Robust Chat for Airborne Command and Control

by

Prasad Ramanan
B.S. Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2003

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

AUGUST 22, 2003

The author hereby grants to M.I.T. permission to reproduce and distribute publicly paper and electronic copies of this thesis and to grant others the right to do so.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 2 0 2004
LIBRARIES

Author______________________________
Department of Electrical Engineering and Computer Science
August 22, 2003

Certified by______________________________
Clifford J. Weinstein
Group Leader, Information Systems Technology Group, MIT Lincoln Laboratory
Thesis Supervisor

Certified by______________________________
Roger I. Khazan
Staff Member, Information Systems Technology Group, MIT Lincoln Laboratory
Thesis Supervisor

Accepted by______________________________
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

This research was sponsored by the Department of the Air Force under Air Force Contract F19628-95-C-0002. Opinions, interpretations, conclusions, and recommendations are not necessarily endorsed by the US Government.

BARKER

Robust Chat for Airborne Command and Control

by

Prasad Ramanan

Submitted to the
Department of Electrical Engineering and Computer Science

August 22, 2003

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

We present the design, implementation, and evaluation of a prototype robust textual chat application to be utilized in dynamic distributed environments such as the Multi-Sensor Command and Control Constellation (MC2C). The MC2C environment consists of a set of airborne and ground sites, each of which contains a cluster of clients. Intra-site communication is reliable and exhibits high performance, whereas the performance and reliability of inter-site communication is variable and unpredictable. The two primary goals of the chat application are to deliver messages to clients with low latency and a globally consistent ordering. Since these goals conflict, our protocols strike a balance by satisfying a new property that we call the Intermittent Global Order (IGO) property. A protocol satisfying the IGO property guarantees global order while network connectivity permits, and sacrifices global order for bounded message delivery latencies while maintaining an intuitive and well-defined ordering on the message delivery when critical network connections are lost. We implemented our protocols on a hierarchical system architecture that places a server at every MC2C site. We developed a test-bed that simulates four MC2C sites in order to test the prototype. Testing revealed that the various IGO protocols implemented in the prototype all achieve the goals of robust and efficient collaborative communication even in the face of frequent link outages, but differ in how each balances global order and latency.

Thesis Supervisor: Clifford J. Weinstein
Title: Group Leader, Information Systems Technology Group, MIT Lincoln Laboratory

Thesis Supervisor: Roger I. Khazan
Title: Staff Member, Information Systems Technology Group, MIT Lincoln Laboratory

Dedicated to
Bhagavan Shri Sathya Sai Baba
“Anyadha Sharanam Nasthi
Thwameva Sharanam Mama”

Acknowledgements

I'd like thank my advisors, Cliff Weinstein and Roger Khazan, for giving me this opportunity and for making it a rewarding experience. Roger – thanks for your detailed and insightful feedback and for your patience. Scott Lewandowski provided valuable insight at every step of the way and got me through a number of challenges, technical and otherwise – thanks for everything, Scott. Tom Parks was a valuable addition to our group.

A number of friends at MIT made my experience unforgettable; Bobby, Bilal, Omar, Faisal, Bassel. Bobby, I'm including a prayer for your mother in this thesis; by God's grace everything is always fine.You are one of the most selfless and dedicated friends I have ever had. Oka bye. Bilal – I am truly grateful for you. What more can I say? I will always school you in basketball. Omar – thanks for the many insightful discussions and I am looking forward to many, many more. Faisal, bee-bee-ji, I am the rwc and you know it. Sorry for leaving jummah early, but how did you see me leave, if you were supposed to be praying? Bassel – Mr. le16, why don't you do something useful? Thanks, you guys, for everything.

The Brookline/Somerville Sai center was a home away from home, and I'll always be grateful.

Sai Kiran and the NJ boys – Sai Shyam, Sampath, Sri Sai, Swarup, Cooper; can we ever understand how much grace Bhagavan has given us?

Nithya and Nirmalan – thanks for your support and for the countless things you have both done for me.

Ammamma and Kittuppa, Patti, and all of the elders – everything good in my life is because of your blessings.

Amma and Appa – you taught me the real value of education and work. With Bhagavan's grace, I will live up to the ideals you have placed before me. Thank you for everything.

