Group Communication Support for Distributed Collaboration Systems*

Injong Rhee† Shun Yan Cheung‡ Phillip W. Hutto‡
Alan T. Krantz‡ Vaidy S. Sunderam‡

Abstract

The Collaborative Computing Transport Layer (CCTL) is a communication substrate consisting of a suite of group communication protocols. The design of CCTL supports the needs of distributed collaborative applications. CCTL is based on a two-level group hierarchy that naturally matches the structure of many collaborative applications and that allows several implementation optimizations. Logical interconnections among processes, called *channels*, define an efficient, light-weight group mechanism, providing a variety of communication services such as reliability and message ordering. Related channels are associated with a heavy-weight group, called a *session*, that provides group management services, such as membership, for its associated channels. Sessions and channels run different protocol stacks, allowing a flexible and useful separation of group managements semantics and communication service quality. This also allows the efficient reuse of existing group management services when introducing new communication services.

1 Introduction

High-speed, wide-area networks such as the Internet encourage the transfer of large volumes of data between potentially distant hosts. Novel collaborative technologies now allow geographically dispersed groups of co-workers to interactively conduct a range of work-related activities. Collaboration-aware applications are designed and implemented with features to support collaboration such as concurrency and floor control (who gets to do what, when). Stand-alone programs can be jointly manipulated using special application-sharing programs that multiplex the input and output streams across several workstations [1]. Computer-supported cooperative work (CSCW) systems combine these technologies to offer an integrated computing environment for conferencing and collaborative work.

*Supported in part by NSF grant ASC-9527186.

†Department of Computer Science, North Carolina State University, Raleigh, NC 27695-7534, USA. E-mail: rhee@eos.ncsu.edu, This author is also supported in part by Faculty Research and Professional Development Fund of North Carolina State University.

‡Department of Mathematics and Computer Science Emory University, Atlanta, GA 30322, USA. {atk,cheung,pwh,vss}@mathcs.emory.edu

Group communication is a natural paradigm for implementing collaborative work systems, but such systems require more than a simple multicast capability. CSCW applications can be demanding of the underlying communication subsystem both in terms of performance and flexibility. Applications may require point-to-point, multicast or broadcast capabilities and these services may be required at various ordering, reliability and throughput quality levels. Moreover, a single application may request a range of these services. For example, while audio/video conferencing tools may tolerate loss of data to maintain timely delivery, participant and control information must be transmitted reliably. In addition, many collaborative applications implement WYSIWIS ("what you see is what I see") interfaces that present all participants with a consistent view of the shared artifact (data or application). Simple reliable communication usually suffices for these applications yet occasional total-ordering communication for synchronization greatly simplifies implementation.

CSCW systems present the user with a rich media mix including audio and video. Such systems must support (soft) real-time multimedia transmission as well as collaboration and sharing. Consider an adaptive video conferencing tool [22, 16] as an example of a CSCW application with demanding communication requirements. In a fair and scalable transmission scheme, image quality can be modulated to match receiver capacity where capacity depends on processor speed, load and network bandwidth. Fairness and scalability can be ensured by splitting a single data stream into multiple components of differing video qualities. Receivers combine components to synthesize an image with a quality appropriate to the available transmission rate [17, 5, 22]. Receivers may reconfigure their quality of service by joining or leaving streams in response to changing network load. This scheme requires the underlying communication system to support multiple channels with independent membership and service qualities, and channel membership operations (joining and leaving) must be very efficient.

Application-sharing systems also demonstrate a need for multiple data streams with independent membership. Application multiplexors in the X Windows environment are typically implemented by interposing a "pseudo-server" between clients and servers. The pseudo-server assumes the role of an X server when interacting with X clients and the role of an X client when interacting with an X server. When input operations are restricted to a single host, an X pseudo-server provides "output multiplexing", allowing the output of a single application to be viewed simultaneously on multiple screens [1]. An alternate approach called "input multiplexing" [15, 20] replicates the client and employs a simplified pseudo-server. Output operations are passed to the local X server while input operations are multiplexed and communicated to remote replicas via the pseudo-server. A third approach periodically samples and transmits the client screen image. A CSCW environment can support all three mechanisms simultaneously by providing streams with different service qualities and memberships. In the spirit of the adaptive video tool, participants may subscribe to the service most appropriate for their needs. A single participant may even operate several applications simultaneously, each using a distinct application-sharing technique.

As a third example, group communication can be used to implement distributed systems composed of interacting objects with a resulting unification of naming and object invocation mechanisms. If each distributed object is associated with a unique multicast group, the group name can be used to uniquely reference the object. Method invocation on such objects can be implemented by joining the associated multicast group and requesting the appropriate operation. This scheme can be used to implement shared files systems or data spaces common in collaborative environments [26] but a large number of multicast groups may be required. Once again the common requirement of the communication substrate is the ability to support a large number of multicast groups with efficient, independent group membership operations.

In summary, (1) CSCW applications often require access to multiple data streams; (2) these data streams frequently require distinct service qualities; (3) membership should be defined on a per-stream basis; and (4) membership operations on these data streams (join and leave) must be efficient.

In this paper, we describe the design and implementation of a group-based communication system (CCTL) that efficiently supports the communication requirements of (small to medium scale) interactive collaboration involving 10 to 40 participants. CCTL is currently the communication layer of the Collaborative Computing Frameworks Project (CCF) under development at Emory University. CCF aims to develop an integrated framework supporting collaborative computing in natural sciences, although our tools and techniques are widely applicable [26].

CCTL is based on a two-level group hierarchy. Logical interconnections among processes, called *channels*, define an efficient and light-weight group mechanism. Channels support a variety of communication services, such as reliability, quality of services, and message ordering. Related channels are associated with a heavy-weight group, called a *session*, that provides group management services as well as a reliable multicast communication service. Sessions provide naming, membership and failure services for their associated channels. Several benefits result from CCTL's hierarchical architecture:

Resource saving and extensibility. We have seen that collaborative applications frequently require process groups to communicate and that these groups frequently require distinct communication services. It is also characteristic of collaborative applications that distinct process groups often require similar group management services such as naming, failure detection, and authorization. CCTL sessions efficiently provide common group management services while allowing distinct groups to communicate using channels with independent communication services. Sharing common services saves system resources that would be wasted replicating group management functions for each channel. In addition, this separation of concerns simplifies the implementation of new channel service qualities, improving the extensibility of the system.

Resource sharing. The CCTL architecture allows related channels to run on the same protocol stack if they provide the same communication service. The resources allocated to a channel, such as buffer spaces, bandwidth, and file descriptors, can be efficiently shared with other channels. Since not every channel actively uses the resources at a time, the resources can be time-shared among channels. A protocol stack of CCTL routes and multiplexes messages sent to different groups in order to minimize interference among groups running on the same protocol stack.

Separation of data and control paths. While channels may share a common protocol stack, channels and sessions run on different protocol stacks. This separates the data communication and group management control paths, preventing the potentially complex group management semantics provided by a session from adversely affecting the performance of its associated channels. This separation of data and control paths is important in building multimedia applications with stringent service quality requirements.

