-like editor but this time assume that changes are communicated in terms of the whole buffer. In this case, even if the editor ensures that two users do not concurrently edit the same section, since edits that changes the buffer length (e.g insertAt 1 str: "hello world, insertAt: 100 str: "goodbye world") have conflicts. In cases such as these, causal multicast is not sufficient, and what we want is atomic multicast. But implementations of atomic multicast must serialize concurrent messages through a central process, thereby reducing the performance benefits of replication. For this reason, some applications such as Grove use application-specific optimistic schemes for ensuring consistency. Centralization is less of an issue in systems in which replication is introduced for fault-tolerance rather than performance.

1.3.3 *MBone: Network Multicast.* In ISIS, Multicast is supported in the library layer. It can be supported in lower-level layers to give better performance. In particular, if it is supported in the kernel layer, then it is sufficient to make a single kernel call to multicast a message to a process group; and if it is supported in the network layer, then it may be sufficient to send a single message along a network link to multicast the message to multiple processes connected by the link. For this reason MBone [?] (Multicast backbone) provides network-level support for multicast. Multicast messages are targeted at special IP addresses that are associated with groups of hosts.

The saving in kernel calls and network messages is particularly important for scalable video conferencing, which must send large number of large messages to a large number of sites. MBone has been designed for such applications and includes an interesting set of applications for making distributed presentations including audio and conferencing tools and a whiteboard application. All of these tools are scalable in that they have been used by hundreds of users.

Reliability in scalable multicasts can be a tricky issue. In many reliable unicast protocols such as TCP/IP, the sender is responsible for ensuring that the message is delivered to the receiver by waiting for a positive ack from the latter. This is not a good approach in scalable multicast, since it puts an undue burden on the sending site, which must wait for a large number of positive acks. It is better to use to make each receiver responsible for ensuring reliability by sending negative acks in case messages are lost.

Wb [?], an Mbone shared whiteboard, illustrates another use for negative acks in a replicated implementation of a collaborative application. It uses them for supporting dynamic addition of users in an ongoing collaborative session: the replicas created for these users simply send negative acks to receive the current state of the conference. They do not have to, as in our replicated implementation of the example, send application-specific messages requesting the current conference state.

1.3.4 *X: Network I/O.* Consider now the centralized I/O agent-based implementation of a collaborative application. There is an unnecessary level of indirection here, the I/O agent does no more than forward requests to the local I/O package. Moreover, the master process and I/O managers are responsible for encoding the

I/O calls as messages of some IPC mechanism. This indirection and use of an IPC mechanism would not be necessary if the application could use (directly or indirectly through a toolkit) an I/O package that provided a *network interface*, that is, allowed a remote application to directly invoke its functions, and if an application could concurrently interact with multiple I/O packages on different workstations.

The X window system [?] is perhaps the first I/O package to offer such an interface. An X client can open connections to one or more *X servers*. Each X server runs on a workstation and allows its clients to perform window-based I/O on that workstation. It receives messages from X clients in the form of *X requests* and sends messages to them in the form of *X events*. An X request from a client asks the server to create a window, map it on the screen, draw text in it, or perform some other output task on the screen. An X event to a client informs the client about keyboard presses, mouse movements, and other input actions of the user. The X window system is "almost" portable in that it has been accepted by a large number of vendors. "Almost" because Microsoft decided not to adopt it.

Centralized Example: As before, we create a master process, which now interacts with the users of the collaborative application by directly communicating with the X servers managing their workstations. Instead of invoking I/O managers, they invoke special join processes whose sole job is to inform the master process about the identities of their X servers.

This implementation is certainly easier to code and more efficient than the centralized implementation we saw before. However, it is not without disadvantages. The application programmer must still implement a program used to join the session, which involves using some IPC mechanism to communicate with the master program. More important, a larger amount of information is communicated from the master to the workstations, since, in comparison to high-level requests/events, low-level requests/events are typically larger in number and size. X was designed for high-speed local area networks, and is thus not ideal for wide-area, real-time collaboration.

1.3.5 *Web Browsers: High-Level Network I/O.* So what we would like is a higher-level I/O package that, like X, is portable and offers a network interface. A Web browser [?], in fact, is such package. It is not a typical "I/O package" in that a remote process does not execute output procedures to display information, instead it sends *documents* or *pages* describing the display layout in HTML (HyperText Markup Language). An HTML document is stored at any internet site and is managed by an HTTP (Hyper Text Transfer Protocol) server at that site. It is associated with an internet-wide name called URL (Universal Resource Locator), which identifies the server managing it. A browser can request a document from any server by sending it the URL of the document.

