# Stateful Multicast Services for Supporting Collaborative Applications

Hyong Sop Shim, Robert W. Hall, Radu Litiu and Atul Prakash
Department of Electrical Engineering and Computer Science,
University of Michigan, Ann Arbor, MI 48109-2122 USA
E-mail: hyongsop,rhall,radu,aprakash@eecs.umich.edu

**Abstract**

Collaborative, multi-user applications require group multicast services with ordering guarantees for maintaining consistency of replicated shared state among collaborating processes. In traditional group multicast services (such as Isis), the group's shared state is maintained only by the clients and the multicast service facilitates atomic state transfer from existing clients to new members. In this paper, we argue that, in order to support collaborative applications in Internet-type environments, a copy of the group's state should also be maintained by the multicast service. We show that by maintaining a copy of group's state, the multicast service can provide consistently fast group-join and state transfer time when both slow and fast clients are present or when clients are unreliable — a crucial requirement in collaborative, multi-user applications where users may dynamically join and leave a collaborative session and expect predictable join times and interactive response time even in the presence of slow or unreliable clients. We show that the overheads incurred by a multicast service in managing each group's shared state can be made minimal and that the multicast service does not have to be aware of the semantics of the group's state. We present the design of such a multicast service, present performance results, and discuss how it meets the various needs of collaborative applications.

## 1 Introduction

As with traditional systems for distributed computing such as distributed operating and database systems, computer-supported collaborative systems enable multiple users to share data, both synchronously and asynchronously. However, traditional distributed systems strive to provide the illusion of working alone, and thus the data sharing is provided as transparently as possible [7]. The goal of a collaborative system is to empower geographically dispersed users to effectively *work together* over distance. Thus the sharing of data is made apparent in collaborative systems, and the mechanics of data sharing often dictates the overall effectiveness of collaboration.

Therefore, the management of shared data or *shared state* in collaborative systems places unique requirements that are not met by existing group multicast services. For example, the application responsiveness takes on much more importance in a collaborative system designed to provide a highly interactive collaboration environment. If a system is not responsive at times (perhaps due to other clients being slow or disconnecting), users are likely to get frustrated with the system. The drastically increased usage of the Internet and World Wide Web over the last few years have exasperated the problem of shared state management in collaborative systems. While the Internet and WWW have drastically increased the visibility and usage of collaborative systems in general, a system designed to support collaboration in such network environments cannot assume anything about the availability and reliability of network and computing resources such as bandwidth and processing power.

Another dimension to collaborative applications is that users often require awareness about other group members and their status in a session. Multicast services need to provide support for maintaining awareness information and notifying clients when the information changes. Further, some clients may only be interested in awareness information whereas others are only interested in receiving state updates. Multicast service needs to be flexible in supporting needs of different types of clients efficiently.

Replicated state consistency/correctness requirements may also differ in different collaborative applications. For instance, some collaborative applications need to rely on the use of locks to ensure that their updates only apply to the state in which the updates were generated, i.e., no intervening updates from other users have taken place. Other collaborative applications only require consistency of replicas, with no requirement of preventing intervening updates.

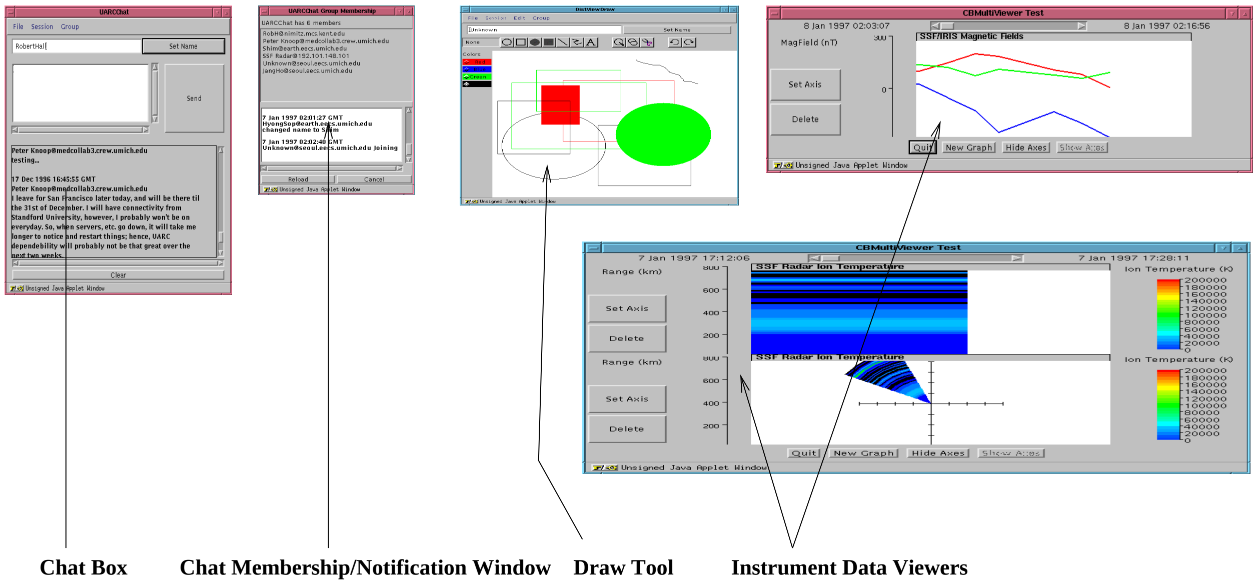
This paper presents our approach to providing a multicast service that meets the above requirements. The services are provided by our Corona stateful, multicast server and are designed to support both synchronous and asynchronous collaboration over the World Wide Web, where collaborating clients may be dynamically downloaded over the Internet.