Contents

List of Figures

List of Tables

1 Introduction

This thesis covers the design, implementation, and evaluation of a prototype robust collaborative textual chat application to be deployed on a Multi-Sensor Command and Control Constellation (MC2C), which includes a set of aircraft and their supporting ground sites. Collaborative textual chat (hereafter, chat) is defined as a broadcast of text messages with certain ordering properties on the display; the most desirable property on the ordering is that all users see the same sequence of messages displayed on the screen. The chat application will be used for communication amongst aircraft crew and ground crew during ground moving-target indication (GMTI) and air moving-target indication (AMTI) missions.

This thesis is motivated by the fact that traditional chat architectures are vulnerable and suffer from service degradation in networks with varying connectivity, which is a salient characteristic of the MC2C environment. We address these problems by designing a chat application specifically for dynamic environments patterned after the MC2C environment. Our goal was to design a chat application that maintains message delivery latencies that are low enough to support human conversation and imposes intuitive ordering properties on the message display despite the presence of disconnections and network partitions. Our prototype achieves this goal by adapting existing algorithms for globally ordered message delivery so that they provide service with bounded latencies of message delivery while maintaining relaxed and intuitive message ordering properties on the display regardless of the state of network connectivity. We evaluate the prototype in a test environment built as a part of this thesis and designed to simulate a subset of the operational characteristics of the MC2C environment and we show that the prototype exhibits the desired properties.

Our solution utilizes a hierarchical design that serves to isolate the unreliable portion of the network and provide fault tolerance and uninterrupted collaboration amongst groups of well-connected chat clients. We also define and implement an ordering property known as Intermittent Global Order (IGO) which imposes a global order on message display as long as message latencies are bounded, and otherwise sacrifices global order for a less strict but still well-defined ordering. Intermittent Global Order is an ordering property that recognizes the important tradeoff between the conflicting goals of low latency and globally ordered message delivery.

This thesis is organized as follows. Chapter 1 introduces the problem and outlines our goals. Chapter 2 presents background and related work. Chapter 3 describes the MC2C environment and its characteristics. Chapter 4 presents the general design of the chat prototype, its accompanying protocols, and their properties. Chapter 5 is an overview of the implementation of the chat prototype. Chapter 6 describes the test-bed, test experiments, and their results, and Chapter 7 describes directions for future work.

1.1 Previous Experience with Chat on MC2C

The importance of chat as a military application has been underscored by its use in the Joint Expeditionary Forces Experiment 2002 (JEFX 2002) as well as its widespread use during the wars in Afghanistan and Iraq [1, 2]. Colonel John Feda, the Air Staff's deputy director for surveillance and reconnaissance systems, said after the Iraqi war that as a result of chat, "...the information flow and the [situational awareness] of everybody is higher, so the response time is lower" [2].

JEFX 2002 utilized an off-the-shelf chat application supporting tens of public access chat rooms with hundreds of users distributed between one aircraft and one ground station, and involved one air-ground communications link. The chat application used a single ground-based server; messages for each chat room were sent to this server for ordering and distribution to users located on the aircraft and on the ground. The results of the experiment were mixed. The potential usefulness of chat for MC2C operations was beyond question. Chat based collaboration enabled the passage of numeric coordinates between operators and for the verification of sensor and target locations. It facilitated the free and open communication of ideas for solutions and served as a free channel to request and receive information [3].

However, the usefulness of the chat application was limited by several deficiencies. The Tactical Common Data Link (TCDL) used for air-ground communication requires line-of-sight for successful communication. As a result, it suffered from periodic dropout due to wing blockage during mission orbit. It also suffered from intermittent failures due to hardware limitations and Radio Frequency Interference (RFI). The chat application was not designed to handle this degradation of the underlying communication service. It therefore failed without notification. Lt. Col. Terry D. Tichenor said after the experiment that "...we didn't understand how important the collaborative chat room would be, but it became a main situational awareness tool. As long as it was up, we were talking and making a lot of progress in terms of [collecting] test data. When it lost link, the system would shut down. We have to make sure that our applications are more robust in the dirty communications environment" [3].