An HTML document may include a directive to periodically load a particular page. More important, it may display not only hyperlinked static text, but also dynamic forms for sending input to remote procedures. A form contains interactors (or widgets) such as text items, menu choices, sliders, and nested forms for entering input values. It is also associated with a *remote callback* called a CGI (Common Gateway Interface) script. The script is invoked when the user submits the form and receives as parameters the input values entered in the form. In response, it

sends the browser an HTML document describing the new display. A CGI script is associated with a URL and is executed by the HTTP server that manages it.

There are, of course, many flavours of browsers and servers, but all of them must support the currently accepted HTML and HTTP, respectively, though they can provide their own extensions. Of particular interest to use are browsers that provide a CCI (Client Communication Interface). Such an interface, typically, allows some other, possibly local, process to both monitor and invoke actions of the browser such as loading of a new page.

Centralized Example: We define an HTML document for the application, whose URL serves as the session name. Users join the session by loading this document in their browsers. The document displays a form containing a text item showing the greeting. A users may edit and submit the form to change the greeting. The CGI script invoked to process the form changes the greeting field in the HTML document and asks the browser to reload this page.

How do other users view a change to the greeting? The simplest approach is for user to explicitly reload the document to see the current value. Instead of making the user poll for the new value, we can specify in the document that the browser should automatically poll for it. These are essentially the user-interfaces we saw in the repository-based solutions. Implementing a non-polling interface is more difficult but possible if the browser provides a CCI interface. At the server site, we create a session process responsible for notifying browsers of changes to the HTML document and monitoring its loads. (If CCI allows only local processes to talk to a browser, then we need to create, for each browser, an additional local process connecting it to the session manager.) as that user's surrogate. When a browser first loads the HTML document, the session manager registers it in a directory. The CGI script now, in addition to changing the document, also informs the session manager about the change, which then asks all the registered browsers to reload the page.

This approach to implementing a centralized application offers several advantages over the X-based one. A web browser does higher-level processing of I/O, thereby requiring less communication with the remote master process. In our example, it provides local processing of scrolling, backspace, and other browser commands, sending and receiving only the greeting string from the master. Moreover, support for URLs automates session naming. The browser also provides a standard user-interface for interacting with applications. Furthermore, we can use a single set of physical windows created by a particular instance of the browser to interact with and navigate among multiple application. Perhaps most important, Web browsers are more portable than X in that everyone (including Microsoft!) has ported it. A directory of HTML documents can also be associated with access lists describing the list of authorized users and their passwords. Since an HTML document is stored as a file in the native OS, all file-based collaboration functions such as concurrency control can also be used, but these are OS-specific and thus would not be portable. Thus, we get can combine some of the benefits of schemes based on repositories and distributed communication.

On the other hand, this approach is far from ideal for developing general collaborative applications. The access-control provided by it is very coarse-grained, it only decides who can access the document without distinguishing between different kinds

of accesses such as read and write. Finer-grained control must be implemented by the CGI scripts. More important, users cannot receive incremental feedback to (local or remote) input, since a form is processed when it is submitted and not as users make changes to it. For instance, in our application, users cannot see incremental changes to the greeting made by their collaborators. Conversely, this approach does not work well when a CGI script needs to make an incremental change in the display, since a whole new document describing the complete display must be sent to the browser and processed by it, which makes the interaction slower and causes the display to flicker. (Current browsers do not calculate the diffs between the original and new dispays). Since a browser is a high-level I/O package, it cannot create arbitrary user interfaces. The non-polling solutions are far from ideal since the session manager and user agents must be coded using some IPC mechanism, and perhaps more important, a large number of processes are involved in the communication of information from a local user to a remote user: the local browsers, the HTTP server that receives the CGI request, the copy of the server forked to execute the CGI script, the session manager, and finally, the remote browser. Moreover, if the CCI supports communication among local processes, then an additional local process would be involved. In the X-based solution, only three processes are involved: the host and the local and remote X servers. If the browsers are implemented on top of X, then the two X servers would still be involved in the communication.

1.3.6 *Java: Code Downloading + Remote Method Invocation.* One way to combine the benefits of the X- and Web- based approaches is offered by the Java object-oriented programming language. An HTML document can refer to a Java program called a Java applet that is stored at the site of the server managing the document. When a browser fetches the document, it also dynamically loads the applet code and executes it. The applet code executes within the context of the browser, that is, uses, for I/O, a section of the space allocated by the browser to display the document. Thus, like an X client, a Java applet is programmer-defined and can create an arbitrary user interface, and like a Web browser, it executes at the local site. This idea of dynamically downloading code in an I/O package is not new, and was pioneered (by the inventor of Java) in the Network Window System (NeWS) [?]. Dynamic downloading of code was also supported in Computational Mail, as we saw earlier. As we saw in computational mail, downloading and executing code from an arbitrary site raises security issues. Java addresses them by allowing a Java applet to accesses files at and send messages to only the site from which it was downloaded.