The rest of the paper is organized as follows. Section 2 motivates our work in providing support for computer-supported collaboration. Section 3 discusses the key requirements of shared state management. Section 4 discusses related work. Section 5 provides an overview of the Corona multicast service. Section 6 details the suite of shared state management and multicast services that Corona provides. Section 7 discusses the current implementation status and reports on preliminary performance results. Section 4 discusses related work, and Section 8 concludes the paper with a brief summary of Corona and our future plan.

## 2 Motivation

Our work on the management of shared state in computer-supported synchronous collaboration has its origin in a project focusing on developing an experimental testbed for wide-area scientific collaboratory work. This testbed is implemented as a large object-oriented distributed system on the Internet and provides a collaboratory environment in which a geographically dispersed community of scientists perform real-time experiments at remote facilities without having to leave their home institutions. This community of scientists has extensively used our system over the last few years and has expressed a high degree of satisfaction with its mechanisms for remote collaboration.

Client applications supported by the Corona services provide the users with various facilities for remotely conducting their science. These include various shared data viewers for graphically displaying instrument data, a multi-party chat box for exchanging textual messages, and a notebook-like draw tool for saving and sharing notes, images and drawings. Figure 1 shows the graphical interfaces of some of the Corona-based collaboration tools.



Peter Knoop@medcollab3.crew.umich.edu
testing...

17 Dec 1996 16:45:55 GMT
Peter Knoop@medcollab3.crew.umich.edu
I leave for San Francisco later today, and will be there til the 31st of December. I will have connectivity from Standford University, however, I probably won't be on everyday. So, when servers, etc. go down, it will take me longer to notice and restart things; hence, UARC dependebility will probably not be that great over the next two weeks.

RobH@nimitz.mcs.kent.edu
Peter Knoop@medcollab3.crew.umich.edu
Shim@earth.eecs.umich.edu
SSF Radar@192.101.148.101
Unknown@seoul.eecs.umich.edu
JangHo@seoul.eecs.umich.edu

7 Jan 1997 02:01:27 GMT
HyongSop@earth.eecs.umich.edu changed name to Shim

7 Jan 1997 02:02:40 GMT
Unknown@seoul.eecs.umich.edu Joining

Figure 1: Collaboration tools using the Corona communication services

Each tool has a different set of shared state management requirements. Consider the multi-party chat box and a shared data viewer. The chat-box allows scientists to exchange textual messages. The shared state of the chat box is defined to be the history of messages scientists have exchanged. The history is updated in real time as more messages are exchanged, and when a scientist joins an on-going chat session, the scientist is given the history of exchanged messages. The latter feature is provided on demand. A shared data viewer provides multiple scientists with a synchronized view of live instrument data, including the range of data being displayed and the physical appearance of the viewer; when a scientist resizes his or her copy of the viewer, the viewers of the scientist's colleagues also change to the same size.

One distinction in the shared state management of the described tools is in how updates are applied. When a new message arrives, the chat box simply appends the new message to its history of exchanged messages. In contrast, when the size of the shared viewer is changed, all the replicas of the viewer have to resize. Hence, an update in the chat box, i.e. a new message, does not convey the complete information about the current shared state and thus does not override the existing value of the shared state; the update contains an *incremental* change. However, an update in the viewer, i.e. a new size of the viewer, overrides the existing value. The rationale is that it is not beneficial to the collaboration of scientists to know the past history of the viewer size. Hence, the resize update contains a *complete* change to part of the shared state of the viewer.

Another distinction concerns the degree of synchronization. The users are not very concerned about the total order of messages shown in the display area of the chat box, especially when the messages are timestamped. Instead, they want to interact with each other without delay by being able to exchange messages as freely as possible. In the case of the shared viewer, however, concurrent updates on the viewer should be serialized. Otherwise, the shared state of the viewer seen by different scientists may be inconsistent. For example, the viewer may be of different sizes, or the instrument data displayed in the window may be in different display modes.

Over the years, this system has evolved through several generations of prototypes. The current design is an applet-based architecture implemented in Java. It takes advantage of the accessibility and ubiquity of the World Wide Web and Java platform-independence. A key component of the system are the Corona communication services, which embody our approach to shared state management; these services are powerful and flexible enough to meet the varying shared state management needs of collaborative applets as well as general collaboration environments.

## 3 Design Requirements

In CSCW, the management of a group of processes sharing data places unique demands on the underlying support system. Because the runtime behavior of the processes have a direct impact on the effectiveness of collaborative work of end users of the system, a collaborative system cannot blindly apply solutions developed for general distributed computing problems. Instead, it should adequately address requirements unique to CSCW. Our experience with CSCW systems shows that users often want to interact with collaborative applications as freely as they do with single-users applications and that they tend to be impatient with any delays caused by network congestion, faulty processes etc. A developer should take into account such considerations when designing a collaborative system. Critical issues concerning a collaborative system design include:

- **Support for Group Communication Services:** A collaborative system should provide support for group communication [4]. Processes should be able to create, join, and leave groups. A process in a group should be able to communicate with the other members of the group without the knowledge of the full membership of the group. Furthermore, the actions taken by processes in a group should be synchronized so that the processes have a consistent view of the shared data of the group.

- **Fast Join:** A process should join a group of collaborating processes as fast as possible. If the group has shared data, the current state of the shared data should also be transferred to the new process with a predictable response time even in the presence of network failures and faulty processes. Further, a process should be able to join and leave a group unobtrusively; the existing processes in the group should be able to carry on with their operations in the presence of multiple, concurrent joins and leaves.

- **Fast Response Time:** A process should be responsive to user input. The fact that the process may be in collaboration with other processes should not prevent the user from freely interacting with the process.