In the following section we survey related work. Section 3 introduces our system model and informally defines guarantees such as group management semantics that CCTL provides to applications. Section 4 describes the overall system architecture and details the membership protocols of channels. In Section 5 we describe our implementation of CCTL and in section 6 we discuss CCTL performance. In Section 7, we present our experience with a CCTL application. Section 8 qualitatively compares our work to related systems and sketch an implementation of CCTL using Horus. We conclude in Section 9 by discussing future work.

2 Related work

ISIS [3] pioneered multicast group communication and introduced the notion of virtual synchrony. Virtual synchrony allows consistent dynamic group membership while guaranteeing reliable message delivery. Failed processes appear to leave groups they are participating in. ISIS provides good support for distributed systems that require high reliability.

RMP (Reliable Multicast Protocol) [28] provides various per-packet service qualities. Messages may be unordered, FIFO, causally or totally ordered, or unreliable. Streams supporting differing service qualities may be embedded in a single group but all streams in a single group share a common membership. Streams with distinct service qualities may be implemented using independent RMP groups. However, this approach is inefficient because each group independently performs the same type of group management services. RMP provides no optimization for sharing resources (name service, fault-detection, etc.) among related groups.

Transis and Totem [2, 18] extend virtual synchrony to tolerate network partitioning. Like ISIS, these systems are designed for applications that require high reliability. The existing system offers limited communication services to support collaborative and multimedia applications. A recent proposal [6], called *Multimedia Multicast Transport Service (MMTS)*, provides additional flexibility. Reliable messages are used in a novel way to coordinate and synchronize streams offering differing service qualities. Collections of coordinated streams are called *bunches*. However, the proposal does not allow distinct stream memberships among bunches. A collaborative tool that relies on multiple communication service qualities with varying membership, such as layered video tools [17, 5, 22], would be difficult to implement.

Horus [27] succeeds ISIS providing modular and extensible group communication with many service qualities. Horus efficiently supports multicast groups with differing needs and allows run-time reconfiguration. Horus allows the creation of multiple streams with independent service qualities but suffers the same inefficiencies as RMP because resources cannot be shared among related groups. Light-weight groups (LWG) can be used to share resources among multiple groups, but they can be used only for virtually synchronous reliable multicast groups. In addition, Horus membership protocols assumes an underlying reliable transport making it very difficult to provide a membership service for groups with (less-reliable) communication service quality. For instance, the membership layer requires reliable message delivery protocol stacks (such as NAK). This dependency limits the flexibility of group protocol construction. Suppose that a programmer wants to build its own communication service (such as a real-time protocol), but wants to use the membership service of Horus. The programmer would have to build his/her own service on top of Horus membership stack, which requires a reliable protocol stack. This implies that every data message incurs the overhead of executing the reliable protocol stacks of Horus. The LWG implementation of the Horus inherits the same limitation. Since Horus LWGs are mapped only to a virtually synchronous reliable group (heavy-weight group), all LWGs pay the same overhead of executing the protocol stack of their HWGs.

Ensemble [9] is the next generation of Horus, implemented in a functional language called OCAML. Inspired by CCTL, Ensemble has recently adopted a membership coordination system [4], called *Maestro*, that provides common membership services for process groups running distinct communication protocols (including non-Ensemble protocols).

3 Model and definitions

Our informal system model follows. See [21] for a more formal treatment.

Processes $p_i \in P$ *send* and *receive* messages to and from groups $g_i \in G$. Send and receive are *asynchronous*. Each message m is associated with exactly one group g. Processes *join* and *leave* groups dynamically. Groups are identified by their membership. A member list is called a group *view*. Groups initially have no members. A process may belong to zero or more groups. Processes fail by halting detectably (crashing). Crashes are identified by a correct but unspecified failure detector. That is, all non-faulty processes eventually detect failed processes. Processes are removed from all groups on failure. As processes join and leave groups, a *view sequence* results.

Processes communicate exclusively by sending and receiving messages with many attributes. Some attributes actually appear in our implementation while others are defined for notational convenience. All messages are sent to a particular group g by a particular sender p. All messages are assumed unique and can be distinguished by a sequence number.

We distinguish the "receipt" of a message via the network from the "delivery" of a message to an application. Applications use send and receive to request service from CCTL. We say that CCTL "delivers" a message that may be subsequently received by an application. CCTL *implements* these operations using standard network transport services.

There are four types of messages: *fail*, *join*, *leave* and regular messages. A process is notified of the crash failure of another process on receipt of a fail message. This message is generated by the (unspecified) failure detection mechanism. A process requests to join a group by sending a join message to that group. The *delivery* of a join or leave message is considered a *view change* event and alters the receiving process's view of the group. Normally the sending process is added or removed from the group view. Join messages include the *current group view* when delivered. Similarly when a process receives its own leave request, its view of the group returns to the empty set. The *sending view* of a join request contains only the sending process. When a process receives its own join request, the *receiving view* is the initial view supplied with the message. The receiving view of a regular join or leave message is the view that results from adding or deleting the sending process.

CCTL offers three types of delivery ordering: *atomic*, *FIFO* and *unordered*. FIFO ordering ensures that messages sent to process q by process p are received in the same order in which they were sent. FIFO guarantees "point-to-point" ordering but places no constraint on the relative order of messages sent by p and q when received by a third process r. Atomic, or total ordering ensures that all members of a group receive all messages sent to that group in the same order in addition to providing FIFO ordering.

CCTL offers both *reliable* and *unreliable* message delivery. Reliable delivery guarantees that messages sent by a non-faulty process are eventually received (exactly once) by all non-faulty processes in some designated *destination set*. In a group communication system, this set can only be defined in relation to view change operations (*membership protocols*).

CCTL offers two membership protocols: *relaxed virtual synchrony* and *reliable membership*. Both protocols order membership operations on a given group. Relaxed synchrony totally orders membership operations so that participating processes see identical view sequences for the group. Virtual synchrony also guarantees that if a non-faulty process receives a message in a particular view, then all non-faulty processes in that view will receive the same message during that view. Finally, relaxed virtual synchrony relates the view in which a message is sent with the view in which it is received. Relaxed synchrony guarantees that messages sent by non-faulty processes will eventually be delivered but allows messages to be received in *any* subsequent view containing the sender.

Reliable membership provides relaxed synchrony for membership operations only. Since regular messages may be lost in unreliable channels, ensuring ordering between membership changes and regular messages is meaningless. Participating processes may view differing sequences of membership operations, although all of the operations performed by a single process appear in FIFO order.

4 Architecture and Implementation

4.1 Hierarchical group architecture

The group paradigm is commonly used in distributed environments for sharing information among related processes. Multicasting makes information known to all group members. As previously mentioned, distinct groups may have differing communication service quality requirements. For example, real-time data may require bounded delay while financial data may require reliable and secure transfer.

Collaboration participants often use a variety of tools (audio, chat, clearboard, etc.) in a given session. Tool users are always a subset of session participants but each tool may have a distinct set of users. This structure of collaboration sessions allows significant optimizations in the group communication system implementation. Sharing group management services among related groups in a session provides an elegant, modular implementation of groups and saves system resources. Shared services can be provided by a heavy-weight mother group while communication services can be efficiently implemented by light-weight child groups.

This modular construction allows efficient resource sharing among groups. When a process group is created, needed resources such as bandwidth, buffer space, and file descriptors are allocated in order to provide the requested quality of service. If child group resource usage can be predicted, the mother group can pre-allocate resources and maintain a shared resource pool. If resource usage is not continuous, the resources can be time-shared among multiple child groups. Allocating resources per mother group also reduces allocation overhead.