It makes sense to not only download code in an HTML browser but also an HTTP server, as illustrated by the Sun Java HTTP server, wich is written in Java. The server is composed of smaller modules, called *servlets*, each of which serves a particular kind of browser request. Like applets, user-defined servlets may be dynamically loaded into the server. When a browser request comes in, the server determines the servlet that must process it. If the servlet is trusted, then it executes in the same OS process as the server, but in a separate Java thread; otherwise the it executes in a separate thread group (OS process?).

A new servlet thread does not have to be forked on each request - once created, it keeps processing requests until there are no more requests, after which it may

be terminated. Moreover, unlike an HTTP server, it can (since it is user-defined) create state that persists across multiple browser requests. To allow a browser to name this state, a servlet, like an HTML document or a cgi script, is associated with a URL, which can be used by the browser to invoke a service in it.

Java programs do not have to be dynamically loaded and executed within the context of a Web browser or server. Local Java programs can be executed as *standalone* Java processes, which, like processes written in any other programming language, have all rights of the users who create them.

Java also provides a high-level RPC-based mechanism for communication among arbitrary processes. Traditional RPC executes a global procedure of a remote process and (implicitly or explicitly) identifies the target process. Java extends this idea by supporting *remote method invocation*, that is, direct invocation of a method of an object in a remote process. Instead of identifying the process, the invocation presents a *remote object reference* to directly identify the target object, using the same syntax as a local invocation. A process can register an object it wishes to export with a Java-provided name server, called a *registry*, by associating it with a string name, which can be looked up by some other process to receive a remote reference to the object.

How should parameters [2] of remote invocations be handled? Traditional implementations of remote procedure calls do not allow address parameters and make remote copies of value parameters. Some implementations do allow address parameters, but make copies of the referants at the remote site and return pointers to these copies. As a result, dereferencing the address refers to the local copy and not the original remote object. Java extends these semantics by sending copies of parameters that do not implement the Java *remote* interface and remote references to parameters that do.) The remote references can be dereferenced to access the remote objects and not local copies.

Another interesting feature of Java is that if a process does not have the class of a (copy of or reference to a) remote object it receives from another process, Java will dynamically download the class into the process, much as it dynamically downloads applets into a Web browser. Since Java classes are also objects, they can probably be transmitted explicitly in parameters of methods.

Example: Servlet-based Centralized Version:

This is like the above version, except that we replace the session manager and CGI scipt with a Java servlet, and instead of invoking the CGI script, the browser now invokes the servlet. Not sure if form data can be passed to a servlet.

Example: Applet-based Centralized Version: As in the Web-based version, we define an HTML document for the session, which users load in their browsers to join the session. Instead of containing a form, it contains a Java applet, which, like the form, creates a local copy of the greeting, and displays it the user in an editable text widget. We also create a central stand-alone master Java process at the site from which the document is loaded. As before, it keeps track of the current value of the greeting and the users that have joined the session. It exports to the registry on its machine a session object defining methods for joining/leaving the session. The applet uses the session object objects to register its local copy of the

[2] We consider return values to be special cases of parameters

greeting with the master and get a reference to the global copy. Both copies define update methods, which are used to keep them consistent. Thus, on each character typed by the user, the applet invokes the update method in the global-greeting object with the local value of the greeting. This update method, in turn, invokes the update methods of all local-greeting objects. (Later we will see a more modular Java-based implementation based on the Observer/Observable interface).

Example:Applet and Servlet-based Centralized Version:

This is like the above version, except that we replace the standalone Java session manager with a Java servlet, and the local Java applets communicate with this session manager through the server. Not sure if applets can invoke servlet functions.

Example: Replicated Version: It is also possible to use Java to build a replicated version, but we cannot use Java applets since they can communicate only with processes at the HTTP server site. The implementation is similar to the one we saw under interprocess communication except that the local replicas use the predefined Java registry and Java remote method invocation for IPC. Thus, the session manager registers a session object with its registry, and the session manager and the replicas define local copies of the greeting that provide update methods. The replicas and the session manager to establish connections among their copies of the greeting objects and use the update methods to keep them consistent.

With its support for remote object references, Java can be considered as a language supporting *distributed shared memory*. However, a remote reference is not a first-class reference in that it can not be used to access instance variables of the referant or invoke methods in any interface other than the Java remote interface. (Perhaps with the Java reflection model this will not be an issue since we could invoke reflection methods to retrieve arbitrary fields and invoke arbitrary methods.)

1.3.7 *Obliq: Network Scopes.* Let us now look at a similar language, Obliq [?], that is closer to supporting distributed shared memory. In Obliq, programs have *network scopes*, that is, scopes extending to multiple address spaces/hosts. An Obliq program starts of in some address space and can send portions of its code to other address spaces, much as an HTTP server sends applets to HTML browsers. The code, however, can be bound to variables at the original site, and these bindings are preserved when it executes at the remote site, that is, they become remote references to the values at the original site. As in Java, a distributed name space can be implemented through a registry - network scopes provide another, more convenient, mechanism for implementing suc spaces that use a registry only for bootstrapping purposes.