- **Client-based Semantics:** The interpretation of the semantics of shared data should be the responsibility of collaborating processes. A general purpose group communication service is provided, thus allowing the exchange of data between all types of communication processes.
- **Persistence:** A group and its shared data should be able to outlive the process members of the group. A process joining a group after the group has assumed the null membership is transferred the persistent state of the group's shared data.
- **Robustness:** A group should be robust. Faulty or slow processes should not block the operations of the non-faulty processes in a group. Users joining a group should not be impeded in receiving shared state by the slowness or failure of other group members.
- **Group Membership Support:** Generally, in distributed systems knowledge of membership of a group is important to maintain consistency of replicated state and to ensure strict guarantees of message delivery and ordering. In a collaborative system, group membership takes on an important social aspect of *awareness*– users collaborating over shared state want to be aware of each other and their activities. The system should make this information available to client applications, via queries to the server or notifications of membership changes.

## 4 Related Work

As transport layer subsystems, ISIS [5, 4], and Transis [2, 3] support the notion of process groups, notification of membership changes, and group multicast and may be could be used to build services such as our build our group awareness and group notification services. Both support a fully replicated architecture with individual members maintaining replicated state. In ISIS, the join of a new member involves the execution of a join protocol among all group members, to ensure installation of a consistent view of membership. Slow members can slow the join of a new member.

In comparison, Corona supports fast joins by accomplishing state transfer on join via the server with no interaction of other, potentially slower group members. By not imposing costly ordering requirements that ISIS's causal and atomic broadcast protocols involve, by Corona allows fast joins even in a wide-area network environment, as membership is maintained by the Corona service, not the clients.

Furthermore, any state associated with a group must be transferred from an existing client to the joining client. If the client transferring the state fails, another client must be selected to transfer the state and the group membership view reformation protocol run. The time to complete this join thus reflects the timeout for failure detection and the additional request.

Transis, like ISIS, provides a transport layer with a variety of multicast ordering and delivery semantics. Transis has been used to support distributed replicated database systems. By having replicated Transis processes, a higher level of fault-tolerance that a single single server can achieve has been realized. One Transis-based approach [1] to achieve consistent replication suffers from the inefficiency of using global total ordering with Lamport clocks. Corel [8] addresses this and also addresses fault-tolerance. However, these approaches require consistent membership views and require end-to-end acknowledgments for each message.

In its goals, Lotus' NSTP [10] closely resembles our Corona server with regard to shared state management. Both advocate centralized management of shared state and provide similar administrative services in its support. The semantics of shared state is client-based in both systems so that their services are generalized to a wide range of applications. Furthermore, the notion of *Place* in NSTP is synonymous with the group concept in Corona, and *Things* in a Place correspond to the objects in a shared state of a stateful group in Corona.

However, NSTP and Corona differ in several aspects of their shared state management support. First, NSTP does not support the notion of incremental updates. Each update on a Thing always overwrites the old value of the Thing. This limitation would make the development of tools such as our chat box or draw applet difficult as the updates on the shared states of these tools are fully incremental. One way to simulate an incremental update is to dynamically create and add a new Thing in a Place for each update. However, this would entail a difficult problem of uniquely naming new Things at runtime. Second, NSTP does not transfer shared states to clients; instead, when a client enters a Place, it is only given the names of the Things in the Place. Hence, clients always access shared objects remotely. This may significantly degrade the performance in terms of user responsiveness in a highly interactive collaboration environment, especially over a wide area network. Finally, NSTP does not support any notion of persistence in its Things or Places. On the other hand, the Corona server does not support the *Facade*-like capabilities for viewing the shared states of groups before actually joining the groups.

Many other systems also provide administrative services similar in part to the services provided by the Corona server. Both IRC [9] and Zephyr [6] provide centralized messaging and notification services, which are similar to our group awareness and multicast services. However, neither of these systems supports the concept of shared state or role distinction among members and are not intended to support general synchronous collaborative activities.

## 5 Overview of Corona

In this section, we discuss the architectural features of the Corona system. We first define a few fundamental concepts behind Corona to better discuss our approaches to supporting computer-supported collaboration.

### 5.1 Shared State Model

In CSCW, collaboration is achieved by a group of processes sharing data. The shared data, or *shared state* in Corona, is defined as set $S$:

$$S = \{(O_1, S_1), (O_2, S_2), ..., (O_n, S_n)\}$$

where $i$ is an identifier of a shared object $O_i$, and $S_i$ is a *byte stream* encoding of $O_i$. The identifier of a shared object is used to uniquely identify the object in $S$ and may be automatically generated by a support system for building collaborative applications such as the DistView toolkit [11].

Note that the state of a shared object is type-independent. This is consistent with the requirement of the client-based semantics in Section 3. Corona requires that a share object should be able to write its internal state to a stream as well as set its state with data encoded in a stream upon request. This requirement is commonplace in object-oriented applications that utilize persistent objects.

### 5.2 Communication Groups in Corona

A *group* forms the basic unit of communication in Corona. A group is defined to be a set of processes, termed *members*. A group has a shared state as defined in Section 5.1, and the members of the group operate on the shared state by accessing and modifying the shared objects in the shared state. Corona requires that a process has to be a member of a group in order to operate on the shared state of the group. The group members communicate with each other by exchanging messages among themselves.

A group may be characterized as either *persistent* or *temporal*. A persistent group and its shared state exist even when it has no members. A temporal group ceases to exist when it has no members, and its shared state is lost.

Note that our notion of group is distinct from the concept of a group of users who may be engaged in various collaboration activities. It may be viewed that our group represents a particular collaboration activity that a set of users are presently participating to.

### 5.3 Corona Architecture

The major component of Corona is a *stateful* server that provides group multicast services. The server is stateful because it maintains copies of shared states of groups as defined in Section 5.2. Figure 2 illustrates the Corona architecture. Corona manages groups and their shared states. When a process joins a group, the server transfers a copy of the current shared state of the group.

