Stateful Group Communication Services

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

Abstract

Reliable group multicasts provide a nice abstraction for communicating data reliably among group members and have been used for a variety of applications. In this paper we present Corona, a group communication service for building collaboration tools and reliable data dissemination services in Web-based environments, where clients connect independently of other clients and are not necessarily connected to the group multicast services all the time. The key features of Corona are: (1) the shared state of a group consists of a set of objects shared collectively among group members; (2) Corona supports multiple state transfer policies to accommodate clients with different needs and resources; (3) the communication service provides the current group state or state updates to new clients even when other clients are not available; (4) the service supports persistent groups that tolerate client failures and leaves. We show that the overhead incurred by the multicast service in managing each group's shared state has little impact on the latency seen by the clients or the server throughput. We also show that the multicast service does not have to be aware of the client-specific semantics of the objects in the group's state.

1. Introduction

Group communication services, such as ISIS, Transis, Psync, provide an excellent abstraction for building a variety of distributed applications, including fault-tolerant groups, groupware services, and reliable data dissemination services using the publish-subscribe model. In these systems, group communication services provide reliable delivery of data with various guarantees (e.g., FIFO, causal broadcast, atomic broadcast) as well as state-transfer services among group members so that consistency of state can be maintained among group members.

[0]This work is supported in part by the National Science Foundation under cooperative agreement IRI-9216848 and under grant ATM-9873025, and by an IBM research partnership award.

In this paper, we consider the following question: can we provide a general-purpose group communication service to clients in Internet-like environments where clients (e.g., data publishers, data subscribers, or group members) are not necessarily reliable and may frequently be disconnected from the service, but where reliable state and data transfer when clients connect with the service is a key requirement?

The management of shared data or *shared state* in such an environment places unique requirements that are not met by existing group multicast services. For example, the application responsiveness is much more important in a collaborative system designed to provide a highly interactive environment. Different collaborative applications may have different consistency/correctness requirements for replicated state. Another important class of services implemented using group communication, data dissemination services, manage large amounts of data of various types, obtained from different *publishers*. The delivery semantics ranges from a *push-based* scheme where the communication system distributes the data to the existing *subscribers* to a *pull-based* approach where clients connect occasionally and transfer in asynchronous mode data previously existing in the system. For the latter operating mode, the data dissemination service has to keep the data long time after it has received it from its publisher, since it would be expensive to retrieve the information every time a new client requests it.

Our work on the management of shared state and group communication has been done primarily in the context of computer-supported synchronous collaboration and it originated in a project focusing on developing an experimental testbed for wide-area scientific collaboratory work. Critical issues concerning the design of a collaborative system include:

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

- **Customized state transfer:** Client applications should be able to specify the way they receive the shared state (only the latest updates of the shared state, or only the state of certain objects within the group state) when joining a collaboration group.
- **Client-based semantics:** A general purpose group communication service should be provided. The interpretation of the semantics of shared data should be the responsibility of collaborating processes.
- **Persistence:** A group and its shared data should be able to outlive the process members of the group. A process joining a group after the group has assumed the null membership is transferred the persistent state of the group's shared data.
- **Group membership support:** In distributed systems knowledge of membership of a group is important to maintain consistency of replicated state and to ensure strict guarantees of message delivery and ordering. In a collaborative system, group membership takes on an important social aspect of *awareness* – users collaborating over shared state want to be aware of each other and their activities.

This paper presents our approach to providing a multicast service that meets the various shared state management needs of collaborative environments, and that can also be used in a much larger range of distributed applications (Figure 1). Our Corona stateful multicast service is designed to support both synchronous and asynchronous collaboration over the World Wide Web, where collaborating clients may be dynamically downloaded over the Internet.

The rest of the paper is organized as follows: Section 2 discusses related work. Section 3 details the architectural features of the Corona system, as well as the suite of shared state management and multicast services provided. Section 4 describes some of the issues which arise in a fault-tolerant replicated implementation of the communication service. Section 5 discusses the current implementation status, presents some performance results and briefly outlines our future plans. Finally, Section 6 presents some concluding remarks.