Also, as in Java, objects may be transmitted to a remote address space as parameters and return values of remote method invocations. Obliq distinguishes between *immutable* entities such as procedures and *mutable* entities such as objects. How an entity is sent to a remote address space depends on whether it mutable or immutable. If it is mutable, then a remote reference to it is sent; otherwise a "copy" of it is sent, which may contain remote references to the original site. Obliq constructs the copy of an object as follows: It determines the graph of nodes reachable from the object, creates copies of all the immutable nodes in the graph and the links to them (preserving cycles), and replaces local reference to mutable objects from the copied nodes with corresponding remote references. Obliq also allows a process to

that a copy of a mutable object be sent.

To illustrate how Obliq may be used to create collaborative programs, and also the nature of higher-level collaboration infrastructures, consider Visual Obliq [?], an Obliq library for creating a replicated collaborative application. Each replica of the application creates one or more top-level windows, called forms, for its user. Visual Obliq allows a user to interactively specify the appearance of forms and attach Obliq callback procedures to their fields. The library generates Obliq form classes (with associated form constructors) that implement the form user-interfaces specified by the user, and also an Obliq session-constructor object for adding new users. In addition, for each form class, it creates a global session array that keeps track of the instances of the form created for different users. The application programmer can augment/modify/subclass the generated code.

Let us say we have created a `helloWorld` program using Visual Obliq. A user, at machine `jeeves.cs.unc.edu`, can create a new instance of this program by typing:

```
visobliq -run helloWord
```

A new Obliq process is created, containing all the global entities defined by the program such as the session constructor and the session arrays for the different forms. VisualObliq registers the session-constructor under the name "helloWorld" with a local registry. It also executes this constructor in the new process, which instantiates each form of the application by calling its form-constructor procedure, and adds references to each of these forms to its session array. Thus, this instance of the program is a "server", in that creates the global session data, and a "client", in that it creates local data to interacts with a user and refers to the global data.

Now another user, say at `emsworth.cs.unc.edu` can join this session by typing:

```
visobliq -join helloWorld@jeeves.cs.unc.edu
```

VisObliq creates a new client at the local site, retrieves a reference to the session-constructor from the registry at `jeeves.cs.unc.edu` and invokes the constructor. The session-constructor fetches copies of all the form-constructor procedures from the server, invokes them to create new instances of these forms, attaches local callback procedures to them, and adds (references to) these instances to the global session arrays stored in the server. Notice that the session arrays do not have to stored in a registry, they are automatically visible to the procedures copied from the server.

When a user enters information in a form, VisObliq invokes appropriate programmer-defined callbacks procedures attached to the form. A callback can update not only the local form instance but also the remote instances stored in the session arrays. For instance, in our example, when a user changes the local greeting, the callback can access access the references to the remote form instances replicas and update their greetings fields.

We have seen above how a replicated version of our example can be created using Obliq and VisualObliq. [3] It does not support centralized applications, but Obliq can be used directly to create such applications. A single copy of each application

[3] VisualObliq also provides a mechanism for inviting users to a session rather than requiring them to use the session name to autonomously join the session.

form would be stored in the master process, which would invoke remote methods in the I/O managers to update their displays.

Obliq can be used not only to create migrating applications or *agents* [?]. A migrating agent hops from site to site, collecting and processing information at each site. When it hops to a new site, it brings with it a *suitcase* containing information it has collected from the previous sites, receives a *briefing* from the new site, and executes a procedure parameterized by these two pieces of information. In fact, we can formally define the notion of hopping agents in Obliq [?]:

```
let rec agent =
    proc(suitcase, briefing)
        (* work at the current site *)
        ....
        hop (nextSite, agent, suitcase);
    end;
```

Here nextSite is a remote reference to a server at the next site for the agent.

The hop procedure is implemented not as a language primitive but, in fact, using existing Obliq constructs:

```
let hop =
    proc (agentServer, agent, suitcase)
        agentServer (
            proc(briefing)
                fork(
                    proc()
                        agent(copy(suitcase), briefing);
                    end);
                ok
            end);
    end;
```

The hop procedure invokes the remote agentServer, passing it a procedure as an argument. Since a procedure is an immutable object, Obliq passes a copy of the procedure (and the agent procedure referenced by it) with a remote references to the local mutable `suitcase` object bound to it. On receiving this copy, the remote server executes it after passing it the local briefing. This procedure forks a new process, executes another procedure, while the parent process terminates and unblocks the caller. Meanwhile, the child process retrieves a copy of the suitcase and passes it, along with the briefing, to the copy of the agent procedure it received.