Our implementation of LWGs (child groups) is based on the premise that the execution of group management services should not interfere with data communication. We allow the communication protocols of LWGs to run on different protocol stacks from those of their HWGs. Thus, the overhead of executing the protocol stack of a HWG is incurred only when a service from HWG is required.

In CCTL, LWGs are called *channels* while HWGs are called *sessions*. A session group corresponds to a collaborative session and includes every process that participates in the collaborative session as a member. A channel is statically mapped to a session at its inception and channel membership is always a subset of session membership. A session provides a default, virtually synchronous reliable multicast service among participants that is used to implement group management services.

Figure 1 shows the CCTL architecture. CCTL is logically implemented as a group module, interposed between applications (clients) and the physical network. This module implements the CCTL API and provides session and channel abstractions to clients. The group module consists of *channel membership* (CM), *communication* (COM), and *session* sub-modules. The session module supports the services that a session provides. The CM module implements membership protocols for channels using the reliable message delivery service of the session module. The COM module implements data communication services of channels.

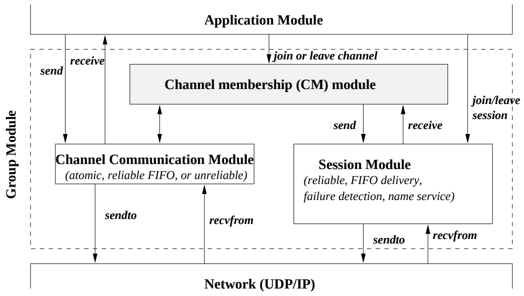

Figure 1: CCTL Architecture

The session module implements a virtually synchronous reliable multicast group which provides group management services, such as virtual synchronous membership, failure detection, and authorization. The session module can be easily implemented using other group systems, such as ISIS, or Horus. Applications can join and leave a session through the API of the session module, and the CM module can communicate with the CM modules of other session members through the session module. The session module notifies the CM module of session membership updates including a member failures. A process must join a session to participate in a collaboration session. Every channel created after session join becomes a child group.

The CM module implements membership protocols for different types of channels. It uses the session module to communicate with the CM modules of other session members, and uses the COM module to enforce (relaxed) virtual synchrony for the channels that require virtual synchrony (e.g., reliable channels). If a channel needs virtual synchrony, the application module uses the API of the CM module to receive and send messages. If it does not, applications send and receive messages directly through the COM module.

A CCTL channel currently supports one of three data communication services: total-ordering reliable, FIFO reliable, and unreliable. These services are implemented within the COM module. CCTL does not yet provide any QoS guarantee on communication, but its architecture is amenable for incorporation of such a service as a new channel type; a new channel type can be easily added to the COM module because its group management services are available from the session module and the CM module.

4.2 Implementation

In this section, we discuss the implementation of the hierarchical architecture of CCTL. We focus on the implementation of the channel membership and communication modules. The session module provides a virtually synchronous FIFO reliable multicast service commonly provided by other group systems (such as ISIS, HORUS, and Transis) so we omit details of its implementation.

4.2.1 Implementation of the channel membership (CM) module

The CM module implements three different membership protocols, one for each channel type. While the membership protocols for total-ordering and FIFO reliable channels provide a variant of virtual synchrony, the membership protocol for unreliable channels only guarantees reliable transfer of membership updates, which can be used for most of QoS supported channels (when they become available).

Using the reliable multicast service of the session module, the CM module implements total-ordering delivery of membership protocol messages as follows. The CM module of the oldest session member, the *session owner*, acts as a message sequencer, receiving and resending protocol messages through the session module. Other session members simply receive messages in the order sent by the owner.

The total ordering of membership messages simplifies the implementation of membership protocols. The CM module of each session member maintains a local database of channel information, called the *channel table*, containing channel name, channel ID, multicast addresses, current membership, etc. Since all messages updating channel information are multicasted to session members using a total-ordering channel, the channel tables replicated by all session members remain consistent. Channel tables are used to provide name services for channels and to facilitate channel joining operations. Since channel information is fully-replicated and available locally name lookups are trivial.

Membership for unreliable channels

Unreliable channels are useful to applications such as audio and video tools that require timely delivery but that can tolerate some message loss. Strong guarantees such as virtual synchrony are too costly and unnecessary for such applications. It suffices to guarantee that a membership change is eventually recognized by every channel member. Unreliable channels provide this minimal guarantee on the membership operations.

A process p wishing to join an unreliable channel calls the channel join call entry in the channel module which sends a join message through the session module. Since the message will be rebroadcast by the session owner, the channel module will receive its join message totally-ordered with respect to other membership messages. On receipt, each process updates the replicated channel table.

After receiving channel join, the channel is created if it does not appear in the channel table. Creating an unreliable channel involves allocating a multicast address from a common resource pool. Since all channel updates are totally-ordered, no two processes choose different channel addresses for the same channel. If the channel exists already, the joining process gets the multicast address of the channel from the channel table, and simply uses this address for data communication. The leave operation is also done by simply multicasting a leave channel message through the session

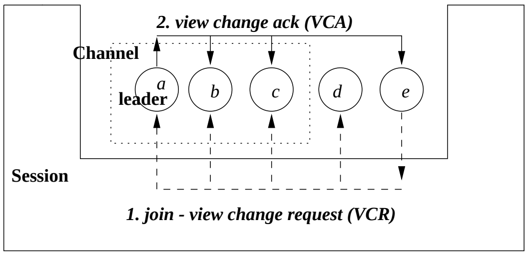

Figure 2: Dotted arrows - atomic session multicast; solid arrows - atomic channel multicast.

module. The CM module of each session member updates its channel table. If there is no member for a channel, the CM removes the entry for the channel from the channel table and deallocates the multicast address. Note that the membership protocol works even if some processes leave the channel during the channel join because all membership updates (join and leave) are totally-ordered and channel tables are maintained according to the delivery order of update messages.

Membership for total-ordering channels

We now sketch a membership protocol for total-ordering channels. We assume a total-ordering multicast primitive, omitting its description because efficient protocols are well known [3, 28]. Process p multicasts a join view change request (VCR_p) through the session module. The oldest channel member q receiving the join request (the *leader*) sends a view change acknowledgment ($VCA(p)_q$) to non-faulty channel members (and p) on the same channel being joined. This acknowledgment will be totally ordered relative to regular messages because the channel being joined totally orders every message.

If the channel p wishes to join contains no members, then we say p is the *creator*. p determines if it is the creator by consulting the channel table after receiving its VCR message. If p is the creator, the join terminates and the channel view contains only p. Otherwise, p waits for a VCA message from the leader, containing the correct membership. On receipt of VCA all channel members update their views and deliver the join. Leave is similar: p is removed from the channel on receipt of $VCA(p)$.

Figures 2 and 3 illustrate the join protocol. Processes a through e are session members. Process e is trying to join a channel containing members a, b and c (a is the leader). Note that regular messages m and m' are totally ordered with respect to VCA_a. This ensures that all processes receive all messages (including VCRs) in the same order.

