Group Communication Support for Distributed Collaboration Systems*

Injong Rhee Shun Yan Cheung Phillip W. Hutto Vaidy S. Sunderam

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

Abstract

The Collaborative Computing Transport Layer (CCTL) is a communication substrate consisting of a suite of multiparty protocols, providing varying service qualities among process groups. CCTL explicitly supports distributed collaborative and multimedia applications. CCTL is based on a two-level group hierarchy. Logical interconnections among entities, called channels, define an efficient and light-weight group mechanism. Channels support a variety of service qualities such as reliability and message ordering. Related channels can also be combined to form sessions, *heavy-weight groups which provide a default atomic multicast service. CCTL supports membership protocols tailored to the quality of service offered by a channel, including a relaxed form of* virtual synchrony. *In this paper, we present three membership protocols and compare and relate our implementations to alternatives. Our two-level architecture allows simple and efficient implementation of the membership protocols.*

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 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 shared using special application-sharing programs such as multiplexors for the X Window system [1]. Computer-supported cooperative work (CSCW) systems combine these technologies to offer an integrated computing environment for conferencing and collaborative work.

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. For example, while audio/video conferencing tools may tolerate loss of data to maintain timely delivery, participant and control information must be transmitted reliably. As another example, many collaborative applications implement WYSIWIS interfaces ("what you see is what I see") which present all participants with a consistent view of the shared artifact (data or application). Simple reliable communication usually suffices for these applications yet occasional atomic communication for synchronization greatly simplifies implementation.

CSCW systems present the user with a rich media mix including audio and video. Such systems must support real-time multimedia transmission as well as collaboration and sharing. An adaptive video conferencing tool [18, 13] is an example of a demanding CSCW application. In a fair and scalable transmission scheme, image quality can be modulated to match receiver capacity. Capacity depends on processor speed, load and network bandwidth. One approach splits 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 [5, 18]. 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 service quality and membership, and joining and leaving channels must be efficient.

Application-sharing systems also demonstrate a need for data streams with independent service quality and membership. One approach to application-sharing under X Windows interposes 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" [12, 16] replicates the client and employs a simplified pseudo-server. Output operations

*Supported in part by NSF grant ASC-9527186.

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. Participants 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.

Scalability is also important for group communication systems [11]. Group communication systems can be used to implemented distributed systems composed of interacting objects, resulting in a unification of naming and object invocation. Each distributed object is associated with a unique multicast group and the multicast group name is used to reference the object. This approach gives rise to a potentially large number of multicast groups. Method invocation involves joining the associated multicast group and requesting operations. The combined paradigm can be used to implement shared files systems (data spaces) common in collaborative environments [22]. Clearly, such systems must support a large number of groups and efficient group membership operations.

In summary, (1) CSCW applications require access to a potentially large number of data streams; (2) data streams must support distinct service qualities; (3) membership is 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 collaboration systems. CCTL is the communication layer of the Collaborative Computing Framework (CCF) under development at Emory University. The CCF Project aims to develop an integrated framework to support collaboration in natural sciences, although our tools are not necessarily limited to that domain [22].

Next we survey related work. Section 3 introduces our system model and informally defines relaxed synchrony and the other guarantees that our system provides to applications. Section 4 describes the overall system architecture and details the three membership protocols. We also discuss failure handling in Section 4. In Section 5 we compare our work to related systems and sketch an implementation of CCTL using Horus. Section 6 discusses future work.

2 Related work

ISIS [3, 4] pioneered multicast group communication and introduced the notion of virtual synchrony. Virtual synchrony allows dynamic group membership while guaranteeing reliable message delivery. Failed processes appear to leave groups they are participating in. ISIS provides good support for reliable distributed systems, but lacks quality of service support needed by multimedia and CSCW applications.

RMP (Reliable Multicast Protocol) [24] provides various per-packet service qualities. Messages may be unordered, FIFO, causally or totally ordered. Unreliable delivery is also offered. 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 but this approach is inefficient. RMP provides no optimization for sharing resources (name service, fault-detection, etc.) among related groups. In addition, RMP provides only virtually synchronous atomic multicast. Applications requiring only reliable FIFO message ordering with virtual synchrony are forced to use the more expensive total ordering.

Transis and Totem [2, 14] extend virtual synchrony to tolerate network partitioning. The existing system offers limited service qualities although a recent proposal [6] 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. Video tools [5, 18] can be difficult to implement without independent membership.