In JEFX 2002, a single air-ground TCDL link caused severe performance impediments. However, future MC2C experiments and missions will involve multiple air-ground and air-air links. The MC2C environment envisioned for future experiments involves anywhere from 2-10 sites, where a 'site' is either an aircraft or a ground location, with up to a hundred users located at each site. Links connecting users within a site (intra-site links) are high performance and highly reliable (for example, Ethernet). However, links that connect different sites (inter-site links) have extremely varying characteristics. Inter-site links are unreliable; outages ranging anywhere from milliseconds to minutes are to be expected. Figure 1 outlines a vision of the future MC2C environment; in chapter 3, we provide a more detailed description of the MC2C environment. The prototype that we present in this thesis is designed to be robust and to provide high performance when deployed in future MC2C environments.

1.2 Goals

We start by outlining the performance and design goals we had in mind when designing the chat prototype. We then outline the scope of the implementation of the prototype.

1.2.1 Performance Goals

In terms of performance, there are two important metrics; latency and the ordering of the message display. Latency is important since chat is a real-time application and the information that is transmitted is time-sensitive. The goal is not to minimize the latency of message display; rather the goal is to display messages with latencies small enough to support human conversation. The worst case message delivery latency should be bounded from above. For example, the difference between a message latency of 10ms and 100ms is not important for the purposes of chat, but message delivery latencies greater than 15 seconds may be considered unacceptable.

The second important metric is the ordering of messages. Ideally, each client should see messages in the same order and this order should respect causal relationships between messages. This condition on the ordering of the message display is known as global order and is defined in section 2.1. However, it is impossible to implement such a strong property during network partitions and disconnections without tolerating unbounded message delivery latencies; the reason for this is discussed further in section 2.4.

Therefore, a reasonable performance goal is to maintain global order as long as the resulting message delivery latency is smaller than a certain defined threshold. This threshold is defined so as to facilitate normal human conversation using the chat application. If that threshold is in danger of being exceeded, then messages should be delivered with a weaker but well-understood ordering, and the global ordering should be re-achieved as soon as possible. We refer to this performance property as the Intermittent Global Order property.

1.2.2 Design Goals

The design should be such that the prototype is scalable, layered, and modular, with well-defined functions at each layer. The prototype should be able to scale to support hundreds of users, since the MC2C environment consists of multiple sites, with up to a hundred clients at each site. As will be explained in chapter 1, the site paradigm is critical to the MC2C environment, so clients should be site-aware; each client should know the site in which it is located, and it should know the site at which other clients are located.

1.2.3 Implementation Scope

A chat application usually supports conversations in multiple chat rooms, allows for the creation and deletion of chat rooms, and includes protocol (possibly involving secure authentication) for joining and leaving chat rooms. Eventually, the chat application deployed on MC2C should support multiple chat rooms and include mechanisms for security and authentication[1]. However, to simplify the implementation, our prototype provides one chat

[1] It would also be useful to regulate access to chat rooms according to security clearance, and have varying levels of encryption for chat rooms with varying security clearances.

room with global membership. We also decided not to have any application-level security or encryption. The absence of multiple chat rooms allows us to ignore security issues associated with membership and access control and to focus on the problem of creating reliable and robust broadcast primitives with ordering properties that balance the desired latency requirements. A justification for this simplification is that a protocol for multiple chat rooms can easily be built on top of the broadcast primitives provided by our prototype. Similarly, application-level security and access control are higher-level services that can be built on top of reliable and robust message delivery primitives.