To enforce virtual synchrony, all processes must receive the same messages in a given view. Relaxed synchrony additionally allows the sending and receiving views of a message to differ. For example, in Figure 3 the receiving view of m' contains e; the sending view does not. To guarantee virtual synchrony, the sender p compares the sending and receiving views for each message sent. p

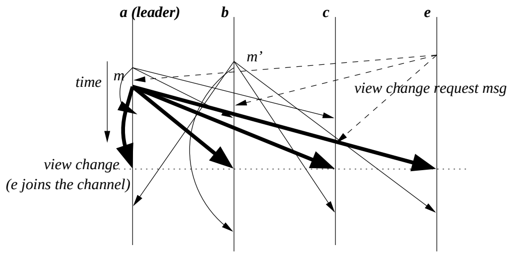

Figure 3: Regular messages m and m' ordered with respect to a view change message (thick arrows).

retransmits messages to the processes that are in the receiving view of the messages but not in the sending view of the messages.

Membership for reliable channels

View changes and regular messages must be explicitly ordered on reliable channels since total-ordering is not available (note that view changes are different from view change requests which are sent through the session module). To join or leave a channel C, p first sends VCR_p through the session module. All channel members respond by sending $VCA(p)$ on the reliable channel C. On receipt of $VCA(p)_r$ ¿from a process r, processes delay delivery of subsequent messages from r until p's view change request is acknowledged by all non-faulty channel members. The delayed messages will be delivered after the view change. Since the VCRs are totally ordered and the channel implements a reliable (FIFO) multicast, every message that a process sends after its transmission of VCA, is delivered after the corresponding view change. Thus, the protocol guarantees that within the same view, all (non-faulty) members will receive the same set of messages.

Handling failures

When process p fails, the session module of every non-faulty session member will deliver a fail notification of process p ($fail_p$) to the CM module according to virtual synchrony enforced by the session module. Since p may have failed while sending, some channel members may have not received all messages sent by p. Before removing p from the channel, channel members must decide which messages from p are to be delivered. We use a *flush* procedure, similar to that used in ISIS [3]. A message received at all its intended destinations is said to be *stable*. Receipt of $fail_p$ causes a process to retransmit all messages received from p that are not stable. Each process then receives and acknowledges retransmitted messages from other channel members. Retransmitted messages acknowledged by all members are considered stable. When all retransmitted messages are stabilized process q sends $flush_p(q)$ to all channel members. When $flush_p$ is received from all channel members the flush protocol terminates.

When a channel leader fails a new leader must be elected. Leaders are selected by seniority (join order). Assume q is the current leader and r is the second oldest channel member. On receipt of $fail_q$, r flushes. If *VCR* messages remain unanswered, r assumes leadership and responds with VCA_r. Conflicting *VCA* messages are not possible because new leaders must flush before assuming leadership.

Similar protocols [3, 23, 10] flush at *every* view change. This is because they requires messages to be delivered in the same view sent. Before a view change, they need to "flush" all the messages sent in the current view. In our protocol, flush is required only when some process fails to force agreement on which messages of the failed process should be delivered. Because relaxed synchrony allows messages to be delivered in any view as long as the same set of messages are delivered within the same view, normal view change operations do not require flush, and the receiving view of a message is determined by the delivery order of the message and a *VCA* message from the channel leader.

Flushing delays (blocks) view changes and delivery of subsequent messages. ISIS, Transis, and Totem even delay message sending to ensure that flush terminates. Our protocol, however, does not delay message sending during normal view changes (those not initiated by *fail*). Message delivery is delayed in a reliable channel only for the messages that are sent after a VCA is sent, but are received before the corresponding view change (which happens after a process receives a VCA from all non-faulty processes in the channel). See [21] for a detailed description of the protocol.

4.3 Implementation of the communication (COM) module

CCTL is essentially middleware that supports the communication needs of collaborative applications. Since CCTL runs in user-space, it must be sparing in its use of system resources such as heap and file descriptors.

The COM module allows effective resource sharing among related channels by taking advantage of the hierarchical group structure of CCTL. When a process joins a session, the COM module allocates a pool of buffers. Channels use buffers from this pool to store packets that are not acknowledged or cannot be delivered immediately. Channels may also create their own buffer pools to ensure fairness. Since channels typically alternate between periods of activity and inactivity, the shared session buffer pool effectively optimizes resource usage.

The COM module implements a protocol stack for each channel type. Multiple channels of the same type may share a protocol stack or they may create their own. Channel ID is included in messages sent through the protocol stack and the COM module multiplexes messages to the appropriate channel based on this ID. The COM module discards messages not intended for the current process. Sockets are allocated on a per-stack basis so sharing protocol stacks saves descriptors.

Each protocol stack uses bounded IP multicast (limited to the current subnet). If a channel contains members on two or more isolated subnets, messages are sent to the remote subnet via a UDP tunnel and then multicast locally. Figure 4 illustrates CCTL's routing mechanism. A limited form of pruning is implemented. When a packet does not contain a member in a remote subnet, its

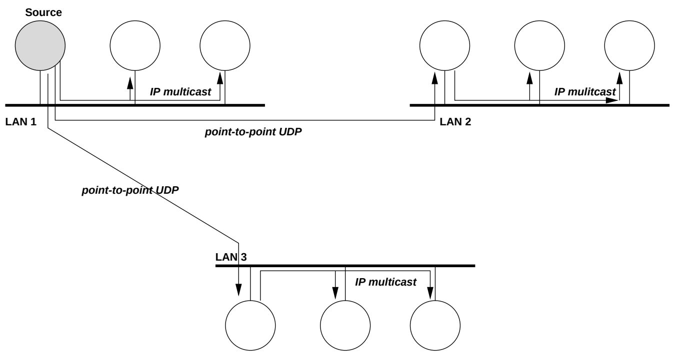

Figure 4: CCTL routing

stack does not send the packet to the subnet. This technique reduces interference between channels sharing a stack but with distinct memberships. We did not use MBone for multicast over WAN because there are sites that do not support MBone yet, and the current CCTL routing mechanism is sufficient to support small scale collaboration sessions.

A reliable channel implements message fragmentation and reassembly, retransmission, and flow and congestion control. Each reliable channel contains two queues, one queue to store messages to be transmitted (send queue), and the other queue to store messages to be delivered to the application module (receive queue). The protocol stack of a reliable channel is implemented through three threads: *sender*, *receiver*, and *acker*. The sender thread takes messages from the send queue, and fragments them into packets. The fragments are transmitted using the CCTL routing protocol. The rate of transmission is determined by flow and congestion control. CCTL adopts a TCP-like flow and congestion control scheme with one difference to TCP: flow and congestion windows are updated when each packet is acknowledged by all of its receivers. The receiver thread acquires packets through a multicast socket and reassembles fragments into messages. Based on the channel ID contained in each message, it multiplexes the message into the receive queues of channels. It sends acknowledgments to the acker thread of the source. The acker thread collects acknowledgments and passes them to the sender thread which adjusts its flow and congestion windows based on these acknowledgments.

Channels providing total-ordering of all messages are implemented using a simple token protocol on top of reliable channels. All messages are sequenced by the token holder. Consecutive sequence numbers define the delivery order. The current token holder may send messages (tagged with appropriate sequence numbers) with no additional communication required. These messages are multicast on a reliable channel and are eventually received by all non-faulty participants. When a non-token holder wishes to send, the message is multicast and tagged as a *token-request*. All participants buffer the unsequenced message for later delivery. When the token holder receives the request, a *token-transfer* message is multicast, sequencing the message containing the token-request and transferring the token to the requester.