Migrating agents could be used for several purposes. They can be used to implement: POLITeam like workflow, taking a form to each user in the routing slip; computational mail, executing a program mailed to a user at the local site; and follow a user from site to site. If an agent is interactive, as in the examples above, it puts the user-interface state in the suitcase, and recreates it on the local display

received as a briefing.

Currently Obliq does not support migration of multiuser agents since it does not maintain connectivity with distributed processes as the agent moves.

1.3.8 *Sumatra: Mobile Objects.* Migration also in Java though RMI.

Not efficient Not true migration - no redirection of references.

execution engines - Java interpreters.

Object group - I/O objects not moved but retargetted (suitcase.)

forwarding adrress left and object-moved exception.

Can also create a new thread at remote site a la Obliq. rexec main method. Non blocking.

Can also migrate thread. - go instruction. stack sent and non object groups. remote references to object groups.

Code?

send thread code also: expensive, no retargetting.

assume same program all sites- too conservative.

send code from go to all visible gos - gos on control path. compiler support.

send code from go to enabled gos - uses dynamic stack info.

Policy? Application-defined (mobili-aware app.)

Factors:

spatial variations - links with diff. speeds. population variations - users join/leave. one time placement. temporal variations: link speed changes.

Experiments show: large spatial variations. US hosts: 15 ms to 863ms. Non Us hosts: 84ms to 4000ms.

Time variation: short-term small jitter. occasional sharp jumps over short time intervals. in small windows (10 mins): 70-90over one day, mode variation: US hosts: 500 ms, Non US: 5750ms.

Application can request monitoring of link with specified frequency. Mode of last 10 min window written in shared memory.

Example application:

Centralized: Master migrates. Thread or obj group? remote method vs remote thread execution.

replicated: session manager. thread or obj group?

Policy: adapttalk - minimize max response time. tradeoff between stability and good reponse. multiple rounds - decison cycle (50n). win threshold for reponse (25n). loss threshold (12n).

consider all kinds of variations.

1.3.9 *Rover: Disconnected Replicated Objects.* While we did see how a repository can offer disconnected or weakly connected operation, we have yet seen the same for distributed communication. Rover [**?**], illustrates a solution for systems supporting communication of objects. Rover can be considered a cross between Obliq and Coda: like Obliq it supports user-defined objects and allows them to migrate or "rove" among computers; and like Coda, it supports strongly-connected, weakly-connected, and disconnected operation.

Rover supports the notion of a *relocatable dynamic object* - an object that is managed by some server, called the home server of the object, from where it can be dynamically relocated to the address space of a client of the server that has

established a *session* with it. The relocation occurs when the client explicitly *imports* a copy of the object. The client can now *invoke* one or more operations on the local copy, and at some later time, explicitly *export* the changes back to the server copy. Thus, this model is based on the explicit check-in/check-out concept of version control systems rather than the automatic caching scheme of Coda. As in Coda, the client can explicitly *prefetch* objects it expects to access (import).

The prefetch, import, and export are executed as remote procedure calls (RPCs) invoked by the client in the server. Unlike traditional systems, Rover does not invoke an RPC synchronously, blocking the invoker until the procedure is executed at the remote site and returned results. Instead, it supports *Queued RPC*, that is, unblocks the client, queues the RPC in a stable log at the client site, asynchronously invokes the operation in the server, and communicates the results back to the client by invoking a callback.

As in Coda, the scheduled used to process the client operations depends on the connectivity to the server. In the strongly-connected case, each QRPC can be sent immediately to the server, in the disconnected case, the RPCs are sent when connection is next established, and in the weakly-connected case, the cost of communication and the nature of the queued operation influences the schedule. Queued RPCs are not necessarily executed in the order in which they are queued. They can be reordered to let the more urgent operations be executed earlier. A client associates a QRPC with an urgency or QOS (Quality of Service) parameter which is used to determine how soon that call is invoked. The authors of QRPC compare this parameter to the different degrees of urgency we associate with postal mail - ordinary, two-day express, one-day express, and so on. Thus, an import or fetch RPC can be given higher priority than an export RPC to simulate the Coda weak-connection policy.

Despite its name, an RDO, does not truly relocate or migrate to the client when it is imported. In fact, it is *shared* with the client - the import operation creates a copy of the object, leaving the original copy to be concurrently modified by the server or imported in other address spaces.

Rover provides several policies to support consistency among the copies of the object. It allows, copies of the object to be modified concurrently, using a type-specific optimistic concurrency control mechanism to check for conflicts. We will look at type-specific CC in more detail later; to give an example – a Queue-specific CC can allow concurrent queuing operations. In case of conflicts, a Coda-like conflict resolution procedure can be used to resolve the conflicts - instead of working on files, it would work on objects. To eliminate such conflicts, a shared object can be locked by a client, thereby preventing the server and other clients from modifying it. However, as we saw earlier, this policy is suitable only if the the lock holder is strongly connected. A disconnected or weakly-connected client can reduce the chance of conflicts by specifying, for a local object, the following consistency options:

—*Uncacheable*: do not allow the object to be imported. We might associate this option with an exported object that has not reached the server.