2. Related Work

As transport layer subsystems, ISIS [5, 4], and Transis [2, 3] support the notions of process groups, notification of membership changes, and group multicast and may be used

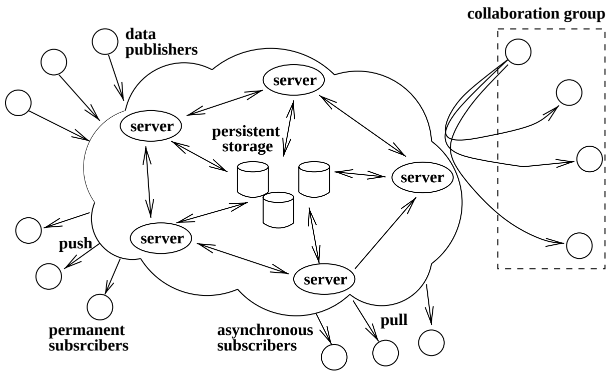

Figure 1. Stateful group communication services. A server pool maintains the shared information, both in memory and on stable storage. Publishers submit data to the communication servers, subscribers receive the data either in synchronous or asynchronous mode. Collaborative applications are provided services such as: shared state maintenance, incremental state update, persistence, group membership awareness.

to build services such as group awareness and group notification services. Both support a fully replicated architecture with individual members maintaining replicated state. In ISIS, the join of a new member involves the execution of a join protocol among all group members, and slow members can slow down the join operation. Furthermore, in ISIS any state associated with a group must be transferred to the joining client from an existing client, which may occasionally fail. Thus the time to complete the join reflects the time-out for failure detection and making an additional request to another client.

Transis provides a transport layer with a variety of multicast ordering and delivery semantics and it is used primarily in the context of distributed replicated database systems. By having replicated Transis processes, a higher level of fault-tolerance than a single server can achieve has been realized. One Transis-based approach [1] to achieve consistent replication suffers from the inefficiencies of using global total ordering with Lamport clocks. Corel [9] addresses this problem and also the fault-tolerance. However, these approaches require consistent membership views and require end-to-end acknowledgments for each message.

Horus [16] achieves consistent and fault-tolerant data replication by implementing the virtually synchronous process group, in which all processes receive consistent information about the group membership in the form of *views*.

Consul [13] implements the *replicated state machine* approach to provide a collection of fault-tolerant services for building distributed applications. It is based on the x-kernel [8] and Psync [12], a group-oriented atomic multicast protocol that preserves the partial/causal order of messages.

JavaSoft's JavaSpace is a RMI-based service for cooperative computing, implemented as an object repository (similar to Linda) that provides persistence, template-matching lookup, and transactions. The client-server protocol is seen as an exchange of objects, clients write tasks to a space as objects. One of the available servers retrieves and processes a request and writes back the object representing the result, which is read by the client that submitted the task. JavaSpace advocates simplicity and good scalability for cooperative, loosely-coupled systems, in which clients and servers don't have any knowledge about each other.

3. Group Communication Services in Corona

In this section, we discuss the architectural features of the Corona system and the group communication services provided.

3.1. Communication Groups and the Shared State Model

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

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

where i is a unique identifier of a shared object O_i, and S_i is a *byte stream* encoding of O_i. The state of a shared object is type-independent; a shared object should be able to write its internal state to a stream as well as to set its state to the data encoded in a stream upon request.

A *group* represents the basic unit of communication in Corona. A group consists of a set of processes, called *members*[1], that communicate with each other by exchanging messages and operate on the shared state by accessing and modifying the shared objects in the shared state. Only members of a group can operate on the shared state of the group. A group may be characterized as either *persistent* or *transient*. A persistent group and its shared state exist even when it has no members. A transient group ceases to exist when it has no members, and its shared state is lost.

The major component of Corona is a *stateful* logical[2] server that provides group multicast services. The server is stateful because it maintains up-to-date copies of shared states of various groups. From the server's point of view, the shared state of a group is a set of byte streams tagged with object identifiers, and thus the server cannot perform application-specific operations on its copy of the shared state. Instead, the group members update the server's copy through the server's group multicast services.

[1]member roles (*principal, observer*) are used to specify the relationships among members of a group

[2]the server may be replicated

3.2. Corona Services

Corona provides a suite of services for building groupware applications, such as: *group membership*, *group multicast*, and *state log reduction*. Corona also provides interfaces for synchronizing client updates through locks. A group of clients may subscribe to any combination of services and specify how a particular service is provided depending on the collaboration semantics.