CCTL provides join and leave notifications as messages on reliable channels. These notifications are properly sequenced by virtual synchrony but must be totally-ordered with respect to data messages. A join notification is also considered to be a token-request and results in a token-transfer to newly joined members. The token-transfer effectively sequences the new member join. Similarly, leave notifications are sequenced by the token holder (without transferring the token).

CCTL also assigns unique numerical ids to all channel participants. Token holders leaving the channel must participate in a leave protocol that transfers the token to the channel member with the smallest id. Several pathological cases are possible. For example, the member with the smallest id may also be attempting to leave. The leave protocol in conjunction with virtual synchrony ultimately guarantees that some remaining member will receive the token. If a member fails while holding the token (or, equivalently, leaves without participating in the leave protocol) a recovery protocol requiring participation of all channel members will correctly reassign the token.

The total-ordering protocol is implemented by a per-channel receiver thread that interposes protocol control messages with the data sent on the underlying reliable channel. Buffers containing unsequenced messages and sequencing information are maintained. When an unsequenced message is matched with its sequencing information, the message is added to the delivery queue.

Unreliable channels implement RTP-like transport services on top of the CCTL routing protocol. They perform packet sequencing, loss detection, and time stamping. Each unreliable channel allocates a separate socket file descriptor to avoid the message-filtering overhead, and maintains a separate pool of buffers to avoid other channels from monopolizing buffers.

5 Performance Evaluation

We conducted throughput and latency measurements of CCTL channels under various execution scenarios. This section presents the results of the experiment over a 10 Mbit and a 100 Mbits Ethernet LANs. Since WAN tests yield highly variable results it is difficult to meaningfully compare performance data for two systems such as TCP and CCTL across the Internet. We have omitted WAN tests from this discussion. We used a collection of Sparc-20s and Ultra-Sparcs for the experiment.

The main intent of this experiment was to quantify the computational overhead of CCTL (recall that CCTL runs in application address space). We do this by comparing CCTL and TCP performance. Note that we are not comparing CCTL multicast to TCP unicast to demonstrate the obvious point that CCTL scales better than TCP. Rather we seek the point where CCTL outperforms TCP. At this point, the benefits of multicast compensate for CCTL overhead and make CCTL a viable alternative to TCP for collaborative applications.

We first measured the throughput of reliable channels with various group sizes. In this experiment we measured one-way throughput produced by a single process sending continuously on a channel to multiple receivers on distinct remote machines. We compare the throughput of TCP when we run the same number of TCP connections as the number of receivers in a channel. Figures

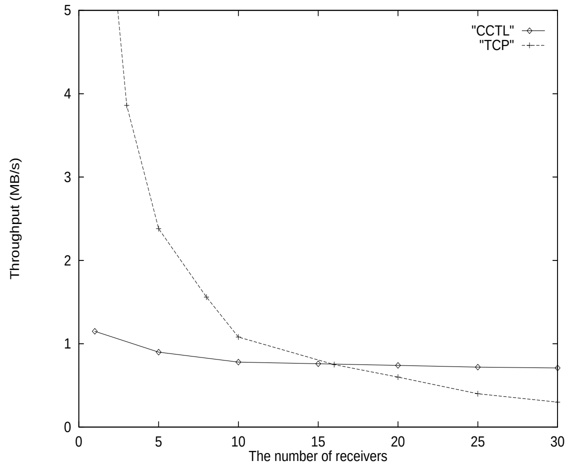

Figure 5: Throughput: CCTL vs. TCP (100 Mbit Ethernet)

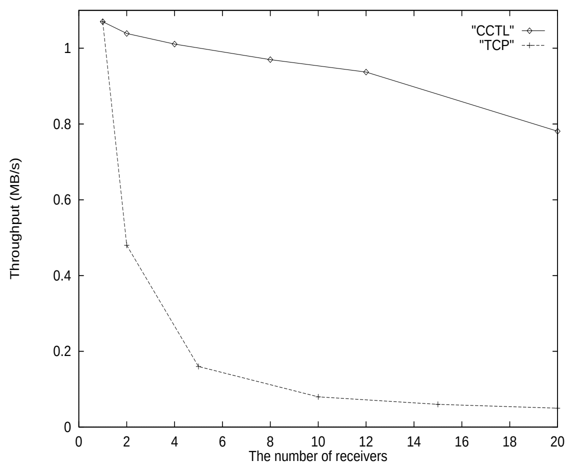

Figure 6: Throughput: CCTL vs. TCP (10 Mbit Ethernet)

5 and 6 shows the result. When bandwidth is not a bottleneck, the CPU overhead of CCTL is more pronounced as shown in Figure 5 where the throughput of TCP is about 7 to 8 times better than CCTL with few receivers. However, TCP's throughput reduces rapidly as the group size increases because TCP uses more bandwidth, and bandwidth becomes bottleneck. Around 15 receivers, the throughputs of TCP and CCTL break even. The adverse impact of bandwidth to throughput is clearly visible in Figure 6 when transmission is done to a 10 Mbit Ethernet. Thus, if an application is bandwidth-intensive, CCTL can provide a substantial performance improvement when the available bandwidth is less than about 1 MB/s.

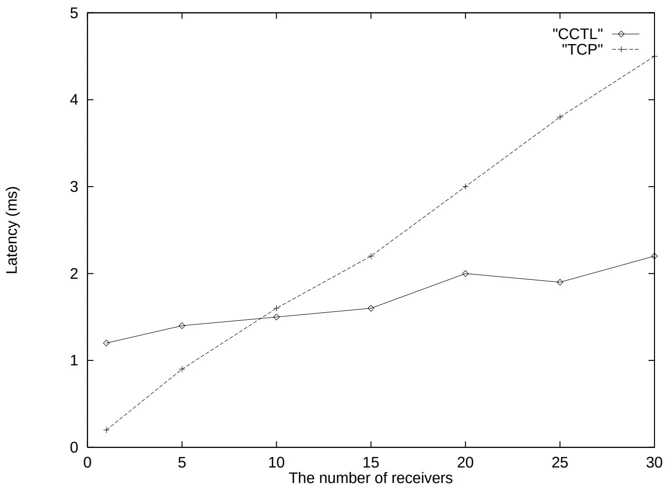

Figure 7: Latency: CCTL reliable channel vs. TCP

We also measured the latency of CCTL reliable channels with various group sizes. In this experiment, the sender sends a message (100 bytes) to all receivers in a channel, and one designated receiver in the channel responds to the message by sending a reply message (100 bytes) back to the sender. The round-trip latency is recorded by the sender on receipt of the reply. Figure 7 reports the mean latency for 50,000 trials for both CCTL and TCP. With less than 10 receivers, TCP runs faster. However, as the number of receivers increases beyond 10, TCP latency is much higher than CCTL. This is because TCP has to send a unicast message to each receiver while CCTL multicasts. The latency degradation of CCTL is partially due to the sender's overhead to handle acknowledgments from all the receivers. Note that acknowledgments are needed to guarantee reliable data transfer to all the members.

