Individual and Group QoS Issues in Communication Services for Groupware Systems

Radu Litiu and Atul Prakash
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122 USA
Email: radu,aprakash@eecs.umich.edu

Abstract

The proliferation of computer networks in the last decade and the ubiquity of the World Wide Web have led to increased interest in the development of computer-supported cooperative work (CSCW) systems. Collaborative, multi-user applications require group multicast services that provide ordering guarantees for maintaining consistency of replicated shared context as well as provide a high degree of interactivity, even under varying load on the multicast servers. While the most common view of the quality of service (QoS) in a distributed system is in terms of the guarantee of the network connection parameters (bandwidth, end-to-end delay), in this paper we investigate QoS from the perspective of the various requirements placed on group communication servers with limited resources by multiple and diverse groups of collaborative users. We show that in the absence of QoS considerations in the design of a group communication service, some groups or individual users can be severely affected by bursty traffic or increase in the size of other groups. We present the design of a best-effort QoS-based adaptive group communication service for supporting reliable data communication in CSCW systems. Our QoS considerations address both group's requirements and individual user's requirements. We present performance results showing the effectiveness of the approach and discuss some of the open issues for future work.

1 Introduction

The proliferation of personal computers and workstations in the last decade has fueled the growth of computer networks and an extensive development of distributed applications. Though traditional systems for distributed computing such as distributed operating and database systems strive to provide the illusion of working alone in a networked environment, computer-supported collaborative systems aim to empower geographically dispersed users to effectively *share data* and *work together* over distance. Thus the sharing of data is made apparent in collaborative systems, and the mechanics of data sharing often dictates the overall effectiveness of collaboration.

The management of shared data and the necessity to provide high-quality group communication in large-scale collaborative systems places unique requirements on group multicast services. For example, the application responsiveness takes on much more importance in a collaborative system designed to provide a highly interactive collaboration environment. In other cases, such as distributed multimedia systems, parameters such as throughput, frame rate, resolution are emphasized. Different groupware applications and even clients within the same group place a different load on the communication system and have different demands with respect to the quality of service they receive.

This paper presents our approach to configuring multicast servers to support efficiently the wide range of user needs in different types of collaborative applications. Our system provides QoS support based on priorities and explicit control over the scheduling of different activities and by dynamic adjustment of its policies according to system load, user input, application requirements and current global configuration. We have incorporated the approach in our Java-based Corona multicast server, which is being used to support both synchronous and asynchronous collaboration over the World Wide Web, where collaborating clients may be dynamically downloaded over the Internet. The focus in this paper is in providing better quality of service for synchronous collaboration, where QoS requirements are more stringent. We show that servers can be made more responsive to user/group requirements, even in best-effort systems.

The rest of the paper is organized as follows. Section 2 motivates our work in providing QoS support for computer-supported collaboration and discusses the key requirements of the design of a QoS-based large scale group communication system. Section 3 discusses related work. Section 4 details the solution we propose to address the QoS issues in synchronous collaboration. Section 5 reports performance results that compare our approach with a non-QoS based approach. Section 6 concludes the paper with a brief summary of our work and our future plans.

2 Background

Our work on the quality of service in computer-supported synchronous collaboration has its origin in a NSF-sponsored project, called the Upper Atmospheric Research Collaboratory (UARC) [6]. The UARC project focuses on developing an experimental testbed for wide-area scientific collaboratory work. This testbed is implemented as a large object-oriented distributed system on the Internet and provides a collaboratory environment in which a geographically dispersed community of scientists perform real-time experiments at remote facilities without having to leave their home institutions. This community of scientists has extensively used our system over the last few years.

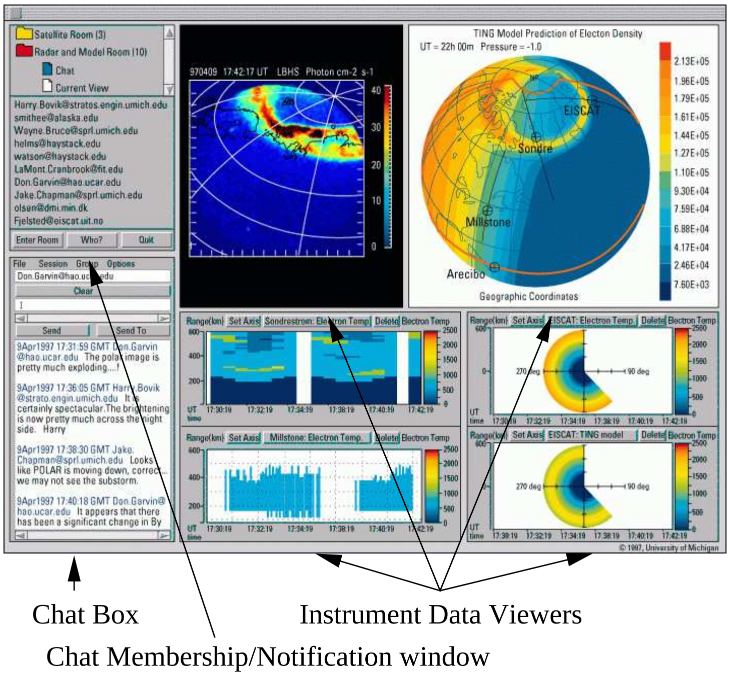

Figure 1: Collaboration tools using the Corona communication services

The implementation of the system has evolved through several generations of prototypes. The current design is an applet-based architecture implemented in Java, taking advantage of the accessibility and ubiquity of the World

Wide Web and Java platform-independence. A server, called Corona ([8], [17]), provides multicast services to support various collaboration tools. These include various shared data viewers for graphically displaying instrument data, a multi-party chat box for exchanging textual messages, and a notebook-like draw tool for saving and sharing notes, images and drawings. An audio-conferencing tool, which enables participants to exchange live audio data is currently under development. The server is currently written in Java, to allow scientists quick prototyping and experimentation of the server on various platforms. Figure 1 shows the graphical interfaces of some of the Corona-based collaboration tools.