The **group membership service** provides interfaces for creating, deleting, joining, leaving groups, etc. The Corona server works in conjunction with an external workspace session manager that determines which client is allowed to execute these actions. When creating a group, a client specifies the initial state of the group as defined in Section 3.1. A persistent group's state is kept by the service even if its membership has become null. The service deletes the group only in response to the *deleteGroup()* message, and the shared state of a deleted group is lost.

When a client joins a group, the service transfers a copy of the current shared state of the group to the new client. Based on the speed of its connection to the server and application characteristics, the client may request either to receive the whole state of the group or the latest n updates to the state (for incremental updates). It may also request to be transferred only the state of certain objects in the shared state of the group. The join protocol does not involve the existing members of a group, which are not aware of the fact that a new client is joining, unless they request explicitly membership change notifications. In this case, when the membership of a group changes, the service multicasts membership change notifications to group members. A member may also query the service for the membership information at any time by sending a *getMembership()* message.

The **group multicast and message logging service** provides interfaces for broadcasting updates on shared state. The service is stateful in the sense that all the multicast messages are logged both in memory and on stable storage, thus ensuring persistence of shared state and fault tolerance.

The service provides two forms of group multicast: *bcastState()* and *bcastUpdate()*. The information in a *bcastState()* contains a new state of the object specified, and the new state *overrides* the present state of the object. In contrast, the update information in a *bcastUpdate()* contains an *incremental* change to the state of the object, and the change is appended to the existing state, thus preserving the history of updates on a shared object.

A multicast message, M, is sent by a client either *sender-inclusively* or *sender-exclusively*. If M is sent sender-inclusively, the service multicasts M to all the members of a group to which the sender of M belongs, including the sender itself. If M is sent sender-exclusively, the service does not multicast M to the sender. A client multicasts a message sender-inclusively when the client needs certain operations that the service performs on the message (e.g., timestamping the message with real time).

State log reduction service: At the request of the communication service (several policies may be implemented based on factors such as the state log size and the type of the data) or, under certain circumstances, when desired by a client, the history of state updates for a group may be trimmed up to a point and replaced with the consistent group state existing at that point. The new state is equivalent with the initial state plus the history of state updates. For groups with incremental state update, log reduction may simply cause discarding the old state updates up to a certain sequence number.

4. Replication

One of the challenges of a distributed server implementation is to optimize the distribution of groups over multiple servers. One alternative is to use servers dedicated to different groups, thus eliminating the potential traffic among servers that maintain the shared state of particular groups. The broadcast messages within a group are sent only among the members of the group and the corresponding server. A server exchanges with other servers control and membership information, as well as queries the other servers for information that does not exist locally. This approach works well for groups of moderated size, especially when the clients are located in the proximity of the server, but in the case of large groups of users, the performance can be significantly degraded due to the high load on a server. On the other hand, if the users are widely distributed over different networks, bandwidth is wasted for sending the same data multiple times over the same network segments. The latter problem is eliminated if IP-multicast is used for communication between a server and its clients. Using solely IP-multicast is not a feasible solution in all cases, since the system is designed as a general purpose group communication layer in a web-based environment, where some clients are connected through ISPs that do not provide IP-multicast capabilities. Moreover, group membership awareness and secure and reliable communication are harder to implement over IP-multicast and also require point-to-point connections.

Another alternative for organizing the groups is to split each group over multiple servers, taking advantage of the location of the users relatively to the servers. This eliminates some of the network traffic due to the broadcast of a message to large groups and also reduces the load per server. This approach is more scalable for large groups and it is the one we chose in the design of our system.

We have designed and partially implemented a replicated version of the Corona server, described in the subsequent sections. The crash recovery scheme presented in section 4.2 has not been fully implemented yet.

4.1. Topology

Our replicated architecture has a star-like topology (Figure 2), with one server acting as coordinator and the other servers being its clients. The coordinator acts as a sequencer for messages. A multicast message is assigned a unique *sequence number*, which increases monotonically and thus imposes a total order on multicast messages within a group. When a client sends a broadcast message to its server, the server forwards the message to the coordinator, which distributes it to the whole group through the corresponding servers. Only the servers who have members in that particular group will receive the broadcast message. By using a centralized sequencer we obtain a total and causal order of the messages, and a FIFO order with respect to a sender of messages. To increase the efficiency, in some cases, the totally ordered semantics may be relaxed (e.g., when a new client joins a group, leaves a group or changes a parameter which does not directly affect the other members of the group) and a broadcast message may be distributed locally by the server connected with the sender of the message before being sent to the clients connected to other servers.