It may be viewed that the Corona server maintains a cache of the shared state of a group, and group members keep the server's copy consistent. From the server's point of view, the shared state of a group is a set of byte streams identified by object identifiers, and thus, the server cannot perform application-specific operations on

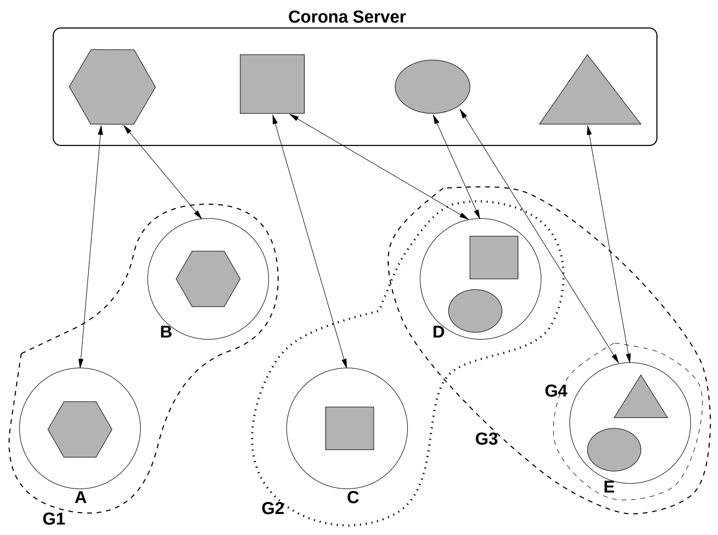


Figure 2: Architectural Overview of Corona. Circles represent clients, dotted lines depict groups, and different shapes represent different shared states. Note that clients may belong to different groups; Client D belongs to both Group G2 and Group G3, and Client E belong to Group G3 and Group G4. Group G4 presently has Client E as its only member.

its copy of the shared state. Instead, the group members update the server's copy through the server's group multicast services.

## 6 Corona Services

The Corona server provides a suite of group multicast and related services. In describing the services, we show how Corona meets the design requirements in Section 3. The services may be categorized as: *group membership*, *group multicast*, *synchronization*, and *checkpointing*. The group membership service provides interfaces for creating, joining, leaving groups etc. The group multicast service provides interfaces for broadcasting updates on shared state. The synchronization service provides interfaces for synchronizing client updates through locks, and the checkpointing service allows a client to take a snapshot of a shared state. A group of clients may subscribe to any combination of services and specify how a particular service is provided depending on collaboration semantics. Figure 3 shows a usage example of the Corona services.

We assume that messages between the Corona server and clients are reliably and FIFO delivered. Table 1 provides the main interfaces for the Corona multicast services. Note that the interfaces for the group multicast and checkpointing services use *sequence numbers* obtained from the server via *acquireSeqNum()*. The usage of sequence numbers is explained when discussing appropriate services.

### 6.1 Group Membership Service

An authorized client may create and delete a group with *createGroup()* and *deleteGroup()* messages respectively. The Corona server works in conjunction with an external workspace session manager that determines which client has a privilege to take these actions. When creating a group, a client specifies the initial state of the group as defined in Section 5.1. By default, a group persists in the server even if its membership has become null. The server deletes a group only in response to the *deleteGroup()* message, and the shared state of a deleted group is lost.

A process joins a group by sending a *joinGroup()* message to the server. A process may query the server for currently existing groups by sending *getGroupNames()* message. Because the server always has the latest copy of the shared state of a group, as discussed in Section 6.2, the server immediately transfers the current shared

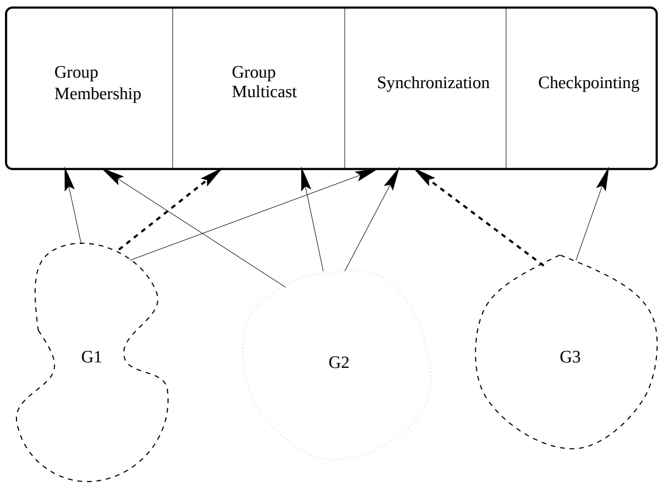


Figure 3: Corona Services. The arrows represent subscriptions. Group G1 is subscribing to group membership, group multicast, and synchronization services. The dashed arrows show that Group G3 is using the synchronization service differently from Group G1 and that Group G2 is using the group multicast service differently from Group G1.