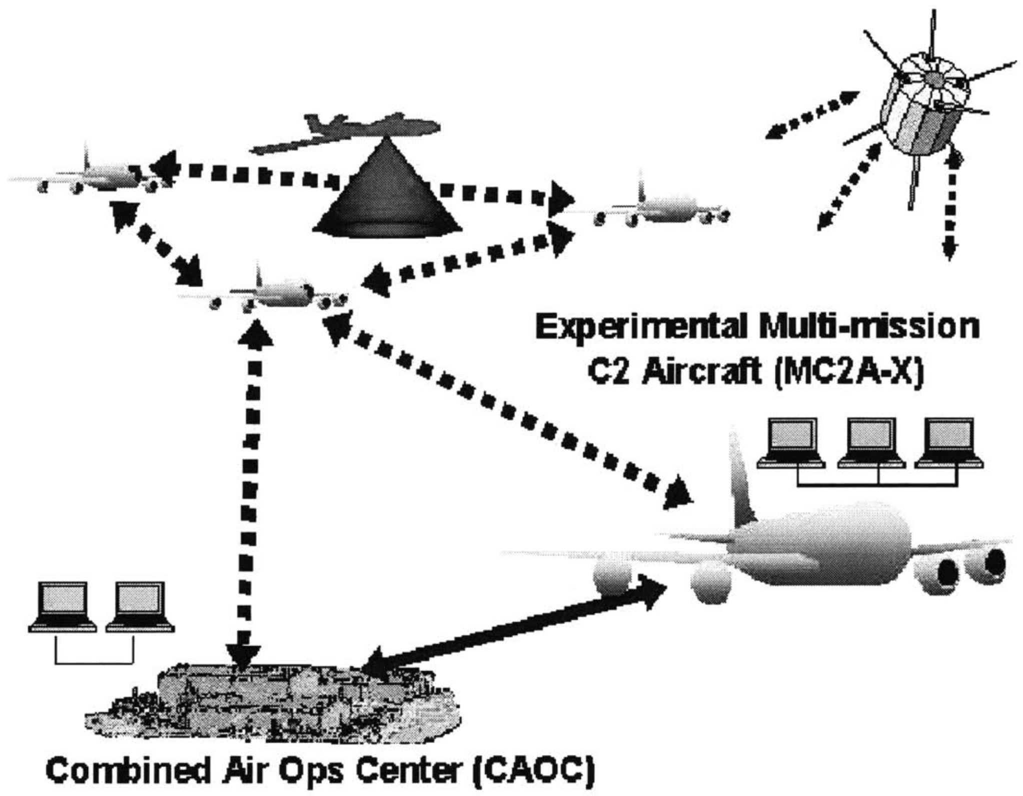


Figure 1: Future MC2C environment. Sites include the Experimental Multi-mission Command and Control Aircraft (MC2A-X), and the Combined Air Operations Center (CAOC). Each site has a number of end users. Notice the hierarchy inherent in the system architecture; there are multiple sites, each with a number of users.

1.3 Thesis Contributions

Our work has resulted in a prototype robust chat application designed to run in environments similar to the MC2C environment. The prototype has a base architecture that takes advantage of the site-based hierarchy inherent in the MC2C environment and provides the basis for efficient collaboration. We specify the Intermittent Global Order property that recognizes the tradeoff between global order and low latency message delivery. The protocols in our prototype implement this property in order to achieve collaborative chat with bounded latencies of message delivery and intuitive message delivery orderings regardless of the state of network connectivity. Our prototype utilizes two lower-level protocols for globally ordered message delivery, and modifies these protocols to implement the IGO property. The two different protocols differ in how they balance global order and latency. The architecture and protocols developed in this prototype can be used to achieve robust collaboration with bounded latencies in any dynamic environment patterned after the MC2C environment.

We built a test-bed with four sites that simulates a subset of the operational characteristics of the MC2C environment: links can be brought up and down asymmetrically, link bandwidths are adjustable asymmetrically, and link status and bandwidth is configurable as a function of time. We ran numerous experiments involving network disconnections and partitions, using automated bots to exercise the chat application, and recorded statistics in order to verify that the prototype exhibits the desired behavior and performance. These experiments verify the robust properties of our prototype and provide us with insights to inform the design and implementation of a next-generation chat prototype.

2 Theory and Related Work

The theoretical issues that arise when designing a chat application for the MC2C environment are associated with the problem of broadcast in a distributed system. We can think of our chat application as a distributed system; each client is an independent process and the broadcast, receipt, and delivery of messages by processes are the events in our distributed system. Messages are broadcast between processes using an underlying communication mechanism such as FIFO broadcast. A process receives a message and may buffer it along with other received messages and perform some ordering on the received messages before eventually delivering it to the user. In our chat application, delivery corresponds to a message being displayed on the screen. An important issue is the order of message delivery; ideally the ordering must respect causal relationships between messages and be the same at every process.

In this section, we give an overview of the theory of broadcast in distributed systems. We first give an overview of the theoretical issues associated with ordering events and messages in a distributed system, and give definitions of the terms causal order and global order. We then describe a few common methods of achieving global order. Finally we evaluate existing implementations of broadcast services (implemented as group communication services). We also describe why these solutions are inadequate or sub-optimal for the problem of chat in the MC2C environment.