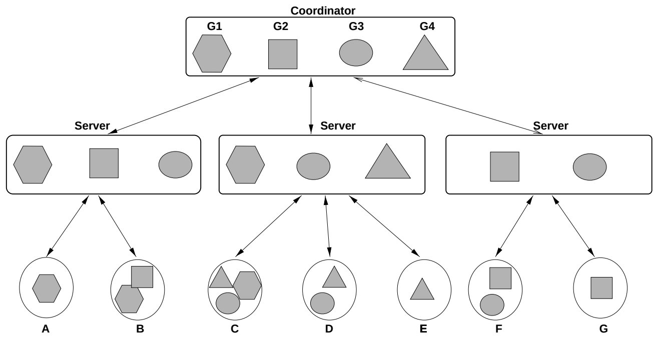

Figure 2. Replicated Corona Server. Rectangles represent servers, circles represent clients, and different shapes represent different shared states. Clients may belong to different groups: client A belongs to group G1, Client B belongs to groups G1 and Group G2, etc.

Using an architecture with a coordinator and a few tens of servers, we can achieve a potential scale up in the number of clients supported of at least an order of magnitude for the price of doubling the costs (communication cost + the cost of message handling by the server). Since the coordinator manages only a reduced number of connections, it is not heavily loaded and thus it does not represent a performance bottleneck. Moreover, since it is assumed that the servers run on well established and reliable hosts, it is possible to use IP-multicast for broadcasting messages among the servers, while also maintaining point-to-point connections for direct communication and for running the crash recovery protocol.

Depending on the scale that we want to achieve, the topology can be extended to a tree with multiple levels. In this situation the complexity of the recovery scheme in case of a crash increases, since the tree has to be reorganized. For the tree structure we also have to consider more elaborate algorithms for balancing the load on different servers and on communication paths.

Recall that, in order to ensure fast state transfer, the Corona service keeps the state of a group. The replicated service maintains multiple copies of the state for a group. All the server replicas who directly support some members of a group keep a copy of the state for that group. At least two copies of the state exist at any moment, in order to provide a hot standby in the case of a server crash. Otherwise, although a server saves the state on stable storage, the information may be unavailable during the time the server is down. When there is only one server replica which supports members of a group, a backup is elected from one of the other servers. The above approach works for availability purpose provided that there are no simultaneous server crashes. The next section shows how the system can be extended to support multiple server crashes.

4.2. Fault tolerance

A companion paper [15] shows how we built support for fault-tolerance in a single server implementation. It also shows how to deal with client or link failures and how to maintain state consistency through client reconnection. This section discusses the issues encountered in the design and implementation of a fault-tolerant replicated service.

Several approaches to the design of fault-tolerant replicated systems exist [14, 10], depending on the failure assumptions. We assume the *fail-stop* fault model. The solution we propose is similar to the *state-machine* model in the sense that multiple sites maintain the shared state and provide services simultaneously. A difference is that not all the servers keep the state for all the groups. For any group there exist at least two sites that maintain the state of the group, as in the *primary-backup* model, but these sites are active simultaneously. Another difference from the state-machine approach is that the servers are not identical, one of them having the special role of coordinator.

The configuration of the communication service is not fixed, servers may crash and new servers may be started. It is important that every server has enough information to establish connections with other servers. All the servers, including the coordinator, maintain a list (sorted in the order the servers have been brought up) of the other servers, containing their IP addresses and port numbers. This information is loaded at startup from the configuration files and it is updated as a result of the changes (server joins or leaves) sent from the coordinator to every server. Maintaining this information can be done for a negligible cost, since the addition and removal of servers does not happen too often.

To detect failures, we use heartbeat messages between the coordinator and the other servers and timeouts as upper bounds for communication delays. After an interval (greater than the heartbeat interval) in which the coordinator hasn't been able to communicate with a server, the coordinator assumes that either the server is disconnected or it is down, in which case it removes it from the list. When the coordinator crashes, the first server in the list becomes the new coordinator. Due to the heartbeat, the servers should detect the coordinator crash at about the same time. The first server sends a message to all the other servers and it assumes the role of coordinator when it receives acknowledgments from half+1 of the remaining servers. If the first server wrongfully assumes that the coordinator is down, (some of) the other servers will notice this and will respond with a *nack*.