—*Immutable*: the object is guaranteed to be immutable, so write methods must not be allowed on the object. Rover, thus, must distinguish between read and write methods on an object.

—*Verify-before-use*: Before invoking a method on a object, check if the object is still upto date. (What is the difference between locking an object and verifying before use?)

—*Verify-if-service-is-accessible*: Same as before except that the verification is not done if the cost of communicating with the server is high (more than some specified threshold).

—*Expires-after-date*: No operations can be performed on the object after a certain date.

—*Service callback*: Notify the client if the server object changes. Not sure what happens if the client is disconnected - doubt that server keeps track of callbacks it must invoke.

Some of these options are orthogonal, and users must be able to specify multiple options simultaneously.

As mentioned before, callbacks are also used to inform applications about the completion import and export operations issued by them. When an application invokes an import operation, it specifies the session id, the object id, QOS specification, and a callback with the operation. Rover queues an RPC for the operation and sends it to the server when an appropriate level of connectivity is established with the server. The server fetches the object and sends it to the client, which deletes the queued RPC and invokes the application callback.

Similarly, when a client exports an object, it specifies the session and object id, the QOS priority, and a callback. All operations invoked on the object since it was imported are sent as QRPCs. The server executes them if there are no conflicts or if conflicts can be resolved automatically (based on an application-provided procedure), sends back to the client an indication of whether the object was updated or an unresolvable conflict was detected. The client removes the QRPCs from the log, and invokes the application callback with the values returned by the server. When a method is invoked on a cached object, it is marked as *tentatively committed*, and once it is successfully changed by the server, it is marked as *committed*. Applications can inform users about the commitment status of objects so that they know which values they can depend on. The RPC request and response are *split* in that they can be sent on communication links with different properties.

As in Obliq, the importing of an object can trigger the creation of a new, "agent", thread for executing operations on the object. As in Coda, logged QRPCs can be compressed, but Rover excepts application programmers to provide the compression procedures. Rover supports both the interactive TCP/IP and batch SMTP protocols for communication among clients and servers, the choice between them is based on the QOS desired. Moreover, instead of using TCP/IP directly, a client can instead use the higher level HTTP protocol. An object is identified by a URN (Uniform Resource Name), which is built on top of a URL.

Rover has been used to built several applications, including versions of a mail reader, calendar scheduler, and a Web browser that do not require strong connection to the data they access. (The current Web browsers allow users to access cached data when they are disconnected, what additional facilities can a Rover-based Web browser provide?).

How well does Rover perform compared to the traditional communication model,

wherein no data are cached in the local client, and it manipulates objects by invoking operations in the server. Rover can be compared with the traditional model only when there is some connectivity between the client and server, in case of disconnection, it clearly offers access where the traditional model would deny it. In the connected case, the cost of a QRPC is much more than cost of traditional communication because of the cost of logging the RPC in stable log. A TCP/IP-based "null" RPC (290-byte request with a 5-byte reply) takes 47 milliseconds, over 10 Mbit/sec Ethernet, whereas using TCP/IP directly takes 8 milliseconds to communicate the same data. Writing a log entry takes 37 milliseconds, which explains the difference.

However, every RPC in the traditional model does not have to be correspond to a QRPC in the Rover model. If an object has been imported in an application, then it corresponds to a local operation in the application, which takes 3.2 msec for a null RPC. We need to also consider the time to import/export the object. The time to export the object from a client to a server machine is 59 msec. Once an object has been exported, it can remain in the client machine, in case the application imports it again. The cost of retrieving a null object into the address space of the client from the address space of the system is 7.5 msec, which is the cost of a local null RPC. LRPCs are not implemented efficiently in Rover - hence the high cost. (All numbers above are for the ethernet case.) The cost of exporting and importing objects is, of course, amortized over multiple object invocations.

So how effective is the amortization in real applications. The authors of Rover did several additional experiments with the applications they build to see overall interaction times for some user-level actions. For instance, they considered the cost of reading eight messages in a folder whose total size was 65.4 Kbytes. They considered several cases, including:

—Strongly-connected remote exmh: remote exmh connected to local X server through 10 MB/sec Ethernet. Time taken: 0:55 (minutes:seconds).

—Weakly-connected (14.4 Kbit/sec) remote exmh: remote exmh connected to local X server through 14.4 Kbit/sec connection. Time taken: 9:08.