2.1 Ordering Messages in a Distributed System: Causal and Global Ordering

In this section, we define the terms causal order and global order. We depend heavily on [4] in our treatment of this material, and reproduce its definition of causal ordering. Our treatment in this section is independent of the algorithm used to broadcast messages; we view each process in our distributed system as a totally ordered sequence of events, where an event is the broadcast, receipt, or delivery of a message. In other words, for any two events x and y in a particular process, either x occurred before y or y occurred before x. We define causal order as a partial order $\rightarrow$ on the events of all of the processes in this system such that:

1. If x and y are events in the same process and x occurs before y, then $x \rightarrow y$.

2. If x corresponds to the sending of a message by a process and y corresponds to its receipt by another process, then $x \rightarrow y$.
3. If $x \rightarrow y$ and $y \rightarrow z$, then $x \rightarrow z$[2].

Figure 2 depicts a process-time diagram, in which time advances towards the bottom of the page, and messages sent from one process to another are represented by arrows. The beginning of the arrow is the broadcast of the message, and the end of the arrow is the receipt of the message. Delivery of messages is not explicitly represented in the figure. In Figure 2, examples of causally related pairs of events are *A:1* $\rightarrow$ *B:2* and *B:2* $\rightarrow$ C:4. Intuitively, $a \rightarrow b$ if a can causally affect b, which is why $\rightarrow$ is known as causal order.

We distinguish between the ordering of events and the ordering of messages. An event, as we explain previously, is the broadcast, receipt, or delivery of a message; causal order on events is defined above. A causal order on the messages is a closely related concept and can be defined in terms of the causal order on the events; we say that message a causally affects message b ($a \rightarrow b$) if the event 'broadcast of message a' causally affects the event 'broadcast of message b'. For example, in Figure 2, the message a_1 broadcast in event A:1 causally affects the message b_2 broadcast in event B:2; therefore, $a_1 \rightarrow b_2$.

We also note the distinction between the order in which messages are received and the order in which messages are delivered. We cannot control the order in which messages are received; this is determined by timing issues in the underlying communication system, so messages may not necessarily be received in a causal order. For example, in Figure 2, process C receives message b_2 before message a_1, although $a_1 \rightarrow b_2$. However, received messages can be buffered before they are delivered, and we can implement algorithms that ensure that the order of delivery of messages always respects causal relationships between messages.

For our chat application, we are concerned with the order imposed on the delivery of messages; at each process P, we are interested in the order of the delivery events at that process. The order of delivery events corresponds to a particular message order; we would like the corresponding message order to be a causal order.

A global order (on the delivery of messages) is defined to be any order of delivery such that messages are delivered in the same order at every process in the system. It is

[2] Two distinct events x and y are said to be concurrent if neither $x \rightarrow y$ nor $y \rightarrow x$

important to note that a global order respects causal order between messages; a global order on the message delivery implies that it is a global and causal order. Global order is an important concept because causal order is only a partial order; the latter admits the possibility of two concurrent events and correspondingly, concurrent messages[3]. Implementing a global order on the message delivery ensures that concurrent messages are delivered in the same order at every process.

Causal order on the message delivery is an important property for chat because a message delivery order that violates causal order would make the resulting conversation incoherent to the chat user; imagine receiving a reply before receiving the question. Global order is important because it ensures that all participating parties have the same view of the conversation. This is especially important in chat for military applications, because users may make important decisions based on the chat conversation. Achieving global order ensures that the users are making decisions based on the same data as other users.

[3] Concurrent messages are closely related to concurrent events; we define two messages *a* and *b* to be concurrent if the events corresponding to their broadcast are concurrent events.