A system made up by $k + 1$ servers can tolerate k simultaneous crashes by using increasing timeouts. If the coordinator does crash at the same time with the first server, the first server in the list which is up and running will become the new coordinator. An increasing timeout interval is allowed for each of the servers at the top of the list, e.g., the first server detects that the coordinator is down after time t, the second server detects that both the coordinator and the first server are down after time $2t$, and so on. Alternatively, one of the election algorithms existing in the literature [7, 6] may be used in case of a coordinator crash to select a new coordinator.

In case of a *network partition*, there will ultimately exist two subsets of the server set which run without having knowledge about each other. From the moment the partition occurs, the two server subsets evolve separately and the state updates are made independently in the two subgroups based on the client activity in each subgroup. When the network connectivity between the two subsets is re-established, for each group the last globally consistent state is identified based on the previous checkpoints and the sequence numbers assigned to the state update messages. The application is given the choice of either rolling back to the consistent state, selecting one of the available updated states or evolving as two different groups. The selection of one of these choices is application dependent and should be implemented in the client code.

5. Implementation Status and Performance

5.1. Implementation

The communication servers in the Corona architecture have been implemented as multi-threaded Java applications, supporting downloadable Java applet clients. Corona supports the multicast, membership, state transfer, state log reduction, and synchronization services. The collaboration tools built on top of Corona include a chat box, a draw tool, and a set of instrument data viewers. The chat box provides an edit area for composing messages and a scrollable area for displaying a list of received messages. Similar both to a shared notebook and a whiteboard in its functionality, the draw tool provides a canvas for drawing, taking notes, and importing images. The data viewers provide configurable windows for displaying different kinds of instrument data. The awareness of other users is provided via a membership status window that displays the group membership service's notifications of group membership changes.

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

5.2. Performance Measurements

A goal of our architecture is that the management of shared state by the Corona multicast service should not significantly increase the cost (in terms of message latency) of message multicast or impact the scalability of groups compared to groups without server-maintained shared state, where the server acts as a sequencer only. We compared the performance of group broadcasts when the service maintains shared state and when the service does not maintain shared state, in terms of client throughput, round-trip delivery time to clients, and scalability in terms of the number of clients and message size. In the implementation evaluated, Corona's group multicast service sends a multicast via multiple point-to-point messages from the server to clients[3], using TCP connections.

[3]in contrast to using IP multicast

5.2.1. Overhead of Shared State Maintenance

In Figure 3, we compare the round-trip delay for a multicast using a single server maintaining state (both in memory and on the disk) with the case of a server not maintaining state. The message size is fixed at 1000 bytes, typical for our client applications. For these tests, the machines used were a mix of Sun Sparc 20s and UltraSparc 1s running Solaris and connected through a 10Mbps Ethernet LAN. The server runs as a stand-alone Java application on a UltraSparc 1 with 64 MB RAM.

In this experiment, all clients but one are just receivers; they connect to the server, join a group and receive the broadcast messages addressed to that group. The extra client is both a sender and a receiver and it is used to measure the round-trip delay. This client is the last one (in the group) a broadcast message is sent to, therefore the values measured correspond to the worst case. The clients are at any moment uniformly distributed over 6 machines. A data point is obtained by averaging over 600 successive messages, sent with the rate of a message every 100 msec. The standard deviation ranges between 2-19% of the mean value measured.

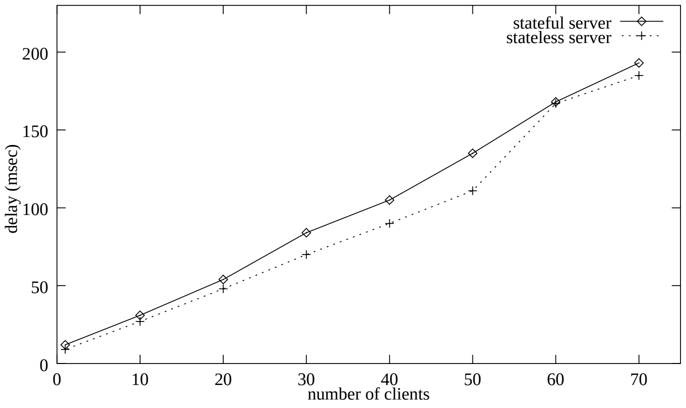

Figure 3. Group multicast with a single server: Round-trip delay vs. #clients for messages of size 1000 bytes. The latency is almost identical regardless whether the server does logging or not.