| | | |
|---|---|---|
| Group Membership | *createGroup(gName,* $S = \{(O_1, S_1), (O_2, S_2), ..., (O_n, S_n)\}$*)* | creates a group of name *gName* and initializes the state of *gName* to set $S$ |
| | *joinGroup(gName, aRole)* | joins a group of name *gName* with role, *aRole* |
| | *joinAck()* | acknowledges the join completion of a member with |
| | *leaveGroup(gName)* | leaves a group of name *gName* |
| | *deleteGroup(gName)* | deletes a group of name *gName* |
| | *changeRole(oldRole, newRole)* | changes the role of a member to *newRole* |
| | *getGroupNames()* | returns the names of groups currently present at the server |
| | *getMembership(gName)* | returns the membership of group *gName* |
| | *enableMembershipNotif(aGroup)* | enables membership change notification for a member |
| | *disableMembershipNotif(aGroup)* | disable membership change notification for a member |
| Group Multicast | *bcastStateExcludeSender(gName,* $O_i$*, stateMsg)* | multicasts new state message for object $O_i$ |
| | *bcastStateIncludeSender(gName,* $O_i$*, stateMsg)* | multicasts new state message for object $O_i$ |
| | *bcastUpdateExcludeSender(gName,* $O_i$*, stateMsg)* | multicasts an incremental change to object $O_i$, *state*, for group *gName* |
| | *bcastUpdateIncludeSender(gName,* $O_i$*, stateMsg)* | multicasts an incremental change to object $O_i$, *state*, for group *gName* |
| Synchronization | *acquireLock(gName,* $S = \{i, j, ...\}$*)* | acquires a lock on an object whose id is in $S$ |
| | *releaseLock(gName,*$S = \{i, j, ...\}$*)* | releases a lock on an object whose id is in $S$ |
| Checkpointing | *checkPoint(gName, oid, objStateMsg, aSeqNum)* | resets the state of $O_{oid}$ to *objStateMsg* |

Table 1: Corona client interface for creating, joining groups and multicasting messages

state of the group to the sender of the *joinGroup()* message along with a *member identifier*. The sender client is considered a *pre_member* of the group and becomes a full member of the group by sending the *joinAck()* message. The distinction between a pre_member and full member is that a pre_member cannot multicast updates on the shared state. The server waits for the *joinAck()* message from a pre_member for a timeout period. When the server timeouts, it removes the pre_member from the group. Figure 4 graphically illustrates the join protocol.

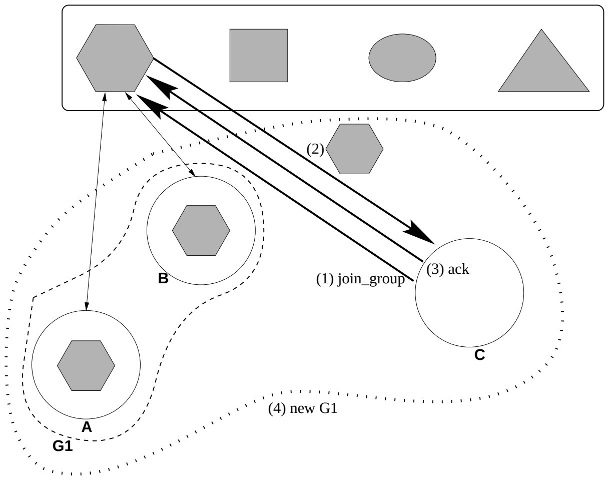


Figure 4: Fast State Transfer in Corona. A new client is joining Group G1. The numbers represent the sequence of actions. C sends a join request to the server, the server transfers the current shared state of G1 to C, C acknowledges the receipt of the state, and then C becomes a full member of G1.

The details of the join and state transfer protocol are described in Section 6.2. Note that the join protocol does not involve the existing members of a group. The existing members are unaware of the fact that a new client is joining, and they continue to work concurrently with the join protocol. If the server receives an update multicast message from an existing member, it multicasts the message to the existing members as well as any pre_members as discussed in Section 6.2.

The Corona server allows clients to specify *roles* when joining a group. The server supports the following member roles: *principal*, *observer*, and *membership-observer*. Principals have update privilege on a shared state. Observers and membership-observers are casual members who may only view the updates on the shared state or the changes on the group membership, respectively. A member may dynamically change its role after join by sending the *changeRole()* message. The server may provide different quality of service (QoS) for members of different roles. For example, the server retries sending messages to principal members after a timeout but removes observer members from a group after the first timeout.

When the membership of a group changes, the server multicasts membership change notifications to group members. A group member may disable membership change notification by sending the *disableMembershipNotif()* message. A member may also query for the membership information at any time by sending the *getMembership()* message. The membership service also maintains a *properties list* for each member–attributes such as the name of an end user of a member process, the member process's host name, etc.

## 6.2 Group Multicast Service

A group member multicasts updates on a shared state by sending *bcast()* messages to the server. An update multicast includes the following information: a member identifier, the identifier of a shared object, and a byte stream encoding of the update to be applied to the object. The server provides two forms of group multicast: *bcastState()* and *bcastUpdate()*. The update information in a *bcastState()* contains a new state of the object specified in the multicast, and the new state *overrides* the present state of the object. In contrast, the update information in a *bcastUpdate()* contains an *incremental* change to the state of the object, and the change is appended to the existing state. Hence, the *bcastUpdate()* is used to preserve the history of updates on a shared object. We use the phrase multicast instead of broadcast to highlight the fact that messages may be sent to a subset of the total membership of a group based of on the roles of the individual group members.

A multicast message is assigned a *sequence number* by the server when it is received at the server. The sequence number monotonically increases for each message received and thus uniquely imposes a total order on multicast messages within a group. For each shared object in the shared state, the server maintains a list of multicast messages sorted in the decreasing order of the sequence numbers of the messages. A shared object may receive both types of update multicast, and clients decides how to multicast an update based on a particular collaboration semantics.

When a multicast message, $M$, arrives, the server first makes a copy of $M$, $M'$, and assigns a sequence number to both $M$ and $M'$. The server then prepends $M'$ to the multicast message list of the object to which $M$ applies and then multicasts $M$ to the members of a group to which the sender of $M$ belongs. Note that a member that receives $M$ would know the sequence number assigned to $M$.