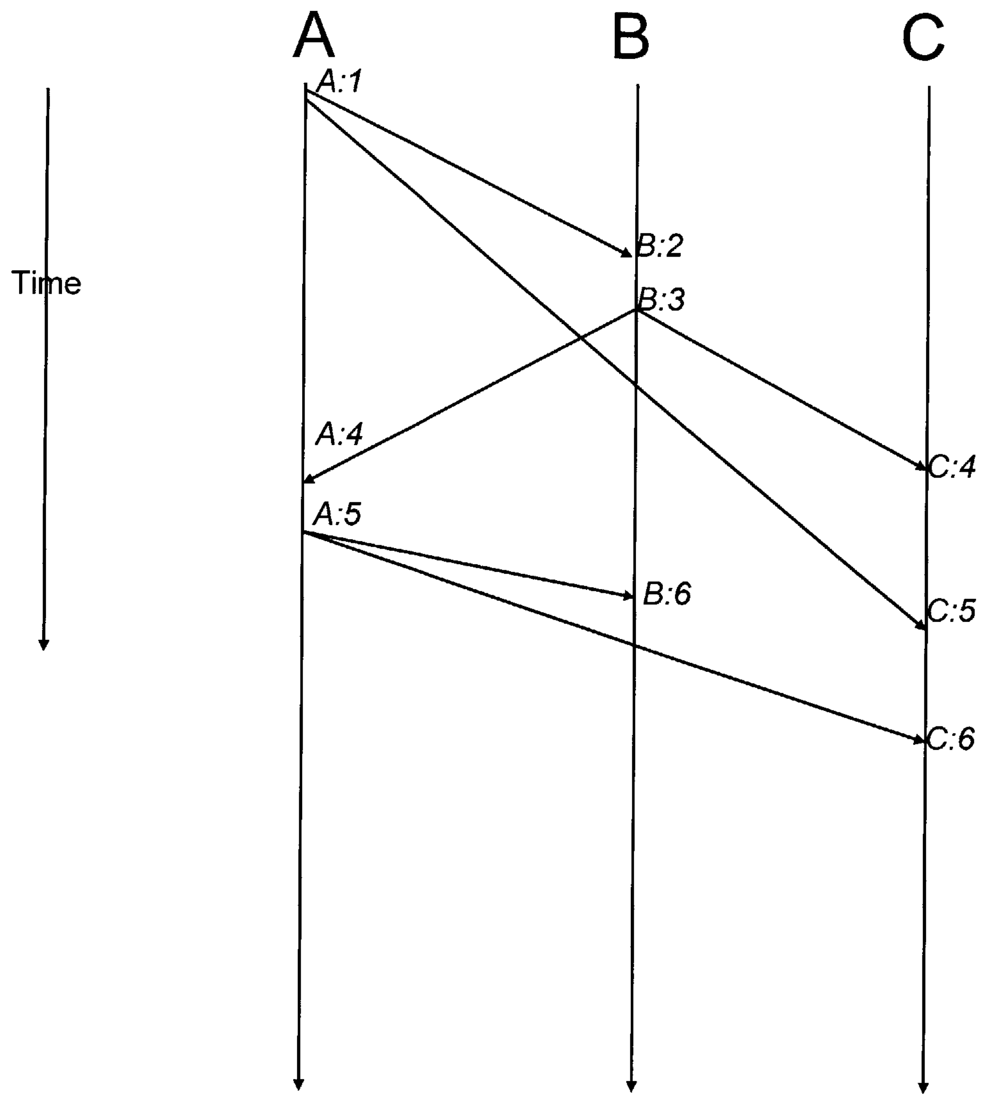


Figure 2: A process-time diagram with three processes. Time advances towards the bottom of the page. Each event is labeled by P:i, where P is the process name, and i is the Lamport timestamp associated with that event. (The concept of Lamport timestamps is defined in section 2.1.1.)

2.1.1 Lamport Clocks

We next introduce the concept of Lamport clocks, defined in [4]. A Lamport clock assigns integer 'times' to the events of a process. Formally, we augment each process P_i with a Lamport clock C_i that assigns a number (or time) $C_i(x)$ to every event x of that process. We define a global Lamport clock C such that $C(x) = C_i(x)$ if x is an event in process P_i.

Lamport clocks are logical clocks that have no reference to physical time. However, they keep time in such a way that it reflects the causal relationships between events. The condition satisfied by Lamport clocks is that if $x \rightarrow y$, then $C(x) < C(y)$, where C is the global Lamport clock. We will refer to this condition as the global Lamport clock condition. In other words, if *y* is causally affected by *x*, then *y* happens at a logical time that is greater than (or after) the logical time of event *x*.