Horus [23] 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. Two forms of virtual synchrony [9] are supported: *strong* and *weak*. Strong synchrony requires the sender and receiver of a message to have the same view (messages must be delivered in the same view in which they are sent). View changes must be delayed until all previous messages have been delivered in their views. Weak synchrony lessens this delay by allowing messages to be delivered in a subsequent view as long as the receiver sees a *suggested* view including the new membership. However, delivering consistent suggested views still involves some overhead and complexity. 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. In addition, Horus membership protocols require an underlying reliable transport making it very difficult to implement unreliable channels with reliable membership.

A light-weight group service for Horus has recently been implemented and described [19]. Horus light-weight groups (LWGs) may be mapped dynamically to heavy-weight groups (HWGs). However, LWGs are essentially multiplexed using a single HWG. Messages sent to a particular LWG are actually delivered to all members of the HWG (and filtered out by non-members of the LWG). LWGs effectively do not have independent membership and members of the HWG must process every message sent to any LWG. While LWGs improve performance by sharing resources, we believe CCTL's approach of statically relating LWGs and HWGs (channels and sessions) will result in additional efficiencies (e.g. an efficient name service).

3 Model and definitions

Our informal system model follows. See [17] 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 protocols

In this section, we describe our hierarchical architecture and present membership protocols.

4.1 Hierarchical group 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. Channels are light-weight groups supporting a variety of service qualities. Related channels combine to form a heavy-weight group called a session. Sessions also provide an atomic virtually synchronous service called the default channel. Sessions and channels support the same fundamental operations (join, leave, send and receive) but many channel operations can be implemented efficiently using the *default session channel*. Session members may join and leave channels dynamically but the quality of service for a

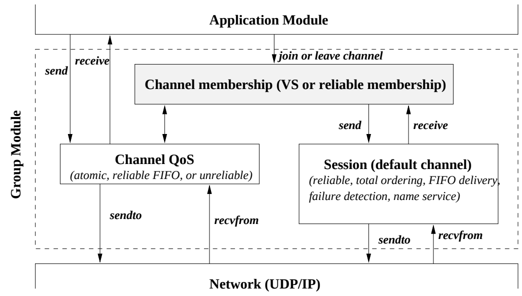

Figure 1: CCTL Architecture

particular channel is fixed at creation. Channels and sessions are destroyed when the last participant leaves.

Figure 1 shows the CCTL architecture. The group module consists of *channel membership* and *QoS* sub-modules. The channel membership module enforces ordering and reliability guarantees for membership change messages (join, leave). The QoS module enforces ordering and reliability guarantees for regular messages. The QoS module also provides an interface to lower-level network protocols such as IP multicast or UDP and handles inter-network routing (IP-multicast to a LAN, UDP over WANs).

Several researchers have proposed communication systems supporting light-weight groups [19]. These systems support the dynamic mapping of many light-weight groups to a small set of heavy-weight groups. CCTL statically binds light-weight channels to a single heavy-weight session, mirroring the semantics of CSCW environments. (Session participants initiate multiple tools that communicate using channels.)

We implement channel membership operations using the default session channel. Session participants initiate a channel view change by multicasting a view change request (join or leave) on the default channel. The channel membership sub-module monitors the default channel and maintains a *channel table* containing name, QoS, and membership for all channels in the session. All session participants have consistent channel tables because view change requests are totally ordered by the default channel. This technique simplifies the implementation of channel membership operations considerably. For instance, the message ordering imposed by the default channel can be used for ordering view changes. Furthermore, implementation of channel name services is simple, requiring a single lookup in the channel table. Our architecture logically separates channel control transmission (using the default channel) and regular data transmission. This separation increases flexiblity by decoupling data service quality from membership semantics.

Our hierarchical architecture is also scalable. Glade et al. [11] argue that the presence of recovery and name services and failure detectors degrades overall system performance as the number of groups increases. Typical failure detectors periodically poll group members (Horus uses one multicast message every two seconds). Per-channel failure detection introduces significant overhead. CCTL performs failure detection on a per-session basis resulting in significant savings. When a failed process is detected a unified recovery procedure removes the process from all channels simultaneously, reducing message overhead.