The round-trip delay increases approximately linearly with the number of clients, when the message size is fixed. As illustrated in Figure 3, the overhead of maintaining the state at the server is for the most part minimal; the two curves are very close to each other. Since the size of the message is fixed in this example, the overhead of storing the state in the server's internal data structures is constant regardless of the number of clients. Other costs such as network communication latency, data processing in the application protocol stack, thread scheduling and occasional garbage collection in the server application are included in the round-trip delay seen by a client.

In these and other experiments, we observed that for messages of size up to a few hundreds of bytes (the typical size of messages generated by the chat or draw tools) the size makes little difference in round-trip times. The influence of the message size is more evident above 1000 bytes. A significant part of the cost associated with broadcasting a message is due to the serialized read/write operations on the shared objects and can be reduced by overloading the default JDK implementation of these methods.

We repeated the experiment with messages of size 10000 bytes and the delay remained linear with the number of clients, but with a higher slope. The delay increases significantly as the number of clients grows, since, for example, for 60 clients, a broadcast rate of one message per second would correspond to an aggregated throughput of 600 kBytes/sec.

5.2.2. Throughput

We also measured the server throughput obtained using 6 clients running on separate machines (Sun Sparc 20s and UltraSparc 1s) and multicasting data as fast as possible through the Corona server, running either on an UltraSparc 1 with 64 MB RAM running Solaris or on a quad processor Pentium II 200 with 256MB RAM running Windows NT. The machines are connected over a 10Mbps Ethernet network. Table 1 presents the results.

	1000 bytes	10000 bytes
UltraSparc 1	91	251
Pentium II	267	455

Table 1. Server throughput obtained using multicast messages of size 1000/10000 bytes

For each one of the server machines used, the limitation of the system did not seem to be as much in the server code as in the network capacity and the inability of some of the slower clients to send, receive and process data at a faster rate, since every time a new client was added, the server throughput increased. By adding more clients we have been able to sustain a throughput of 600 kbytes/sec using the NT server. Moreover, the CPU utilization for the machines running the server code rarely went above 50%. We also noticed significant differences in the throughput obtained by clients running simultaneously on different machines.

5.2.3. Replicated Service Performance

We ran some stress tests using an architecture made up by a coordinator and six servers running on a combination of SPARC machines running Solaris and Intel Pentium machines running Windows NT and with the clients distributed over 12 machines (in the same domain with the servers, but some of them in different local networks, situated a few routers away). Typical results for the round-trip latency for a broadcast message are presented in Table 2. These numbers show that by using the replicated service, in addition to increasing the fault-tolerance of the system, better scalability and responsiveness to user requests are achieved.

	100 clients	200 clients	300 clients
single server	160	350	
multiple servers	40-63	78-100	120-150

Table 2. Roundtrip delay (msec) for a multicast message of size 1000 bytes, using a single server vs. multiple servers

5.3. Current and Future Work

We have implemented a QoS-based adaptive version of the Corona service [11], based on priorities and explicit control over the scheduling of different activities and on dynamic adjustment of its policies according to system load, user input, application requirements and current global configuration. We have also developed a version of the communication system which uses both IP-multicast, whenever possible, and point-to-point TCP connections in order to implement scalable and reliable group communication.

For the future, we intend to extend the range of applications that make use of the communication services provided by Corona. More experience is needed in order to determine the usability of our system and potential directions for extending the services offered. We also intend to add security mechanisms and access control to the system.

6. Conclusions

In this paper, we argued for a group multicast service that also manages replicas of shared state - a set of shared objects - when providing support for computer-supported collaboration in Internet-like environments. Traditional group multicast services solely rely on client processes to manage the consistency of shared state that is fully replicated at clients. In such services, accommodating a new process to a group may block operations of existing clients for an unpredictable amount of time, especially over a wide area network, as they run a complex state-transfer and membership view agreement protocol. In a highly interactive collaboration environment, such a performance degradation may significantly reduce the effectiveness of collaborative work of end users. On the other hand, client applications are unreliable, may crash or may operate in disconnected mode, connecting occasionally for the purpose of communicating with other applications and shared state transfer.