2.1 Overview of Corona

Group Communication: The basic unit of communication in Corona is the *group* [1]. A group is defined to be a set of processes, termed *members*. A group has a *shared state* and the members of the group operate on the shared state by accessing and modifying the shared objects in the shared state. Corona requires a process to be a member of a group in order to operate on the shared state of the group. A client application can be simultaneously a member of several groups. The group members communicate with each other by exchanging messages among themselves. The actions taken by members in a group are synchronized, resulting in the processes having a consistent view of the shared state of the group.

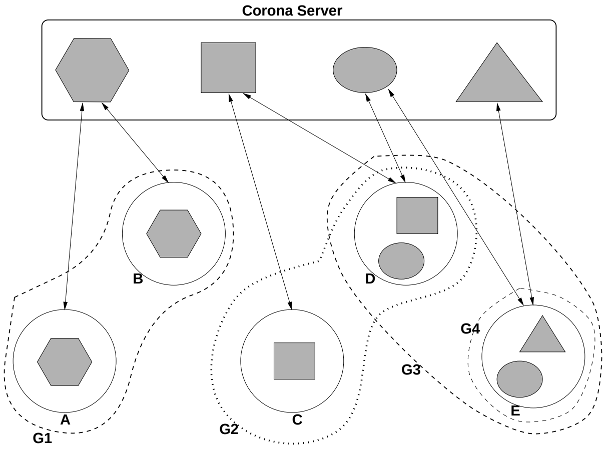

Figure 2: Architectural Overview of Corona. Circles represent clients, dotted lines depict groups, and different shapes represent different shared states. Note that clients may belong to different groups; Client D belongs to both Group G2 and Group G3, and Client E belongs to Group G3 and Group G4. Group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

Corona Architecture: The major component of Corona is a server that provides group multicast services and manages groups and their shared states. When a process joins a group, the server transfers a copy of the current shared state of the group. In order to ensure fast and reliable state transfer, even in the presence of client failures, the Corona server maintains a log of the shared state, including updates. Figure 2 illustrates the Corona architecture. The Corona server manages groups and their shared states.

A key assumption in our design is that clients are *unreliable* but the server is be *reliable*. Furthermore, in order to support different-time collaboration, we need to support *persistence* of group state, even when all members leave. Thus, we decided to use a server-based multicast service, with the server logging all the updates.

The Corona version currently used in UARC is a centralized server, to simplify dealing with Java applet security restrictions in browsers — applets are only allowed to make network connections to machines from which they were downloaded. We have also prototyped distributed versions of the server. The approach discussed in this paper is applicable to both versions of Corona.

Group Services: The Corona server provides a suite of services which may be categorized as: *group membership*, *group multicast*, *synchronization*, and *checkpointing*. The group membership service provides support for creating, deleting, joining, and leaving groups. A client joins a group and receives the shared state of the group or leaves a group unobtrusively; the existing processes in the group are able to carry on with their operations in the presence of multiple, concurrent joins and leaves. The Corona server allows clients to specify *roles* [7] when joining a group.

The group multicast service provides interfaces for broadcasting updates on shared state. Messages from members of a group are multicast as point-to-point messages in the order of arrival to all the members, thus ensuring total order.

The synchronization service provides interfaces for synchronizing client updates through locks, and the checkpointing service allows a client to take a snapshot of a shared state. A group of clients may subscribe to any combination of services and specify how a particular service is provided depending on the collaboration semantics.

2.2 Individual and Group QoS Design Requirements

Multi-group, Web-based CSCW systems, such as UARC, place interesting QoS demands on a multicast service such as Corona. Some of the key characteristics of the demands are:

Service based on Client-Roles: Scientists who are participating in analysis of evolving scientific data tend to be more impatient than casual observers with any delays caused by Corona, network congestion, or load imposed by other users of Corona.

Service based on User Activity Level: A better quality of service needs to be provided to client applets that are being actively used by a user. For example, viewer applets that are iconified or hidden on the desktop can usually tolerate longer latencies than applets that are actively being observed by users.

Unpredictable workload: For data sharing applications, the bandwidth requirements are often bursty. For instance, a shared whiteboard often has low bandwidth requirements but, when images are loaded into the whiteboard by a user, it can require much higher bandwidth in order to multicast the image to all the participants. Also, in a long-duration collaboration, there can be long periods of inactivity followed by bursts of activity, e.g., depending on the interest in the evolving scientific data. So, *apriori* reservation of resources at the servers ([10]) can be difficult for the clients to do or can be inefficient.

Service based on group characteristics: Some multicast groups can be more important than others. For instance, in UARC, it has been found desirable to ensure that Chat traffic, which is low bandwidth, gets through in a reasonable time irrespective of the load imposed by other groups on the server. Also, multicasting of real-time scientific data may be more important to users than other data, such as archived data.