When a client joins a group, the server transfers the current state of the group to the new client. For each object in the shared state, the server performs the following operations to compute the current state of the object: if the head of the multicast message list of the object, $H$, is a *bcastState()*, then $H$ contains the current state of the object. If $H$ is a *bcastUpdate()*, then the server goes down the list, marking each *bcastUpdate()*, until it finds a *bcastState()*. The server then creates a list of marked messages in the ascending order of the sequence numbers of the messages. The resulting list represents the current state of the object.

A multicast message, $M$, can be sent to the server either *sender-inclusively* or *sender-exclusively*. If $M$ is sent sender-inclusively, the server multicasts $M$ to all the members of a group to which the sender of $M$ belongs, including the sender itself. If $M$ is sent sender-exclusively, the server does not multicast $M$ to the sender. A client sender-inclusively multicasts a message when the client needs certain operations that the server performs on multicast messages. An example of global operation that the server performs is timestamping multicast messages with real time, which may be useful in a certain collaboration environment, e.g. a collaborative chat session in which messages exchanged among users have to be sorted in a total order.

## 6.3 Synchronization

Clients may synchronize updates on a shared state by subscribing to the synchronization service of the server. The server implicitly maintains a lock table for a shared state, keyed by the object identifiers of shared objects. Given an identifier of a shared object, the table indicates whether or not the object is locked. Upon receiving an *acquireLock(gName,* $S = \{i, j, ..., k\}$*)* message, where $x \in S$ is an object identifier, the server tries to acquire a lock on an object whose id is specified in $S$. If an object is already locked, the server notifies the sender of the message of the acquisition failure. If successful, the server updates appropriate table entries and notifies the sender of the successful acquisition. A message, *releaseLock(gName,* $S = \{i, j, ..., k\}$*)*, releases the locks on the objects specified in $S$, and the server updates appropriate lock table entries. If the server detects that a member has locked a set of objects for over a timeout period, the server treats the member as faulty and update appropriate table entries to reflect the objects are no longer locked.

## 6.4 Checkpointing

A client may reset the state of a shared object with the checkpointing service. A client sends the *checkPoint(aGroup, oid, objStateMsg, aSeqNum)* message to the server, where *aGroup* is the group to which the client belongs, *oid* is the identifier of a shared object, *objStateMsg* is a message containing the new state of the object, and *aSeqNum* is a sequence number of a multicast message the client has received from the server. *aSeqNum* effectively indicates to the server the point in the history of the state of the object at which the checkpointing should be performed. Upon receiving the message, the server goes down the multicast message list for $O_{obj}$, examining the sequence number of each message encountered. Upon finding the message whose sequence number matches *aSeqNum*, the server deletes the message found as well as any other messages with a lower sequence number from the list and appends *objStateMsg* to the list.

## 7 Implementation Status and Performance

The Corona architecture has been be implemented in a prototype server supporting the DistView user-interface sharing toolkit and applications built to use Corona's services. In this section we discuss the implementation, how applications use Corona's services, and some preliminary performance results.

### 7.1 Implementation

A prototype of the Corona server has been implemented as a multi-threaded Java application, supporting downloadable Java applet clients. The Corona server supports the multicast, membership, synchronization, state transfer, and checkpointing services. The interface in Table 1 has been implemented as a Java class library that applets such as the chat tool, draw, and image viewers utilize.

Corona has been successfully tested and used in various scientific campaigns and project meetings, supporting numerous collaborative scientific experiments and on-line group discussions. In one recent campaign, approximately 40-50 participants utilized our tools to conduct science on atmospheric phenomena over a three day period. The scientists were dispersed throughout North America and Europe, operating on a variety of platforms, including Windows 95, Solaris, and HP-UX with connectivity ranging from high-speed links to modems.

The collaboration tools shown in Figure 1 are implemented as Java applets and include a chat box, a draw tool, and a set of instrument data viewers. The chat box provides an edit area for composing messages and a scrollable area for displaying a list of received messages. Similar both to a shared notebook and whiteboard in its functionality, the draw tool provides a canvas for drawing, taking notes, and importing images. The data viewer provides configurable windows for displaying different kinds of instrument data. The windows of the viewer may be exported and imported to support synchronous collaboration. The shared states of the chat box, the draw tool, and a shared window of the data viewer are respectively defined as the textual messages exchanged among collaborators, the contents of the canvas, and the graphical attributes of a shared window as well as the application-defined objects needed to display instrument data in the window. The awareness of other users is provided via the membership status window that displays the group membership service's notifications of group membership changes. Any client tool may have this window open to monitor the membership changes of its group.

### 7.2 Utilization of Corona Services

The varying collaboration semantics that the above tools are designed to support result in different usages of the Corona services. For example, when users belatedly join an ongoing chat-box discussion, they want to see as many previously exchanged messages as possible so that they can catch up with the current discussion. Thus, when the chat box joins a group, it receives all the previously exchanged messages in the group through the state transfer service of the server. When a user sends a message, new messages do not overwrite previous messages– they constitute memoryful updates on the shared state of a chat box group. The messages are also multicast sender-inclusively as they require timestamping by the server. Updates to the shared state require no explicit locking in order to encourage users to freely exchange messages.

Unlike the chat box, the draw tool and a shared window of the data viewer require tighter synchronization of their respective shared states. In both cases, users should have the same view of the shared states of these tools in order to avoid conflicting updates, which may degrade the effectiveness of collaboration. Hence, both tools employ locks in updating their respective shared states. Furthermore, the updates, e.g. drawing of a new shape or resizing of the shared window, are sender-exclusively group multicast because the updates have been locally processed before being multicast and do not require server timestamping.

### 7.3 Preliminary Performance Measurements

A goal of our architecture is that the management of shared state by the Corona multicast service should not significantly increase the cost (in terms of message latency) of message multicast or impact scalability of groups compared to groups without server-maintained shared state, where the server acts as a sequencer only.