Our work on designing and implementing a stateful group communication services shows the following:

• The implementation of a stateful multicast service is feasible and the overhead of maintaining the state at the service is most of the time negligible, compared with other costs such as network communication overhead, thread scheduling and data processing in the application protocol stack. State logging does not depend on the semantics of the data and it is not in the critical path as far as communication latency is concerned; the service can multicast data to a group in parallel with disk logging. There exists a chance that in the case of a server crash some of the latest updates of the shared state have not been flushed to the disk and they are lost. For typical applications, we consider this risk acceptable, since servers can be replicated and if none of the replicas has logged an update, the update message can be retrieved by the crash recovery algorithm from the original sender of the message, based on the sequence number assigned to the message.

• State logging could limit the server throughput due to disk I/O (typical disk transfer rate is around 3-5 Mbytes/sec). But techniques such as RAID, log-structured file systems or main-memory logging with power backup could be used in order to obtain better I/O bandwidth to the log device.

• New clients join a multicast group and transfer the shared state of the group fast from the communication service. Moreover, the group state is persistent, being maintained even after the group has assumed null membership.

• The multicast service does not have any knowledge of the semantics of data, thus allowing the exchange of information among all types of communication processes and potentially supporting a broad range of applications.

• Our experience with using the system shows the feasibility and usefulness of a group communication service that maintains the shared state at the server. The Corona service has run for several months in a row without experiencing any problems, but client applications (i.e., applets running in browser windows) crashed occasionally. Maintaining the state of a group at the client would have led to a state loss when the client crashed.

• A potential problem with logging the state at the service is that maintaining the state for numerous groups simultaneously may cause a server to exceed its available resources, e.g., memory and disk space. In the case of collaborative applications, in which clients are trusted, subject to authentication and access control, and the collaboration activity has predictable resource needs, the size of the shared data can be easily managed. But for some other distributed applications, such as data dissemination services, large amounts of data may be generated. One way to deal with this problem is to offload the logging of the shared state for certain groups outside the communication service, to application specific servers which act as clients for the communication system and can do some semantic processing of the data, such as compression, checkpointing, etc, in order to reduce the size of the shared state.

References

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

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

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

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

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

[6] N. Fredrickson and N. Lynch. Electing a Leader in a Synchronous Ring. *Journal of the ACM*, 34:98–115, Jan. 1987.

[7] H. Garcia-Molina. Elections in a Distributed Computing System. *IEEE Trans. on Computers*, C-31(1):48–59, Jan. 1982.

[8] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Trans. on Software Engineering*, 17(1):64–76, Jan. 1991.

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

[10] B. Liskov and R. Ladin. Highly-Available Distributed Services and Fault-Tolerant Distributed Garbage Collection. In *Proceedings of the Fifth ACM Annual Symposium on Principles of Distributed Computing*, pages 29–39, Calgary, Canada, 1986.

[11] R. Litiu and A. Prakash. Adaptive Group Communication Services for Groupware Systems. In *Proceedings of the Second International Enterprise Distributed Object Computing Works hop (EDOC'98)*, San Diego, CA, Nov. 1998.

[12] S. Mishra, L. L. Peterson, and R. D. Schlichting. Implementing Fault-Tolerant Replicated Objects Using Psync. In *Proc. of IEEE 8th. Symp. on Reliable Distributed Systems*, pages 42–52, Seattle, WA, Oct. 1989.

[13] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. *Distributed Systems Engineeering Journal*, 1(2):87–103, Dec. 1993.

[14] F. B. Schneider. Implementing Fault-Tolerant Services using the State-Machine Approach. *ACM Computing Surveys*, 22, Dec. 1990.

[15] H.S. Shim and A. Prakash. Tolerating Client and Communication Failures in Distributed Groupware Systems. In *Proc. of the Symposium on Reliable Distributed Systems (SRDS)*, Purdue, 1998.

[16] R. van Renesse, K.P. Birman, and S. Maffeis. Horus, a flexible Group Communication System. *Communications of the ACM*, Apr. 1996.