Simultaneous Support for Small and Large Groups: In UARC, we allow dynamic creation of *rooms*, which are centers of collaboration. Users can participate in multiple rooms simultaneously. The usage of UARC indicates that some rooms can have a much larger number of participants than others. The Corona server needs to ensure that the quality of service to one group does not suffer significantly as a result of the growth in size of other groups.

Adaptability to Bursty Traffic: Low-bandwidth traffic in one group may need to be protected from bursty traffic in another group, even when groups are of equal priorities to users. For example, we have found it desirable in UARC to ensure that Chat does not suffer if a shared whiteboard is being used to transmit a high-resolution image. The servers should usually adapt to bursts of traffic in favor of low-bandwidth groups.

Avoid Starvation: Irrespective of initial priorities, some mechanism should exist to reduce the risk of starvation in the sending of messages to any member of a group, due to the activity of other members or the activity in other groups.

We can use QoS-based policies for scheduling multicasting tasks within the Corona server. Section 4 discusses our approach to incorporating the above requirements into Corona.

3 Related Work

There exists a great deal of interest in providing QoS guarantees of the network communication ([20], [9]). Robin et al. [16] address both the network and host QoS control problem in a system based on the Chorus [2] micro-kernel. Several distinct *policies* for admission control and dynamic quality control are outlined in [10], based on the experience with using Real-Time Mach [18], a micro-kernel architecture which supports the notion of *processor capacity reserve*. Mehra et al. [12] introduce the *real-time channel* as a paradigm for guaranteed-QoS communication services in packet-switched networks. The architecture proposed provides services such as admission control, traffic enforcement, buffer management, and CPU and link scheduling.

Rajan et al. [15] propose a formal framework for multimedia collaboration, which distinguishes three levels of abstraction: *streams* at the lowest level, for media communication, *sessions* at the next level, representing collections of semantically related media streams, and *conferences* as temporally related sequences of sessions. An overview of the QoS issues involved in distributed multimedia communication is presented by Vogel et al. [19] from the perspective of communication protocols, operating systems, multimedia databases, and file servers.

Nahrstedt and Smith [13] point out that in order to provide applications with end-to-end guarantees, network resource management alone is not sufficient and indicate a need to balance resources among the application, network, and operating system at the endpoints, and between endpoints and the network. They introduce the *QoS Broker* as an intermediary who performs services such as translation, admission and negotiation in order to properly configure the system to application needs. Chaterjee et al. [3] present two models to facilitate *adaptive* QoS-driven resource management in heterogeneous distributed systems and propose a *graceful degradation* of the application QoS under certain circumstances. Mathur et al. [11] address the QoS problem in group collaboration systems by means of a protocol composition approach.

Greenberg [7] uses *roles* as a distinction among categories of users, as well as among individual users within a group. Edwards [5] presents a specification language for Intermezzo, a collaborative framework which supports static and dynamic roles assigned to users. Roles and priorities are introduced in DCWPL [4], a programming language used to develop user-customizable groupware applications. Based on the similarity with face-to-face meetings, DCWPL assign to users roles of peer or moderator, with the moderator having higher priority.

4 Our Approach — QoS-based Adaptive Corona

An important characteristic of UARC-style, data-oriented, CSCW systems is that the bandwidth requirements are not known in advance. So, it is difficult to design services that provide any QoS guarantees, even if real-time platforms were to be used. On the other hand, users often know what data is more important to them at any time and the services must make the best effort to deliver the data that is more important first.

To deal with this characteristic of CSCW system usage in the design of the Corona server, we propose enhancing standard multicast services to include *priorities* assigned to multicast groups and to users within a group. These Priorities are dynamically adjusted in response to changing user requirements and workload on the system. Below, we discuss the details of the scheme, how these priorities are maintained and used within the server, and how the scheme addresses the QoS design requirements of the system pointed out in Section 2.2.

4.1 Group Priority

Different multicast groups (e.g., chat, data instrument, shared white-board, etc.) can have different importance to users. Corona server allows clients, when creating a group, to specify the group's priority. Usually, the groups corresponding to activities with higher degree of interactivity or requiring lower latencies should be assigned higher priority. Clients can change the group priority dynamically if circumstances change. (Restrictions can be added so that only authorized clients can change priorities; the issue of limiting the ability of clients to ask for highest priority for everything is beyond the scope of this paper.)

Even groups sending similar data may have different priorities, based on the importance of the activity carried on. E.g., consider two shared data viewers, one of which displays real-time data while the other one plays back data previously generated and saved on the hard disk. Users may want events occurring in real-time to be given higher priority at the server than archived events.

4.2 Client Roles and Client Priority

Users may have *roles* in a group, requiring better service for some members than others. As noted in [14], the assignment of roles has a social meaning, the ones who are making a more important contribution being assigned more important roles. In the Corona multicast service, we provide three roles for the users of our system: *principal*, *observer* and *membership-observer*, listed here in the decreasing order of their priority. Principals have update privilege on a shared state. Observers and membership-observers are casual members who may only view the updates on the shared state or the changes on the group membership, respectively. A member may dynamically change its role after it has joined a group. The same user can join a group as principal and another group as observer. Thus, the roles represent the liaison between users and groups.

The assumption is that there are few principals in a group and potentially a large number of observers, with principals being the important members of a group. The delays in multicasting to principals should be largely independent of the number of observers. Thus, the server attempts to give higher priority in multicasting to principals than to observers. It may even decide to disconnect the observers when the size of the group is too large and the server is overloaded by intense traffic.