We compare the performance of group broadcast when the server maintains shared state and when the server does not maintain shared state in terms of client throughput, round-trip delivery time to clients, and scalability in terms of number of clients and message size. Additionally, we evaluate fast joins by comparing the cost of a join in the presence of slow clients for server-managed state contrasted with client-managed state.

Note that Corona's group multicast service currently multicasts via multiple point-to-point messages from the server to clients. [1].

### 7.3.1 Overhead of Shared State Maintenance

In Figure 5, we compare the round-trip delay for a multicast for a server maintaining state with a server not maintaining state, with the message size fixed at 1000 bytes (upper curves) and at 10000 bytes (lower curves). For these tests, the machines used were a mix of Sun Sparc 20s and Ultra Sparc 140s on a LAN. The server runs as a stand-alone Java application.

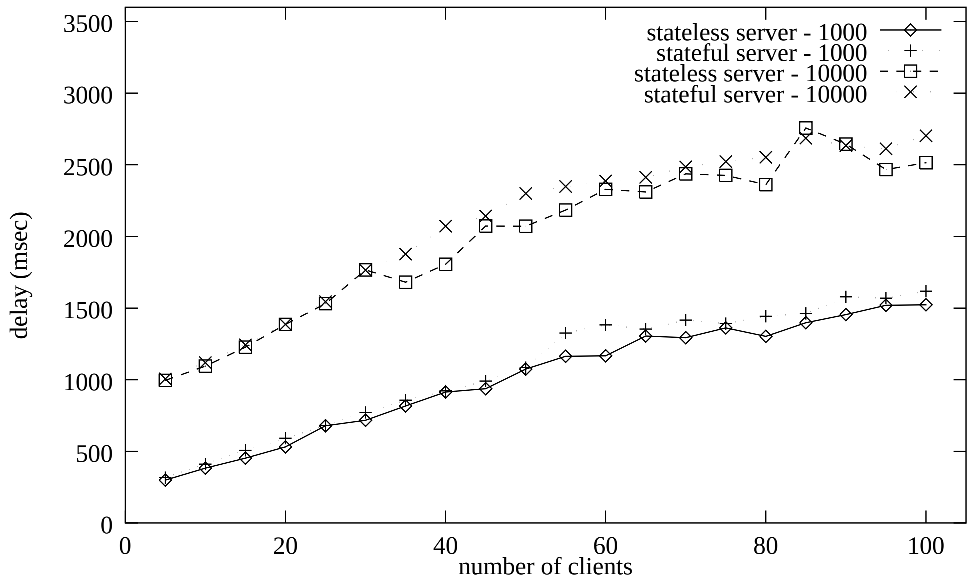


Figure 5: Group Multicast: Round trip delay vs. #clients for a message of size 1000 bytes

In all the experiments all clients but one are just receivers; the receivers only connect to the server, join a group and receive the broadcast messages addressed to that group. The extra client is both a sender and a receiver; we use that to measure the round-trip delay. In order to measure the maximum delay, this client is always the last one that joins its group. This accounts for the number of clients, as we have observed that in multicasts through our server the threads for each client are scheduled (by the Java runtime on Solaris) in the order of membership. Thus, the thread for the last client to join would be scheduled last, and would be sent the message last. The clients are at any moment uniformly distributed over all the machines used.

In these and other experiments, we observed that for messages of size up to a few hundreds of bytes the size makes little difference in round-trip times. The influence of the message size is more evident above 1000 bytes. The same observation applies for the time used by the client to send messages. From observation of real messages exchanged by our client applications, the typical size of a message generated by the chat or draw tools is around 1000 bytes.

In all the cases the round trip delay increases almost linearly with the number of clients, with the message size fixed at 1000 bytes. As illustrated in Figure 5, the overhead of maintaining the state at the server is for the most part minimal. Since the size of the message is fixed in this example, the overhead of storing the state in the server's internal data structures is constant regardless of the number of clients. However, as the graph illustrates,

[1]in contrast to using IP multicast

as the number of clients rises (above 50 in the graph), other factors come into play to impact performance. As the number of clients increases, the number of threads the server maintains increases. The scheduling of these threads for broadcasts varies and garbage collection threads can add additional delay.

Additionally, when the message size increases from 1000 bytes to 10000 bytes, the delay remains linear with the number of clients. The growth ratio for the curves corresponding to the delays in each case (1000, 10000 bytes) remains roughly the same, so the size seems to have little impact on the growth of the delay with the increase in the number of clients.

### 7.3.2 Fast Joins

As previously discussed, a requirement for collaboration is fast group joins. By maintaining the state at the server as Corona does, the goal was to ensure that fast clients (clients with high-bandwidth connections) are not affected by other, slower, bandwidth-limited group members.

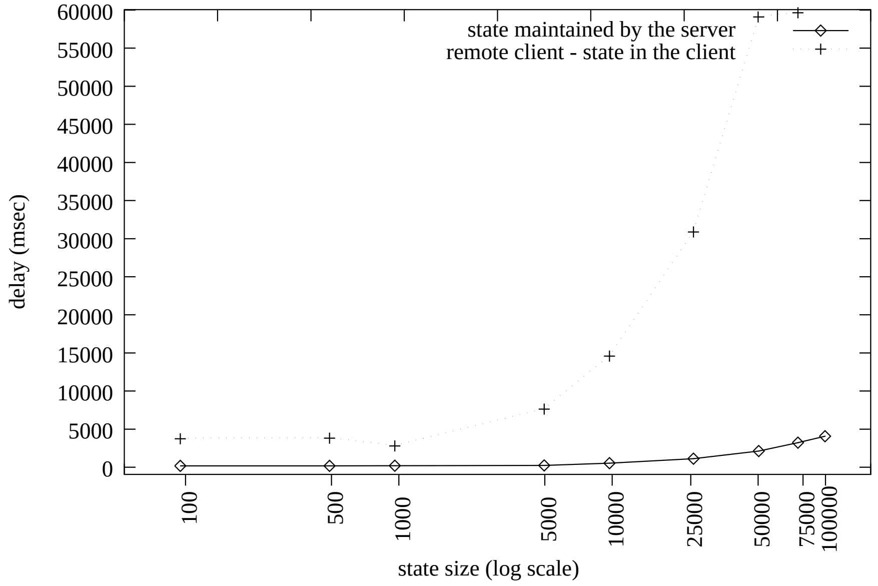


Figure 6: State Transfer: Impact of slow, remote client with stateless, stateful server

In Figure 6, we present the results of the following experiment.