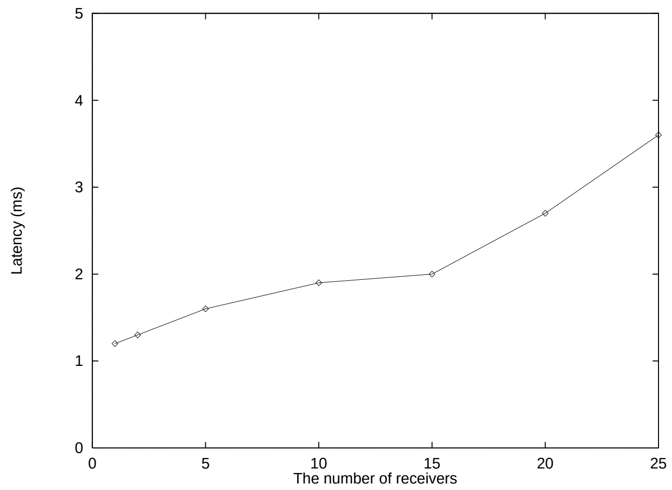

Figure 8: CCTL reliable channel latency under membership changes

CCTL reliable channels implement virtual synchrony. The overhead of virtual synchrony is incurred only during membership changes. Figure 8 shows the impact of virtual synchrony on message latency. While one member continually joins and leaves a reliable channel, another member sends a message (100 bytes) to all members in the channel, and one designated receiver sends a reply message back to the sender through the channel. The same latency measurement as in the previous experiment is conducted. CCTL message latency increases as expected with frequent membership changes since message delivery is delayed until a membership changes are acknowledged by all channel members. Latency increases in relation to the number of channel members. However, this overhead is not significant because it happens only during membership changes and the resulting latency still beats TCP latency.

We also performed similar experiments with unreliable channels. Recall from Section 4.2.1 that unreliable channels run with reliable membership semantics where membership changes are reliably delivered to every channel member. No guarantee on message delivery ordering with respect to membership changes is required. The reliable membership semantics are implemented through the session module which provides a (FIFO) reliable message delivery service. Since unreliable channels do not carry any membership change message, the latency of unreliable channels is not affected by membership change. Furthermore, the latency is not affected by the number of receivers either because the channels do not enforce reliability on the data being transmitted. We observed about 0.2 ms (round-trip) latency for unreliable channels on the LAN.

6 Case Study: CCFX/CCSM – A distributed X multiplexor

In this section we describe a major component of CCF that was implemented using CCTL and draw lessons from this experience concerning the flexibility and performance of CCTL.

One aspect of collaborative work is the sharing of existing applications, enabling researchers to view, discuss and make interactive changes to computer generated data and images during their study. It is important for an application-sharing system to support existing (often proprietary) applications without modification so that collaborative workers are not disadvantaged by having to use specially-designed, collaboration-aware applications only. We have developed the Collaborative Computing Frameworks X-Multiplexor (CCFX) and the Collaborative Computing Session Manager (CCSM) as such a system.

Our design of CCFX/CCSM had three objectives: the ability to share well-behaved, off-the-shelf X applications; support for heterogeneity (including various machine types and display architectures); and data transport efficiency. Sharing complex graphics applications can use significant network bandwidth and low-latency is important for the usability of the system. We devised a distributed architecture that takes advantage of CCTL reliable multicast to assure transport efficiency.

X clients connect to a CCFX which acts as a proxy X server. CCFX typically runs on the same machine or perhaps the same LAN as its client applications to reduce latency. CCFX responds immediately (locally) to client requests when possible. Otherwise it forwards X requests to CCSM in an efficient fashion and collects and forwards responses and X events to its clients. CCFX forces its clients to operate in a known state by creating a virtual X server with known properties. However, because the client may then operate in a mode not compatible with the native server, a front end, CCSM, is needed to convert the client's X requests to be compatible with the native X server.

In addition to processing the X stream, CCSM creates a "desktop" window on the local display. CCSM draws all the windows, supplies a user interface for starting up applications, and ensures that all desktops have an identical appearance. The input control is per client, managed by CCFX, with user interface supplied by CCSM. All CCSMs in a session function collectively as a distributed window manager, maintaining WYSIWIS throughout the session. In essence, CCFX acts as a server to the client program while CCSM renders the images on the user workstation. CCTL is used as a communication backbone for CCSM and CCFX. Since CCTL provides multicast, a message from CCFX can be efficiently delivered to all CCSMs to maintain the same desktop view.

Although there is a one-to-one correspondence between a user and a CCSM, there is no such relationship between CCFX and CCSM. That is, the system supports an arbitrary number of

CCFXs and CCSMs with each CCSM representing a user's display and each CCFX serving local clients. CCTL's dynamic membership allows CCFXs and CCSMs to be efficiently added and removed from the session.

To reduce multiplexing costs, CCFXs and CCSMs join several CCTL reliable channels, each of which carries a different type of data. Channel *ccfx* carries X protocol messages sent from a CCFX to CCSMs while channel *ccsm* carries X protocol messages sent only from a CCSM to CCFXs. In the CCFX/CCSM architecture, no X protocol messages are exchanged from a CCSM to other CCSMs, or from a CCFX to other CCFXs. Thus, for X protocol messages, CCSMs listens only to channel *ccfx* while CCFXs listen only to channel *ccsm*. Thus CCSMs need not receive messages sent to CCFXs and vice versa. CCSMs also share another channel used for sending session control information.

6.1 Quantitative Experiments

We have recently released an Alpha version of CCF and have collected some preliminary performance data.

We ran a series of three benchmarks based on a slightly modified version of the standard X benchmark XBENCH[13]. We compared CCFX/CCSM performance to the performance of a local client, a remote client, and XMX on a LAN, and measured CCFX/CCSM scalability and performance across a WAN. XMX is a simple, centralized X-multiplexor, adding little overhead in multiplexing the X stream; thus it serves well as a baseline in comparison to our distributed architecture.

For our experiments, we chose three benchmarks from XBENCH–*line2*, *line400*, and *arcs400*. *line2*, *line400* and *arcs400* are standard XBENCH tests which draw a series of lines of width 2, lines of width 400, and arcs of width 400. These experiments give some indication of performance as the system becomes less dependent on network bandwidth and more dependent on local X-server processing time.

In the LAN experiments, we ran the benchmarks on an assortment of UltraSparcs and Sparc-20s connected via a 100 Mbit Ethernet LAN. The quad-processor machine was used to execute the client and (where relevant) the CCFX. Individual CCSMs and XMX were executed on the uniprocessor machines. CCFX/CCSM processes the X protocol stream twice (CCFX first receives the stream from a client and then multicasts a modified stream to all CCSMs in the session for further processing) while XMX, since it is centralized, processes the stream only once. As expected, the X benchmark experiments *line400* and *arcs400* show that CCSM/CCFX is about twice as slow as XMX. CCTL session startup overhead is considerably more than TCP so some experiments show CCFX/CCSM to be as much as 8 times slower than XMX but this only holds for sessions with a small number of participants. Experiments requiring significant data transfer highlight the strengths of CCFX/CCSM. The performance of *xmx* rapidly degrades as additional participants are added (more TCP connections are required) but the performance of CCFX/CCSM degrades only slightly since there is little overhead in adding additional multicast recipients. CCFX/CCSM begins to outperform XMX when the number of participants exceeds 12 for communication intensive benchmarks.