4.2 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. Requiring two processes to receive the same messages in the same view is meaningless if messages may be lost. It suffices to guarantee that a membership change is eventually recognized by every channel member. Unreliable channels in CCTL provide this minimal guarantee on the membership operations. CCTL also ensures a total ordering of membership view changes to prevent processes from leaving a channel without joining.

The reliable membership protocol is simple using the default session channel. A process p wishing to join an unreliable channel sends a view change request (VCR) message to the default channel. Each process receiving the VCR message delivers a join message to the application and adds p to its view. The leave operation removes p from the view.

Failure notifications in an unreliable channel require no message transmission (beyond the original notification). When process p receives a fail message for process q, p removes q from any unreliable groups for which it is caching membership and delivers the fail message to the application. Because failure messages are VCRs (hence ordered) all channel members go through the same view sequence.

4.3 Membership for atomic channels

We now sketch a membership protocol for atomic channels. We assume an atomic multicast primitive, omitting its description because efficient protocols are well known [4, 24]. Process p multicasts a join request (VCR_p) on the default channel. 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 channel being joined. This acknowledgement is totally ordered relative to regular messages.

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.

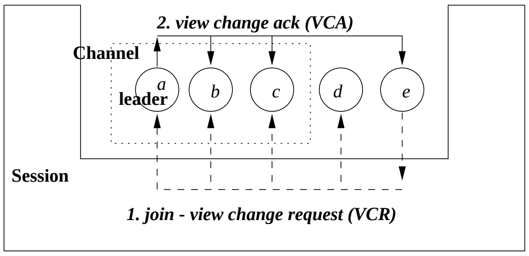

Figure 2: Dotted arrows - atomic session multicast; solid arrows - atomic channel multicast.

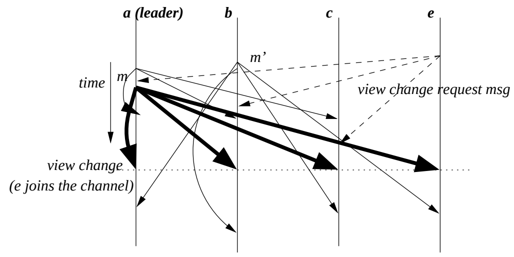

Figure 3: Regular messages m and m' ordered with respect to a view change message (thick arrows).

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

When process p fails, all session members eventually receive $fail_p$ from the unspecified failure detector. 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, members must decide which messages from p are to be delivered. We use a *flush* procedure, similar to that used in ISIS [4]. 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 determined. 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 [4, 19, 9] flush at *every* view change. Our protocol flushes only on receipt of *fail* because relaxed synchrony allows messages to be delivered in any view. The actual receiving view of a message is determined by its order relative to VCA messages. 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 or delivery during normal view changes (those not initiated by *fail*). Membership operations require at most two atomic multicasts. See [17] for a detailed description of the protocol.

4.4 Membership for reliable channels

View changes and regular messages must be explicitly ordered on reliable channels since atomic multicast is not available. To join or leave p sends VCR_p to the default session channel. All channel members respond by sending $VCA(p)$ on the reliable channel. On receipt of $VCA(p)_r$ processes delay delivery of subsequent messages from r until p's view change request is acknowledged by all non-faulty channel members. When all channel members acknowledge, a process adds p to its view, delivers join and processes delayed messages. Fail messages initiate a flush as in the atomic channel protocol.

All session members handle VCRs (including fail messages) in the order received from the default (atomic) session channel. Thus all channel members see the same view sequence. Since the channel implements a reliable (FIFO) multicast, when process q receives a VCA_r message from the channel, it implies that any subsequent message from r must be sent after the VCA. By the protocol, all the messages sent to the channel after any $VCA(p)$ are delivered after the view change message is delivered. If p delivers m after a view change, all channel members deliver m after the same view change. Hence, all receiving views of message m are identical. See [17] for details.

View changes require $n+1$ multicasts (n is the number of channel members). Optimizations are possible by having the leader collect and send sequence numbers of the last message each channel member sent. Channel members delay delivery of view change messages until all messages with earlier sequence numbers are received from all processes. Once all earlier messages are received, processes deliver the join or leave and continue processing delayed messages.

Assume each process q maintains a message sequence number seq_q. On receipt of VCR_p each channel member q sends seq_q to the channel. Processes delay delivery of messages sent by q after seq_q. The channel leader collects and retransmits all sequence numbers (in a single reliable message). Each process delays message delivery until all messages with earlier sequence numbers are received from all processes.