4.3 Active vs. Passive Clients

Clients can dynamically tell the server whether they are *active* or *passive* recipients of the group multicasts. For example, if the application displaying the data is iconified, the application can tell the server that it is in passive mode — latencies are less critical. The client priority within a group is decreased when the client indicates that its activity mode is changing from active to passive within a group, and restored when the client changes back to active mode.

4.4 Adapting to traffic bursts

Groups that use less bandwidth and less computing time for the handling of data exchange are given higher priority. A group's priority is lowered when the group starts using more bandwidth, usually due to data bursts. This policy is similar to the multi-level feedback queue scheme used in CPU scheduling where threads using more CPU are lowered in priority. Our goal is to insulate low bandwidth traffic, usually implying interactive use, from high bandwidth traffic, usually implying batch transfer of shared information.

For each one of the entities mentioned above (group, user) we define a *default priority* and an *instant priority*. The default priority is assigned at creation time. The instant priority initially equals the default one, but can be lowered by the communication system, for example, when there is a burst of data in the group, or increased, if messages are queued up and haven't been sent for a while so that starvation risk is reduced.

createGroup(gName, gType, initState)	creates a group with name *gName* and type *gType* and initializes its state to *initState*
joinGroup(gName, role)	joins the group with name *gName* with role *role*
changeRole(gName, newRole)	changes client role in the group with name *gName* to the value *newRole*
changeActivityMode(gName, newActivityMode)	changes client activity mode to the value *newActivityMode* in the group with name *gName*. *newActivityMode* is either *active* or *passive*.
sendMessage(msg, gName, msgType)	sends the message *msg* of type *msgType* to the group *gName*. Totally ordered delivery within the specified group is guaranteed by the server.

Table 1: Corona client interface for specifying priorities. Additional interface exists for state transfer, membership change notifications, etc., and is not shown.

Table 1 presents some of the functions in the client API used in the assignment of priorities for groups and users.

4.5 Scheduling of Message Transmissions within the Server

The Corona server has been implemented as a multi-threaded Java application, supporting downloadable Java applet clients. The following threads are used by the Corona server:

- **Connection Manager:** accepts new connections from the clients. There is one thread in the server for accepting connection requests.
- **Client Receiver:** for every client, there is Client Receiver thread, which receives data sent by the client.
- **Client Sender:** for every client, there is a Client Sender thread. The thread maintains the outgoing message queue for the messages to be sent to a client and attempts to send the messages on the queue to the client.
- **Scheduler:** the thread operating at highest priority; it controls the order in which all the other threads are scheduled.

Figure 3 presents the interactions between different server threads and the data flow between these threads and between the server and the clients.

The behavior of the Java runtime scheduler with regard to scheduling of more than one thread running at the same priority is not defined. In some systems (e.g., NT) it uses round-robin time-slicing to give all such threads equal time, whereas in other systems (e.g., Solaris) it uses non-preemptive scheduling. Additionally, a thread of lower priority will never be scheduled as long as there exists a higher priority thread running, unless the higher

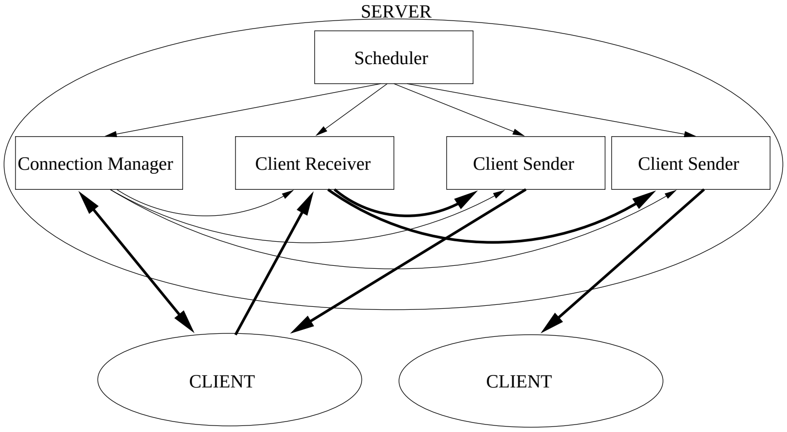

Figure 3: Client-server relationship. The thin lines represent interactions between threads. The thick lines represent the data flow.

priority thread yields explicitly. For this reason we have implemented our own *scheduler*, running at the highest priority, to control the order in which the threads execute.

When a client sends a multicast message to a group, on the server side the message is received by the corresponding Client Receiver thread. The thread inserts a reference to the message in the message queues corresponding to the receiving clients (Figure 4). A Client Sender maintains one message queue per group that the client is a member of. Since the queues contain references to the actual data, copying of the data is avoided. The messages in a sending queue are delivered to the client by the Client Sender thread. A Client Sender is inactive as long as there is no message for it to send and it is notified when a message is received.

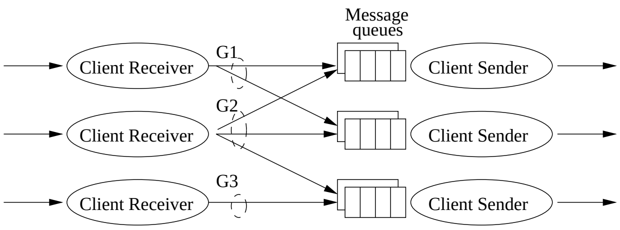

Figure 4: Group multicast. Messages are sent to groups G1, G2, G3. A Client Sender maintains a message queue for each group the client belongs to.

Each message queue is dynamically assigned a *priority*, based on the sum of the current priorities of the group and the receiver associated with the queue. Recall that there is one message queue per group per receiver.