A Corona server runs on a Sun Ultra Sparc 140 with one client attached. The cost of updating data structures for the server copy of the state increases as the size of the state updates increases. This client is running remotely also on a Sun Ultra Sparc, but over a 56 Kbps link in Sondrestromfjord, Greenland. A new client (located on the same LAN as the server, running also on an Ultra), joins the group. In this diagram, we varied the size of the shared state from 100 bytes up to 100000 bytes.

If the server maintains the shared state, the server immediately transfers this state to the joining client, regardless of other clients. The lower curve in Figure 6 shows that the size of the state (shown in $log_n$ scale, has little impact on the transfer time to the fast client, remaining between 1-5 seconds.

If the server doesn't maintain state, but instead relies on transferring state from another client for the join, as other systems use, the server will have to request state transfer from a client, that could be a slow one, then forward the state to the joining client. As the upper curve illustrates, transferring state from the bandwidth-limited client in Greenland for the joining client dramatically slows down the join as the size of the state increases. Thus, slow clients in the system can adversely affect joins of faster clients.

A related problem encountered when the server doesn't maintain state is the side-effects of client failures. If the client that the server requests a state transfer from in order to fulfill a join request fails, the server has to detect the failure and contact another client (if it has more clients). The overhead to the state transfer time in this scenario is approximately $STtime = rtt + N * timeout$, where $rtt$ is the round-trip time for the actual transfer, $N$ is the number of (possibly) failed clients contacted, and *timeout* is time it takes the server to detect a possible client failure (say 1 min). Thus maintaining state at the server enables the server to continue to supply state to new clients in the presence of network partitions and client failures.

## 8 Conclusions and Future Work

In this paper, we presented a group multicast service that also manages replicas of shared state when providing support for computer-supported collaboration in Internet-like environments. Traditional group multicast services solely rely on client processes to manage the consistency of shared state that is fully replicated at clients. In such services, accommodating a new process to a group may affect the operations of existing clients for an unpredictable amount of time, especially over a wide area network, as they run a complex state-transfer and membership view agreement protocol. In a highly interactive collaboration environment, such a performance degradation may significantly reduce the effectiveness of collaborative work of end users.

We showed in the paper that a group multicast service that manages copies of shared states offers a number of advantages over the traditional approach, including fast group-join and predictable state transfer times in the presence of clients of varying bandwidth and processor power. The semantics of shared state can be made transparent to such a service, and the overhead of managing shared state is minimal.

Our current research efforts are focusing on increasing the scalability and robustness of the server and examining the issues involved in a distributed implementation of the server, which include:

- Topology/organization of distributed servers: hierarchical organizations, ring organizations;
- Management of shared state consistency between the servers;
- Exploitation accommodation of user roles to achieve scalability
- Increased persistence and fault-tolerance capabilities.

## 9 Acknowledgments

This work is supported in part by the National Science Foundation under cooperative agreement IRI-9216848, and by IBM.

## References

[1] O. Amir, Y. Amir, and D. Dolev. A Highly Available Application in the Transis Environment. In *Proc. of the Workshop on Hardware and Software Architectures for Fault Tolerance, Lecture Notes in Computer Science 774), month=June, year=1993,*.

[2] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. Technical Report TR CS91-13, Computer Science Dept., Hebrew University, April 1992.

[3] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Robust and Efficient Replication using Group Communication. Technical Report TR CS94-20, Institute of Computer Science, The Hebrew University of Jerusalem, Nov. 1994.

[4] K. P. Birman. The Process Group Approach to Reliable Distributed Computing. *Comm. of the ACM*, 36(12):37–53, Dec. 1993.

[5] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. on Computer Systems*, 9(3):272–314, Aug. 1991.

[6] C. A. DellaFera and M. W. Eichin. The Zephyr Notification Service. In *Proc. of the USENIX WInter Conference*, Dallas, Tx, 1988. USENIX Association.

[7] C.A. Ellis, S.J. Gibbs, and G.L. Rein. Groupware: Some Issues and Experiences. *Communications of the ACM*, pages 38–51, January 1991.

[8] I. Keidar. A Highly Available Paradigm for Consistent Object Replication. In *Master's Thesis, Institute for Computer Science, The Hebrew University of Jerusalem*, 1994.

[9] J. Oikarinen and D. Reed. Internet Relay Chat Protocol. *RFC 1459*, 1993. Available at ftp://ds.intenic.net/rfc/rfc/1459.txt.

[10] J. F. Patterson, M. Day, and J. Kucan. Notification Servers for Synchronous Groupware. In *Proc. of the Sixth ACM Conference on Computer-Supported Cooperative Work*. ACM Press, Nov. 1996.

[11] A. Prakash and H. Shim. DistView: Support for Building Efficient Collaborative Applications using Replicated Objects. In *Proc. of the Fifth ACM Conf. on Computer Supported Cooperative Work*, pages 153–164, Chapel-Hill, NC, Oct. 1994.