—Weakly-connected, local exmh: local exmh connected to remote file system through NFS running over 14.4 Kbit/sec SLIP connection. Time taken: 2:36.

—Weakly-connected, local Rover exmh: Rover exmh, assuming that folder is not cached, connected to server through 14.4 Kbit/sec connection: Time taken: 1:34.

—Weakly-connected, local Rover exmh: Rover exmh, assuming that folder is cached, connected to server through 14.4 Kbit/sec connection: Time taken: 1:06.

—Disconnect, local Rover exmh: Rover exmh assuming that folder is cached. Time taken: 1:02.

As can be seen, the cost of caching data is indeed amortized in this case but the strongly-connected exmh gives the best performance.

Example-Replicated: An initialization program stores the greeting in some server and associates it with a URN to be used as the session name. Each user runs an application that uses the URN to import the object, registers a service callback to receive updates, allows the user to modify the string, exports it after each change or when the user saves it - depending on the connectivity, and immediately imports

it after exporting it. The service callback is used to update the string with the new value of the object. If the application is guaranteed to be strongly connected to the server, it can lock/unlock the string when the user starts/finishes editing it. Otherwise it can provide a procedure to resolve the conflicts or let the user do so interactively.

Unlike the other distributed communication schemes we saw above, Rover provides support for consistency management in the form of concurrency control and merging. This is because it combines features of shared repositories and distributed communication, like the former it has a notion of shared data (RDOs), and like the latter, it provides a mechanism for communicating information among different processes (QRPC). Currently, it does not offer several repository functions such as access lists and version trees, but these can be implemented in an object-based repository, as we shall see when we consider object-oriented databases.

Rover's check-in/check-out model also provides support for isolation: The changes to imported objects are not propagated to other users until the object is exported. Moreover, its QRPCs provides some fault tolerance. An RPCs is queued in a stable log, so if the operation to send it to the server fails (because the client was disconnected, for instance) the client will try again until it gets an ack from the server that the RPC was committed.

However, the support for concurrency control, isolation, and fault tolerance are somewhat limited. An application must explicitly import, export, and lock objects, it cannot atomically lock a set of distributed objects, it cannot make certain changes to an object visible before others, and there is no general mechanism for recovering from failures in arbitrary operations.

1.4 Argus: Distributed Transactions

Argus [?], also developed at MIT, is an example of a system that addresses these problems. Like conventional database systems, it supports transactions. Instead of manipulating relational data in some global database, they manipulate user-defined data dispersed across multiple, autonomous "databases" called *guardians*. A guardian is like a process, module, or monitor in that it is associated with its own address space and exports an interface that can be invoked remotely from another address space. It extends the notion of a module in several ways.

It can declare certain variables as *stable* - changes made to these variables are saved on stable storage. Moreover, its types are *atomic* - that is, concurrent accesses to objects of these types are automatically synchronized. Thus, programmers do not have to worry about synchronization. (This is also the case with monitors, how is a guardian different?) Argus also allows a process to group together a series of operations on one or more distributed guardians in a *transaction*. It allows different transactions to be executed concurrently and automatically locks atomic objects to ensures their serialization, that is, their execution is equivalent to a serial schedule. A transaction might *commit* successfully or *abort*. In case the transaction commits, all changes made by the transaction to stable variables are written to stable storage. In case it aborts, all changed stable variables are restored to the version saved in stable storage.

A transaction might be decomposed into one or more *subtransactions*. Like transactions, subtransactions are groups of operations that can execute concurently and

are serialized. However, changes made by them are not written to stable storage unless the top-level transaction commits. Moreover, aborting a subtransaction does not abort the enclosing transaction, which can try an alternative subtransaction.

Finally, a guardian is associated with a background process that executes when no other operation is being executed in it.

Example- Centralized case: Like the standard IPC case except that IPC is encapsulated in transactions. Thus, an I/O manager sends input to the master in a transaction, and a master broadcasts the output in same transaction. As a result, changes to the master are synchronized and a failure to send output to a single I/O module causes the whole transaction to fail.

Example - Replicated case: Like the standard IPC case, except that a replica creates a transaction for processing a series of changes from the user and broadcasting them to other replicas. As a result, if the transaction aborts for any reason, all the replicas are restored to their original state.

Argus was used to implement the CES editor discussed earlier, and the designers of CES, while appreciating its automatic support for nested transactions, noticed an important shortcoming for creating interactive programs: When a transaction is aborted, any I/O performed by it is not undone. To illustrate why this is a problem, consider the replicated implementation above, which creates a transaction to process a series of changes from its user and distribute them to other replicas. To give the user feedback, the transaction updates the local screen in response to each user command. Now if the transaction aborts (because it could not distribute the change to some failed site, for instance) all replicas are restored to their previous state, thereby ignoring all effects of the user input. But the local display is not restored.