The priority of message queues is mapped to thread priorities of Client Sender threads as follows. Client Sender threads serving messages in higher priority queues are scheduled before threads serving only low priority queues. Threads lower their priority when all of their high priority queues are served. Each thread uses a round-robin policy within the queues of the same priority to deliver messages to its client.

Message queue priorities can be dynamically changed, but all messages in a given queue have the same priority, in order to guarantee ordered delivery of messages in a group.

Client Receiver threads run at the maximum of the priority of groups in which the client has a principal role. Recall that a client can be a member of multiple groups and all messages from that client are received on the same connection. We need to run the receiver threads at the maximum of the priority of the groups to which the member can send messages, so as to avoid a client's membership in a low-priority group from reducing the priority with which the client's messages to a high-priority group are handled by the server.

4.6 Dynamic Adjustment of Group and User Priorities

To insulate low-bandwidth communication from bursty traffic in another group, the instant priority of the group where the bursty traffic occurred is lowered. This leads to a decrease in priority of the message queues corresponding to the group members, causing a decrease of the priority of the corresponding Client Sender threads when they are ready to deliver messages from those queues. The decrease of a group priority with the bandwidth usage in that group is illustrated in Figure 5. Priority is raised when the recent bandwidth usage goes down.

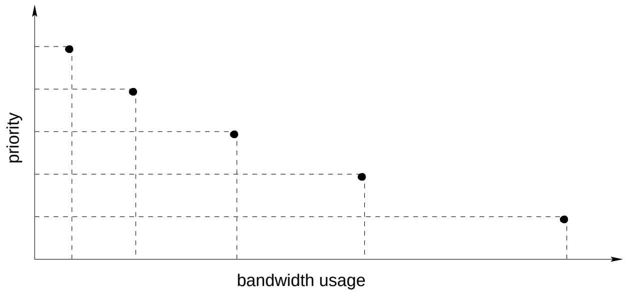

Figure 5: Adjustment of group priority with bandwidth usage

When a client application is still running, but is not active (the application window has been iconified or the window is covered by other windows, denoting a reduced interest from the user in front of the screen), the server is notified and the priority of the message queue corresponding to the client application is decreased, thus offering other potential clients a better quality of service.

To avoid starvation, the instant priority of a queue is temporarily raised above the default value when a queue is not served for a timeout interval. This case, which we call *client aging*, denotes an exceptional situation and should only happen under unusually heavy traffic. The timeout interval value is a matter of policy, but should be reasonably large to prevent low priority traffic from disrupting high priority communication. If timeouts are violated on a queue frequently, a policy of removing low-priority members from the system can be implemented.

The following algorithm outlines the dynamic changes in priority:

```
for each group G
    if traffic(G) > quota(G)
        decrease priority(G)
    endif
    if traffic(G) == back-to-normal
        priority(G) = default
    endif
end for
for each client C
    if stateChange(C)
        if change =  active-to-passive
            decrease priority(C)
        else   /* change = passive-to-active */
```

```
            increase priority(C)
        endif
    endif
    if starved(C)
        increase priority(C)
    endif
endfor
```

A change in the priority of a group will lead to a change in the priority of the queues for all the members of the group. A queue which has its priority raised due to starvation is brought back to its default priority after the sender thread gets a chance to send some data from the queue.

5 Performance Measurements

This section presents some preliminary results obtained by applying the scheduling policies outlined previously to the design of the Corona server. We have used in our tests a mix of Sun Sparc 20s and Ultra Sparc 140s on a LAN. The server runs as a stand-alone Java application on a Sun Sparc 20. In the subsequent experiments, all the clients in a group but one are receivers; the receivers connect to the server, join a group and receive the broadcast messages addressed to that group. One client is both a sender and a receiver and does the measurements. The latency is calculated based on the round-trip delay seen by the client that sends messages. From observation of real messages exchanged by our client applications, the typical size of a message generated by the chat or draw tools is in the order of a few hundreds of bytes. Therefore we have used messages of size 200 bytes (unless otherwise specified) in our experiments.

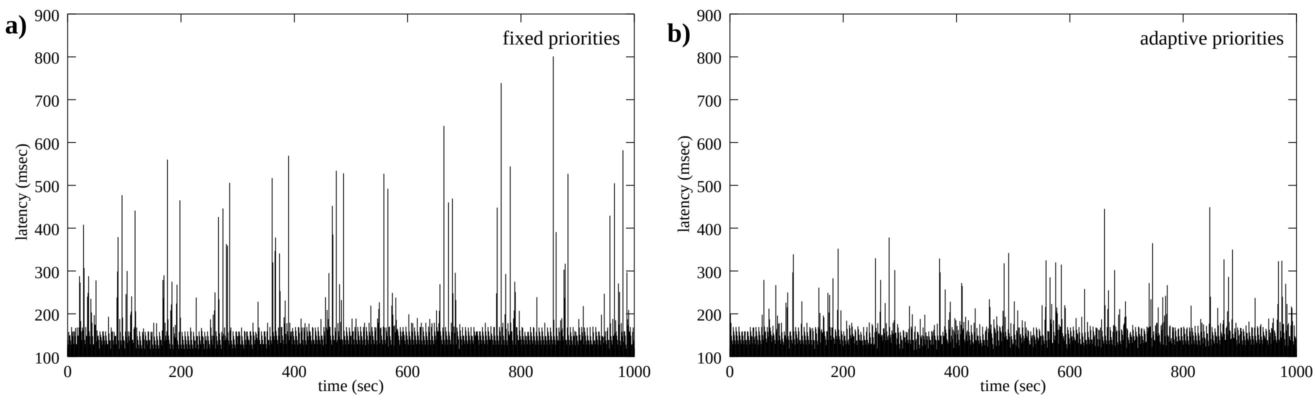

Figure 6: Influence of high bandwidth traffic in one group on the latency seen by clients in a a low bandwidth group. **a)** All clients have equal and constant priority. **b)** The priority of the high bandwidth group is decreased by using the adaptive strategy.