In a qualitative WAN experiment, we observed that CCSM/CCFX performance is better than XMX for display intensive applications (we ran XV and RASMOL which typically send large bitmap data to X-servers). This mirrors and confirms the results directly comparing CCTL to TCP in Section 5.

While the XBENCH results for CCFX/CCSM show some room for improvement, subjective response to the application sharing framework has been positive except for very high-bandwidth, bitmap-intensive applications. Even these applications are usable over all but the slowest network connections although the delay is palpable. We hope to incorporate additional application-driven optimizations such as compression in the coming months to improve performance for this class of applications.

7 Comparison with other group systems

In this section, we compare CCTL to Horus [27, 10], Transis [2, 6] and RMP [28] and show how to build CCTL using a Horus protocol stack.

7.1 Multiple service options in Horus

Horus group protocols can be dynamically composed by stacking sub-protocol layers such as: COM, NAK, MBRSHIP, STABLE, WVS and TOTAL. COM interfaces to standard network protocols such as UDP and IP-multicast and NAK provides reliable FIFO unicast and multicast. MBRSHIP implements virtual synchrony and failure detection and recovery while WVS implements weak virtual synchrony [10]. Finally, STABLE detects message stability and TOTAL totally orders messages. Additional layers (FC, CAUSAL and FRAG) provide reliable transport services.

Various service qualities can be composed by stacking layers, but adding membership services to QoS groups can be expensive in Horus. Membership layers (MBRSHIP, WVS) require a reliable transport service (NAK or STABLE) lower in the stack, forcing regular messages to be sent reliably. Unreliable groups with reliable membership services are unnecessarily costly.

7.2 Multiple service options in Transis

Transis has been elegantly extended [6] to support *quality of service multicast channels* (QMC). Figure 9 shows a reference implementation of QMC using Horus protocol stacks. All session processes belong to a special atomic multicast group called the *reliable multicast module* (RMM) that supports channel membership and synchronization. Unreliable channels simply use a COM layer.

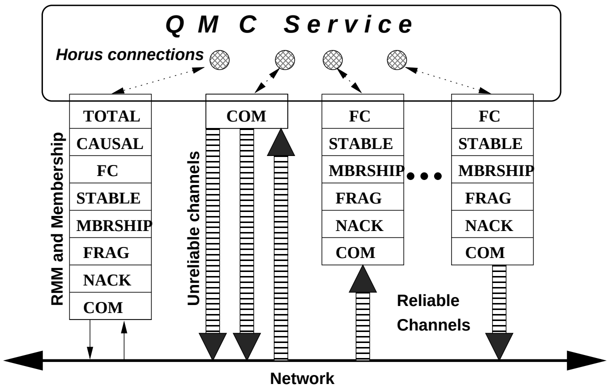

Figure 9: QMC in Horus

Reliable channels use the standard Horus reliable multicast stack. This architecture allows applications to synchronize across channels.

The Horus implementation of QMC has two weaknesses. First, unreliable channels may not have distinct memberships. Unreliable channels share membership with the RMM so messages sent are delivered to every process joining the RMM group. Second, this architecture may not support a large number of reliable channels efficiently because each channel uses a separate Horus connection. Each connection uses a membership layer with a polling failure detector resulting in significant overhead.

Scalability can be improved by using the Horus light-weight group layer (LWG) but inefficiencies still exist. Adding the LWG layer on top of the reliable multicast stack produces a *core* group. This core group multiplexes several distinct light-weight groups (with distinct membership) using a single reliable Horus group. Light-weight group messages are wrapped in regular messages and sent to every core group member. The LWG layer in a receiving process discards messages not intended for that process. This approach increases the cost of multicast and generates unnecessary network traffic.

7.3 Multiple service options in RMP

RMP is a token-based virtual ring protocol where the token holder is responsible for acknowledging and ordering messages. A process accepts the token only after receiving all previous messages. RMP provides quality of service on a per-packet basis, and guarantees virtual synchrony only on atomic multicast packets. RMP can support multi-grade QoS in two different ways but neither is is entirely satisfactory for CSCW applications. It is possible to multiplex different data streams using a single RMP group but this approach has two disadvantages. First, applications suffer the cost of demultiplexing the various streams. Second, all streams are received by all members. It is also possible to use distinct RMP groups for each service quality but this has the same drawback as using multiple heavy-weight groups in Horus.

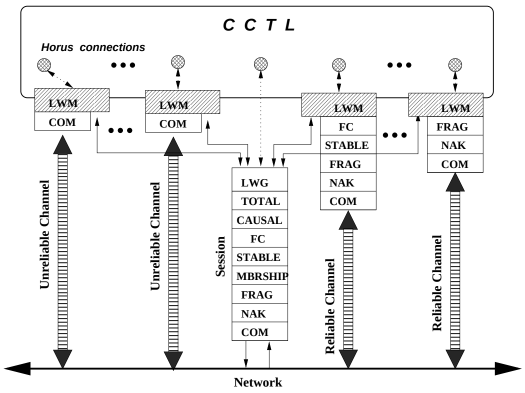

Figure 10: CCTL in Horus

Join and leave operations in RMP are costly and these costs increase almost linearly with the group cardinality. New members immediately acquire the token. Since one of the roles of the token holder is to retransmit messages, the new member must first acquire the last n messages where n is the number of members in the group. (According to the RMP protocol [28], the token will traverse the virtual ring completely after n messages have been sent to the group and accordingly, members will only request transmission from the last n messages sent.) Leaving members reliably multicast a *delete member* message. The leaving member must remain in the group and continue to receive messages until its predecessor in the ring sends a *new token list* message. But the predecessor sends this message only when it has the token. For a process to receive the token, it has to wait in most cases $\frac{n}{2}$ message delays, which will be added to the latency for the leave operation. This inefficiency is especially problematic in multimedia applications where members decide to abandon a stream because of congestion. Delaying the leave operation in this way unnecessarily prolongs and worsens congestion.

7.4 Implementing CCTL using Horus

CCTL can also be implemented using Horus (Figure 10). We propose a new layer, called *light-weight membership* (LWM), supporting membership operations for different service qualities. LWM implements the protocols described in Section 4 and can be used with other Horus stacks to provide various ordering and reliability guarantees. LWM uses QMC's RMM to effectively implement the CCTL default session channel. Unlike MBRSHIP, LWM does not include a failure detector and does not require other layers such as NAK and STABLE. Regular channel messages incur overhead only for the underlying transport stack. To avoid multiplexing CCTL channel control messages (VCRs and VCAs) through the RMM, each channel may join a light-weight group, supported by stacking LWG on top of RMM. Adding LWM to various protocol stacks produces a rich set of service qualities with tailored membership guarantees. Unreliable channels use the stack: unreliable LWM and COM. Atomic channels use the stack: atomic LWM, TOTAL and COM.

Our implementation eliminates the shortcomings cited above. First, channels have membership protocols tailored to the service quality offered. Second, the architecture is scalable to a large number of groups. Common services such as failure detection, naming, and view change ordering are shared via RMM. Third, channel data is transmitted more efficiently because non-channel members are not involved.