CES used an interesting trick to address this problem. Here is my impression regarding how it works: It encapsulated each user-display in a separate guardian and provides operations to update the display buffer. It also defines a stable lock, which is acquired at the start of each operation and released at the end of it. It also defined two non stable variables, `action-count` and `commit-count`, which were incremented at the start and end of each operation. If commit-count was less than action-count and the lock was not busy, then some transaction that updated the display had aborted. So the display could refresh the screen from the guardian that kept its state.

But how does the display discover this condition? Its background process can poll - but CES provides a more efficient solution. The display guardian also defines a trigger-queue, into which the background process enqueues itself. When a transaction executes an operation is the display, it dequeues the background process after acquiring the lock. The background process then blocks itself waiting for the lock. When it acquires the lock, it checks for the abort condition, refreshing the screen in case it holds, and releases the lock. It then goes back and enqueues itself.

There is a timing problem here: Between the time the background process was dequeued and it tested the lock, another process might enter the display and manipulate an invalid display state. Therefore, every operation checks for the abort condition before manipulating the display.

A more elegant solution would have been possible if Argus called programmer-defined abort and and commit handlers for an object when changes to the object

are committed/aborted respectively.

1.5 MVC

What if we want: a) user transparency and b) diff views.

Model, View, Controller. Model ¡- View: addDependent. Model ¡- Model: changed: Aspect, Model ¡- Controler: app-specific. View ¡- Model: update: Aspect. Model ¡ View: app-specific.

Java, similar idea. Observable, Observer.

Now to get distributed protocol. Not so easy - Anshu Problem.

Code application:

Centralized: create central process with multiple views and model.

Replicated: Each replica an observer/dependent of others.

1.6 Rendezvous

Constraints between model and view (dependent).

fonted string.string = model.string

Enforced procedurally in model-aware views and controllers.

What if language offered constraints seprate from depedents.

ALV model in Rendezvous. ALV V = Rendezvous V + C.

one-way, multi-way, two-way constraints.

side effects - for latecomers.

uninitialized vars - data flow approach, waitfor operands.

indirect references - dynamic inheritance, selected objects.

Ultra LWP objects like monitors.

Abstract processes + View processes for fairness.

These enforced

Centralized impl only.

ALV for replicated?

1.6.1 *Clock.*

No side effects.

Pure functional.

Structured objects.

Centralized implem:

root object, Session object, model object, view (component obj)

session predefined: allUsers, userName, userScreen, removeUser, addUser.

root view:

view = Views (map greetingView allUsers)

GreetingView:

view = Window (userName myId, userScreen myId) (Text greeting) (display text, IP address, displayView)

key KeyCode = modifyGreeting KeyCode.

GreetingModel:

greeting = this. modifyGreeting KeyCode = save (new greeting) initially = save "hello world"

Program consists of hierarchical components.

Root view.

View components have display = view + controller.

Model: request method + update methods.

Hierarhical scopes: scopes uses IS-PART-OF relationship. messages can only be sent to parent. constraints flow downwards. so no cycles.

support for transactions: user actions create a series of threads. user actions oredered - orders their threads - orders accesses to resources. server-based locking.

Views execute in local processes.

Distribution schemes:

Naive:

change, notified, request, response.

Request Cache:

cache responses. if cached value not changed - use cache.

server invalidates cache based on dataflow techniques. seven times improvment.

Request Prefetech:

get all invalid values in batch. speedup 1.5 to 5 times.

Request Presend:

Server remembers which values requested by each view. sends these values when it invalidates: 2 speedup.

1.6.2 *MUDS: Room-based session man.*

1.6.3 *Colab Programming Environment: Broadcast Methods.* Replicated WYS-INWYS

1.7 Shared User Interfaces

1.7.1 *GroupKit: Sharedd Environments etc.* broadcast methods -¿ notification.
multicast rpcs.
lexible session mgmt.

1.7.2 *XTV: Shared Windows.* Colab Transparency.

1.7.3 *Concur: Inverted X*

1.7.4 *DistView: Rep. Window System*

1.7.5 *Suite: Parameterized Flex.*

1.7.6 *Feiner's System (Meehan)*

1.7.7 *Cola: Haake (Rusev)*

1.7.8 *Suite: Persistent Unix Objects.* Send to later.

1.7.9 *MMConf: Replicated Windows*

1.7.10 *DistView*

1.8 Multiuser Toolkits

1.8.1 *Sync: Disconnected Operation*

1.8.2 *Brown's Stuff*

1.9 Inter-Application Coordination

1.9.1 *Intermezzo*

1.9.2 *Object-Oriented Database Management System.* Maybe this comes at the very end.

1.9.3 *Oz*

1.9.4 *Trellis*

1.10 Interoperability

1.10.1 *Message Bus*

1.10.2 *CORBA*

1.10.3 *Polylith*