One of the goals of our QoS-based server is to monitor the usage of the system and to protect the users and groups against the over usage and occasional bursts of data from other groups. Figure 6 presents the latency experienced by a client in a group with the traffic kept within the accepted bounds in the presence of bursts of data in another group. Each group has 6 members. In the first group there is one client which broadcasts data continuously with the rate of 5 messages/sec. The burst is introduced by the second group by occasionally sending a sequence of 5 messages of size 40000 bytes. In **a)** a standard non-QoS based is used, with all Client Sender threads having the same priority, while in **b)**, a QoS-based server is used, with the priority of all the members of the group with the bursty traffic decreased until the burst disappears. As shown, in a QoS-based server, the impact on the low bandwidth group is substantially reduced and the predictability of the response time is increased. The impact is not completely eliminated because of the course granularity of scheduling in our user-level scheduler (because we have no control over Java's run-time scheduler) and the attempt by our scheduler to avoid starvation of the high-bandwidth group.

We also investigated the influence of the increase in size of one group on other groups. Figure 7 compares the latency seen by the clients in two groups; one of the groups has two members, while the size of the other one increases gradually up to 50. In one case, all the clients in both groups have equal and fixed priority, as in a non-QoS based server. In the QoS-based server's case, the smaller group is given higher priority than the larger group in which members are added. In each group there is one client which broadcasts data continuously with the rate of 5 messages/sec. The data displayed has been obtained by averaging over several measurements.

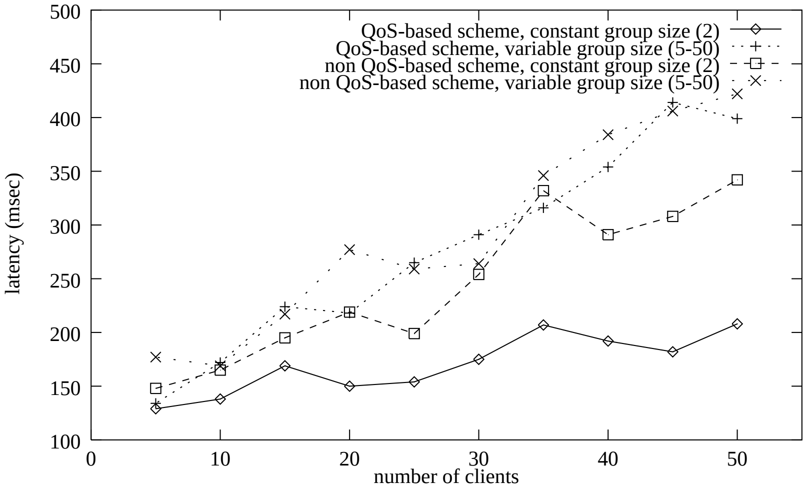

Figure 7: The latency increases with the size of the group. With the adaptive priority strategy, the members of the small group are protected against the increase in size of another group. Without a priority-based strategy, the latencies seen by the small group go up significantly as the size of the other group is increased. The latencies in the larger group are similar in both cases.

One observation from data in Figure 7 is that the latency increases approximately linearly with the size of the group and ultimately with the cumulated size of all the multicast groups. This is because latency is largely determined by scheduling of threads within the server in our environment. Also, in this and other experiments (results not shown for brevity), we have noticed that for messages of size up to a few hundreds of bytes the size makes little difference in round-trip times. The influence of the message size is more evident above 1000 bytes. The same observation applies for the time used by a client to send messages.

Another observation relates to the impact of the increase in the size of one group on the constant-size group. When the server is non-QoS based, the increase in the size of one group causes an almost identical increase in the latency seen by the clients in *both* groups. In the QoS-based server, the impact on the higher-priority smaller group is minimized as the size of the lower-priority group increases.

Figure 8 presents the latency seen by clients with different roles belonging to the same multicast group with 20 members, in an experiment that investigates the relationship between the responsiveness seen by different clients and their priorities within a group. One of the clients broadcasts data at constant rate (3 messages/second). The clients in **a)** and **b)** are equally split in two classes of priority, while in **c)** all the clients in the group have the same priority. There is a noticeable difference between the delay seen by a client with high priority and one with low priority. When all the clients have the same priority, the response is less predictable, denoted by the wider spread of the data points. The seemingly periodic appearance, more obvious for the graph in c), can be explained by an implementation detail. Since a multicast message is sent as multiple point-to-point messages and the order in which messages are sent to equal priority clients may make a difference, we chose to vary circularly the order of the clients in the group. In order to provide higher responsiveness for the client that sends the broadcast message, we can adjust the order in which the message is sent to the clients in the group by choosing the sender to be the first one.