8 Discussion

We have described the design and implementation of CCTL, a flexible, efficient, scalable group communication architecture. CCTL supports CSCW and multimedia applications requiring a variety of ordering and reliability guarantees. We exploit the structure of typical CSCW applications to implement an efficient and scalable light-weight group system architecture. The two-level structure greatly simplifies the implementation of common CSCW components. Our two-level architecture is particularly suitable when a single application requires simultaneous access to a variety of service qualities. We have described membership protocols for three channel types: totally ordering, reliable FIFO and unreliable. In particular, this paper contributes an efficient dynamic membership protocol for unreliable real-time channels. Such channels can effectively and efficiently support adaptive multimedia applications where applications drop and re-join media streams according to the capacity of their connection.

A CCTL prototype has been implemented using UDP to connect to remote sites over wide area networks and using efficient IP-multicast on local networks. Our prototype currently runs under the Solaris and Irix operating systems on Sparc and SGI processors. We have implemented several CSCW tools using CCTL and are in the process of developing a comprehensive collaborative environment using CCTL exclusively for communication. Our efforts will continue in several directions. First, we intend to further refine and improve the efficiency of our prototype. Second, we wish to precisely measure the performance of CCTL for a variety of real-life work loads and compare our performance with that of related systems. Third, we intend to continue developing specific CSCW applications using CCTL. We aim to confirm that the service guarantees offered by CCTL are flexible and efficient and well-suited to the needs of collaborative applications. Finally, we wish to extend our membership protocols to support partitioning and remerging.

References

[1] H. Abdel-Wahab and M. Feit, "XTV: A framework for sharing X window clients in remote synchronous collaboration", *IEEE Conference on Communication Software* , pp. 159–167, Chapel Hill, NC, 1991.

[2] Y. Amir, D. Dolev, S. Kramer, and D. Malki, "Membership algorithms in broadcast domains", *6th Int. Workshop on Distributed Algorithms*, Lecture Notes in Computer Science #674, pp. 292–312, Nov. 1992.

[3] K. Birman and T. Joseph, "Reliable communication in the presence of failures," *ACM Trans. on Computer Systems*, vol. 5, pp. 47–76, Feb. 1987.

[4] Ken Birman, Roy Friedman, Mark Hayden, and Injong Rhee, "Middleware Support for Distributed Multimedia and Collaborative Computing," *the Proceedings of the IS&T/SPIE Symposium on Multimedia Computing and Networking (MMCN'98)*, San Jose, CA, Jan. 1998, pp. 2 – 13.

[5] S. Y. Cheung, M. H. Ammar, and X. Li, "On the use of destination set grouping to improve fairness in multicast video distribution," *15th Int. Conference on Computer Communication (Infocom'96)*, IEEE, IEEE Communications Society, March 1996.

[6] G. Chockler, N. Huleihel, I. Keidar and D. Dolev, "Supporting multiple quality of service options with high performance groupware", in the *Proceedings of TINA'96*. Also Tech Report CS96-3, Institute of Computer Science, Hebrew University, March 1996.

[7] G. Chung, K. Jeffay and H. Abdel-Wahab, "Dynamic participation in computer-based conferencing system", *Journal of Computer Communications*, vol. 17, no. 1, pp. 7–16, Jan. 1994.

[8] D. Dolev and D. Malki, "The Transis approach to high availability cluster communication",*Comm. of the ACM*, vol. 39, no. 9, pp. 64–70, April 1996.

[9] The Ensemble Home Page: http://www.cs.cornell.edu/Info/Projects/Ensemble/.

[10] R. Friedman and R. van Renesse, "Strong and weak virtual synchrony in Horus", in *15th IEEE Symposium on Reliable Distributed Systems*, October, 1996.

[11] V. Jacobson, "Multimedia conferencing on the Internet", Tutorial 4, *ACM SIGCOMM*, Aug. 1994.

[12] O. Jones, "Multidisplay software in X: A survey of Architectures," The X Resource, O'Reilly & Associates, Inc.,pp. 97-113, vol. 6, 1993.

[13] Claus Gittinger, `http://tecfa.unige.ch/pub/software/unix/xbench/`.

[14] B. Glade, K. Birman, R. Cooper, and R. van Renesse, "Light-weight process groups in the ISIS system", *Distributed System Engineering*, vol. 1, pp. 29–36, 1993.

[15] K. Lantz, "An experiment in integrated multimedia conferencing", *CSCW 86*, pp. 267–275. Reprinted in I. Greif(editor), *Computer-Supported Cooperative Work: A book of Readings*, pp. 533–552, Morgan Kaufmann Publishers, 1988.

[16] S. McCanne and V. Jacobson, "Vic: A flexible framework for packet video", *ACM Multimedia*, pp. 511–522, Nov. 1995.

[17] S. McCanne, M. Verrerli, and V. Jacobson, "Low-complexity video coding for receiver-driven layered multicast," *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 6, pp. 983-1001, August 1997

[18] L. Moser, P. Melliar-Smith, D. Agrawal, and C. Lingley-Papadopoulos, "Totem: A fault-tolerant multicast group communication system," *Comm. of the ACM*, vol. 39, no. 9, pp. 64–70, April 1996.

[19] L. Moser, Y. Amir, P. Melliar-Smith and D. Agarwal, "Extended virtual synchrony", *14th IEEE International Conference on Distributed Computing Systems*, pp. 56 – 65, Poznan, Poland, June 1994.

[20] I. Rhee, C. Brueker, A. Krantz and V. S. Sunderam, "Supporting input multiplexing in a Heterogeneous Environment," *Proceedings of Internal Conference of Information Sciences,* pp. 97 – 100, Raleigh, NC, March 1997.

[21] I. Rhee, S. Cheung, P. Hutto and V. S. Sunderam, "Group communication support for distributed multimedia and CSCW systems", Tech Report 96-10/7, Math/CS, Emory University, Atlanta.

[22] I. Rhee, R. Rammohan, S. Cheung and V. S. Sunderam, "Achieving fairness and scalability in multicast video distribution using receiver-only rate control", Tech Report 96-8/7, Math/CS, Emory University, Atlanta.

[23] L. Rodrigues, K. Guo, A. Sargento, R. van Renesse, B. Glade, P. Verissimo, K. Birman, "A transparent light-weight group service", in the proceedings of *15th IEEE Symposium on Reliable Distributed Systems*, Oct., 1996.

[24] A. Schiper and A. Ricciardi, "Virtually-synchronous communication based on a weak failure suspector", *23rd Int. Symposium on Fault-Tolerant Computing*, pp. 534–543, Toulouse, France, June 1993.

[25] A. Schiper and A. Sandoz, "Uniform reliable multicast in a virtually synchronous environment", *13th Int. Conference on Distributed Computing Systems*, pp. 561–568, May 1993.

[26] V. S. Sunderam, "Collaborative computing frameworks for natural science research," CCF overview report, Emory University, http://ccf.mathcs.emory.edu/ccf/overview.ps.

[27] R. van Renesse, K. Birman, and S. Maffeis, "Horus: A flexible group communication system," *Comm. of the ACM*, vol. 39, no. 9, pp. 64–70, April 1996.

[28] B. Whetten, *A reliable multicast protocol.* in *Theory and Practice in Distributed Systems*, K.P. Birman, F. Mattern and A. Schiper, Eds., Springer-Verlag, *Lecture Notes on Computer Science 938*, July 1995.