As usual, process r initiates a flush on receipt of $fail_q$. If $fail_q$ arrives before the leader receives seq_q, seq_q is not needed. No process will receive messages from q after the flush. (Either q is declared faulty before it sends seq_q or seq_q is lost.) If $fail_q$ arrives after the leader receives seq_q, r simply delivers the messages sent before seq_q by q. If messages from q earlier than seq_q are lost, r discards messages with sequence numbers larger than the first lost message.

The optimized protocol requires n point-to-point messages and 2 multicasts (one atomic and one reliable FIFO).

4.5 Handling failures

Rodrigues et al. [19] propose a novel technique to minimize flush messages: flush all channels simultaneously that failed process p belongs to. Using this optimization, $flush_p$ from q means all messages received by q from p on any channel are stable. Below we discuss further optimizations of the failure protocols for atomic and reliable channels (unreliable channels require no failure recovery).

For atomic channels, the following technique eliminates acknowledgement messages entirely. On receipt of $fail_q$, p includes the sequence number of the fail message in all subsequent messages to the channel. On receipt of such a message, other processes remove q from their view, deliver *fail* to the channel and continue processing messages.

The optimization for reliable channels is more complex. Let seq_p^C be the set of sequence numbers of the last message sent by p on any channel in C (channels containing both p and q as members). On receipt of $fail_q$, p includes the sequence number of the fail message in all subsequent messages to the channel as before. In addition, p sends seq_p^C to the leader. The *session leader* (the most senior session member) collects and retransmits these messages to the default channel. Delivery of marked messages (those with fail sequence numbers) is delayed until the sequence set is received from the leader. On receipt of the sequence set, processes deliver delayed messages preceding seq_s^C for all C and s. Finally, each process delivers fail messages for q to C and resumes message processing.

The optimized protocol requires no additional message retransmission and has the same message complexity as the fault-free protocol.

5 Comparison with other systems

In this section, we compare CCTL to Horus [23, 9], Transis [2, 6] and RMP [24] and show how to build CCTL using a Horus protocol stack.

5.1 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 [9]. 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.

5.2 Multiple service options in Transis

Transis has been elegantly extended [6] to support *quality of service multicast channels* (QMC). Figure 4 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. Reliable channels use the standard Horus reliable multicast stack. This architecture allows applications to synchronize across channels.

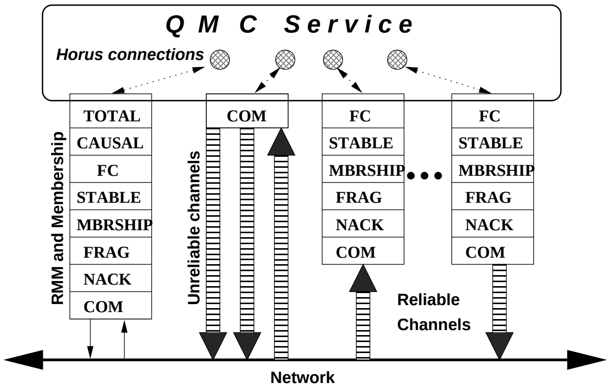

Figure 4: QMC in Horus

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.

5.3 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.

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 [24], 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.

5.4 Implementing CCTL using Horus

CCTL can also be implemented using Horus (Figure 5). 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.

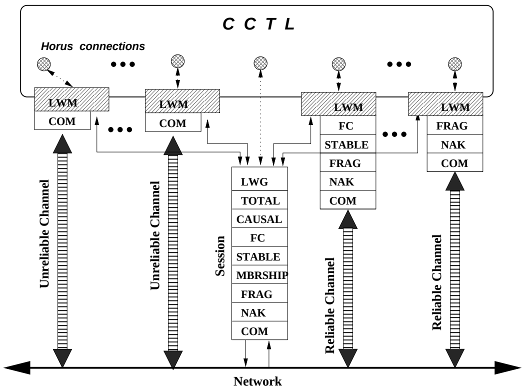

Figure 5: CCTL in Horus

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.

6 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 three membership protocols: atomic, 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] K. Birman, A. Schiper and P. Stephenson, "Lightweight causal and atomic group multicast", *ACM Trans. on Computer Systems*, vol. 9, no. 3, pp. 272–314, Aug. 1991.

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

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.