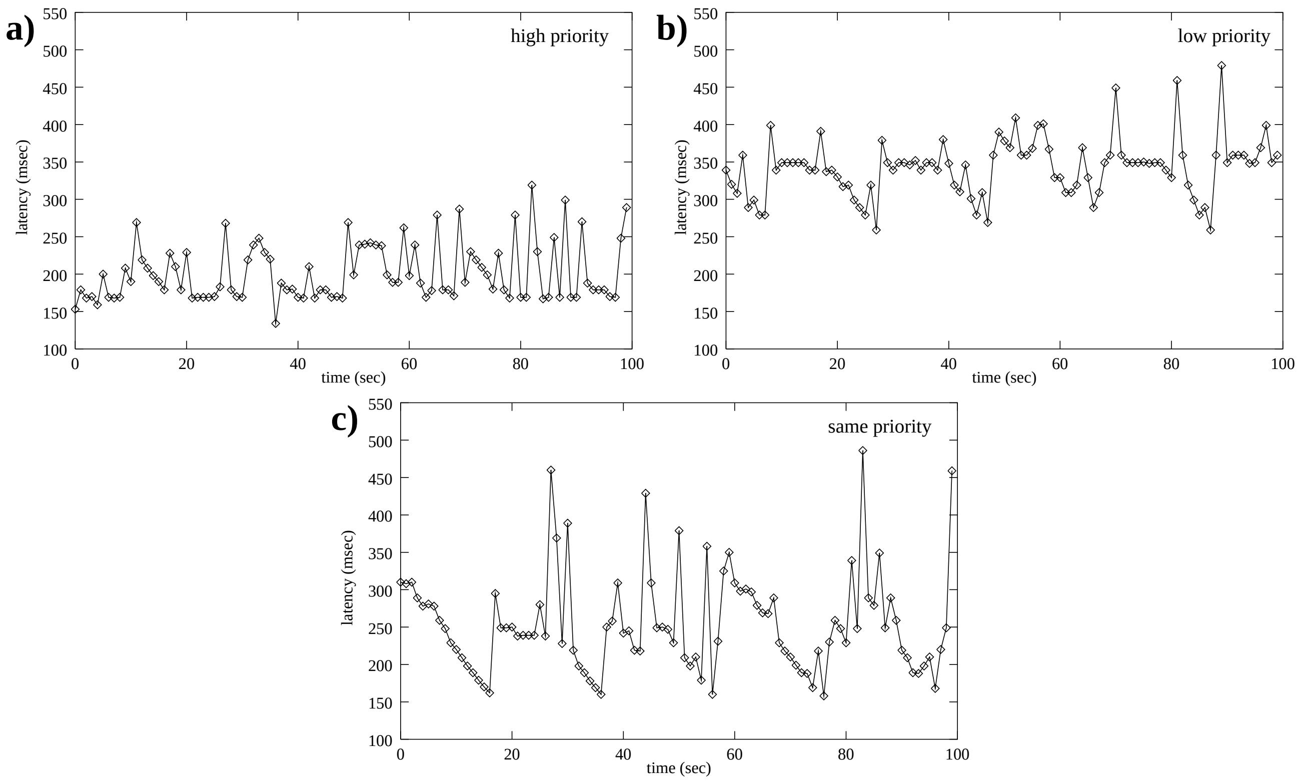

Figure 8: Impact of priority on the latency seen by a client. **a)**The client has high priority. **b)**The client has low priority. **c)**All the clients in the group have the same priority.

We tried to achieve an increased predictability of server response time. By dividing the clients in classes of priority, we have obtained a reduced range in which the latency varies, i.e., between 150-300 msec for high priority clients and between 250-400 msec for low priority clients, while in case **c)** the latency oscillates mainly in the range 150-400 msec.

The priorities are not assigned statically, since in this case a client with low priority could starve in the presence of continuous activity of the high priority clients. Figure 9 displays the adjustment of the priority of a low priority (4) client in a group with 20 other clients having high priority (5). One of the clients sends data continuously. As messages in the queue corresponding to the low priority client age, its priority is temporarily increased above the default value, and it is restored after the aged messages are delivered.

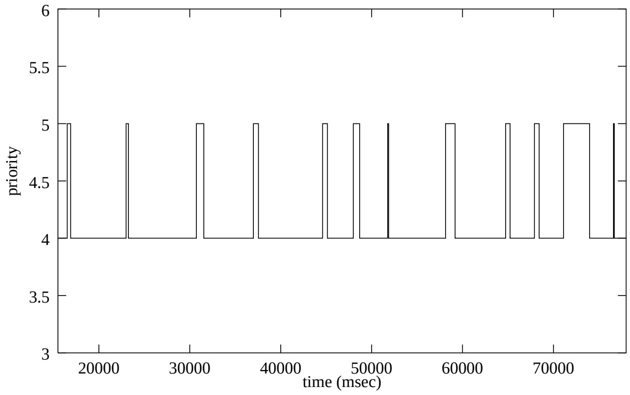

Figure 9: The priority of a low priority client is increased occasionally to prevent starvation

6 Conclusions and Future Work

In this paper we presented our approach to providing flexible group communication services for meeting end-user's QoS requirements in synchronous collaboration systems where workload is bursty or not known to users *apriori*. Large-scale collaboratories, such as the UARC collaboratory we support, appear to have these characteristics. We addressed the QoS issues from both the group and individual point of view. Our solution is based on allowing clients to specify and dynamically change group priorities as well as member priorities within a group. Member priorities are specified by providing role information and activity status information. The server uses the information provided by the clients to determine scheduling priorities of server threads that handle message communication with each client. The server uses an adaptive strategy to protect low-bandwidth communication in one group from bursty communication in other groups and to avoid starvation of low-priority clients.

In the absence of guarantees on the network bandwidth availability or the real-time response from the operating system in a heterogeneous network environment, our approach offers a best-effort service. A prototype of the server with QoS support has been implemented and the experimental results appear promising.

Our current research efforts are focused on increasing the scalability and robustness of the server and examining the issues involved in a distributed implementation of the server. One of the challenges of a distributed server implementation is to optimize the distribution of groups over multiple servers. The alternatives are either to use servers dedicated to different groups, thus eliminating the potential traffic among servers that maintain the shared state of a group, or to split each group among servers, taking advantage of the location of the users relatively to the servers and thus eliminating some of the network traffic due to the broadcast of a message to large groups. The QoS-based strategy discussed in this paper can be applied to each copy of the server. In addition, the QoS information can be useful in determining dynamically the best way to partition the functionality of the service among multiple servers.

Another direction we intend to pursue is to address the scalability problem by using reliable IP-multicast to distribute a message to large groups of users. At present, our server does not do that because most of our clients (which are Java applets running at sites not under our control) do not have IP-multicast support. Another reason is that maintaining group membership can become more difficult with IP-multicasts — IP-multicasts achieve some of their scalability by sacrificing availability of group membership change notifications.

An important problem in a QoS-based scheme is to ensure that clients do not attempt to ask for high priority for everything. If that happens, our server will still provide better service than a non-QoS based server because it will change priorities based on bandwidth usage. However, in general, it is important to have some mechanism to ensure that clients prioritize their needs reasonably. This is an important issue for all QoS-based schemes, and requires its own in-depth treatment. One strategy, for example, may be to authenticate clients and to set quotas on the number of groups in which a client can be a principal or the number of groups in which a client can be active at a time. Non-authenticated clients (e.g., casual participants over the Web) can be restricted to smaller quotas. We intend to investigate this issue further in the context of the UARC Collaboratory project.

7 Acknowledgments

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

References

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

[2] A. Bricker, M. Gien, M. Guillemont, J. Lipskis, D. Orr, and M Rozier. Architectural Issues in Microkernel-based Operating Systems: the CHORUS Experience. *Computer Communication*, 14(6):345–357, July 1991.

[3] S. Chatterjee, J. Sydir, B. Sabata, and T. Lawrence. Modeling Applications for Adaptive QoS-based Resource Management. In *Proceedings of the 2nd IEEE High-Assurance System Engineering Workshop (HASE97)*, Bethesda, Maryland, August 1997.

[4] M. Cortes and P. Mishra. DCWPL: A Programming Language for Describing Collaborative Work. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 21–29, Cambridge, Mass, November 1996.

[5] W. Keith Edwards. Policies and Roles in Collaborative applications. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 11–20, Cambridge, Mass, November 1996.

[6] C.R. Clauer et al. A Prototype Upper Atmospheric Research Collaboratory (UARC). *EOS, Trans. Amer. Geophys. Union*, 74, 1993.

[7] S. Greenberg. Personalizable Groupware: Accomodating individual roles and Group Differences. In *Proceedings of the European Conference on Computer-Supported Cooperative Work(ECSCW)*, Amsterdam, September 1991.

[8] R. W. Hall, A. Mathur, F. Jahanian, A. Prakash, and C. Rasmussen. Corona: A Communication Service for Scalable, Reliable Group Collaboration Systems. In *Proc. of the Sixth ACM Conference on Computer-Supported Cooperative Work*. ACM Press, Nov. 1996.

[9] J Kurose. Open Issues and Challenges in Providing Quality of Service Guarantees in High-Speed Networks. *ACM Computer Comm Review*, 23(1):6–15, January 1993.

[10] C. Lee, R. Rajkumar, and C. Mercer. Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach. In *Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems*, Hiroshima, Japan, June 1996.

[11] A.G. Mathur and A. Prakash. A Protocol Composition-Based Approach to QoS Control in Collaboration Systems. In *Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems*, Hiroshima, Japan, June 1996.

[12] A. Mehra, A. Indiresan, and Kang G. Shin. Structuring Communication Software for Quality-of-Service Guarantees. In *Proceedings of 17th Real-Time Systems Symposium*, December 1996.

[13] K. Nahrstedt and J.M. Smith. The QoS Broker. *IEEE Multimedia*, 2(1), Spring 1995.

[14] C.M. Newirth, D. S. Kaufer, R. Chandhok, and J. Morris. Issues in the Design of Computer Support for Co-authoring and Commenting. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 183–195, Los Angeles, CA, 1990.

[15] S. Rajan, P.V. Rangan, and H.M. Vin. A Formal Basis for Structured Multimedia Collaborations. In *Proceedings of the 2nd IEEE International Conference on Multimedia Computing and Systems*, Washington, D.C., May 1995.

[16] P. Robin, G. Coulson, A. Campbell, G. Blair, and M. Papathomas. Implementing a QoS Controlled ATM-Based Communication System in Chorus. Technical Report MPG-94-05, Dept. of Computing, Lancaster University, March 1994.

[17] H.S. Shim, R. W. Hall, A. Prakash, and F. Jahanian. Providing Flexible Services for Managing Shared State in Collaboration Systems. In *Proceedings of the Fifth European Conference on Computer Supported Cooperative Work*, pages 237–252, 1997.

[18] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *Proceedings of USENIX Mach Workshop*, October 1990.

[19] A. Vogel, B. Kerherve, G. von Bochman, and J. Gecsei. Distributed Multimedia and QoS: A survey. *IEEE Multimedia*, 2(2):10–19, Summer 1995.

[20] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Network*, September 1993.