Since Lamport clocks are logical times used to order events, we can think of the Lamport clock of each process P_i as a counter C_i that is updated by events; $C_i(x)$ is the value of the counter C_i at the event *x*. Each process follows a simple algorithm to update C_i according to events; this algorithm also ensures that the global Lamport clock condition is satisfied. We are only concerned with the events of sending and receiving messages, so we describe the algorithm in terms of these events.

1. event *x*: Process P_i sends a message *m*. Set $T_m = C_i(x)$, where T_m is the timestamp of message *m*. Increment C_i ($C_i = C_i(x) + 1$).
2. event *y*: Process P_i receives a message *m* with timestamp T_m. Set $C_i = \max(T_m + 1, C_i(y) + 1)$.

This algorithm assigns times to events such that if $x \rightarrow y$, then $C(x) < C(y)$; in other words, it ensures that the global Lamport clock condition is satisfied. The numbers labeling events in Figure 2 are examples of valid Lamport timestamps; they ensure that if X:i → Y:j, then i < j. For example, A:1 → B:2, and A:1 → C:4.

2.2 Methods of Achieving Global Order

We next give an overview of some common algorithms for achieving global order on the message delivery in a distributed system. In this section, we give an intuitive treatment of the material by introducing a taxonomy of methods based on the location at which the ordering is imposed. In the discussion that follows, we are concerned with broadcast among a group of processes P containing processes $P_1 \ldots P_n$. We assume that every process has access to a Reliable First-in-first-out (FIFO) broadcast service. Reliable FIFO broadcast from a source to a set of destinations is a broadcast in which no messages are lost and each destination receives the messages in the same order that the messages were sent.

2.2.1 Ordering at the Sender

An example of an algorithm that achieves global order by imposing the ordering at the sender is the ring-based method, described in [5]. This approach arranges the set of processes in a totally ordered logical ring. For example, we can arrange a ring in which we order the processes according to process ID[4], and P_n is followed by P_1. The privilege of sending is given only to the holder of a *token* that is unique to the ring. The token contains a sequence number that is used to label messages. A process holding the token that wishes to send a message labels its message with the sequence number from the token, increments the sequence number on the token, and uses the reliable FIFO broadcast service to send the message to the other processes. The token is passed sequentially along the ring. Some additional protocol may be in place in order to limit the amount of time a process can hold the token.

Processes expect to receive messages in sequential order. However, due to timing issues, a process may receive messages out of order. We illustrate a simple example with three processes- A, B, and C. Process A initially holds the token and broadcasts message m_1. It then passes the token to B, which broadcasts message m_2. If the A-C link is slow enough relative to the B-C link, then process C may receive m_2 before m_1. Since the messages are labeled with sequence numbers, this is immediately detected, and the out-of-order message is buffered until the in-order messages are received.

2.2.2 Ordering at an Intermediary

An example of an algorithm that achieves global order through neither the sender nor the receiver but rather at an intermediary process is the leader-based method. We describe two version of the leader-based method. In the first version, a process that wishes to send a message sends the message to an intermediary known as the leader. The leader maintains a FIFO queue of messages. As soon as it receives a message from a process, it is placed in the FIFO queue. The leader broadcasts messages to processes by removing messages from its FIFO queue. If we assume that the leader uses a reliable FIFO broadcast service to broadcast messages to each process, then there is no need to use sequence numbers in this version of

[4] The process ID of process P_i is i.

the leader-based method, since the reliable FIFO broadcast ensures that each process receives every message in order[5].

An alternative version works in the following manner. A process wishing to send a message requests a sequence number from the leader. In this case, the leader plays a role similar to the token in the ring-based protocol; it is in charge of assigning and incrementing the sequence number, which serves to globally order the messages. Once the process receives the sequence number from the leader, it marks its message with this sequence number and uses the reliable FIFO broadcast service to send the message to other processes. Similar to the ring-based method, a process may still receive messages out of order due to timing issues; this is handled by buffering until the in-order message is received.

2.2.3 Ordering at the Receiver

Finally, there are algorithms that rely on the receiver to achieve global order on the messages. We refer to the method of broadcast in which each process broadcasts its message to every other process as soon as the message is ready, without the aid of a global sequencer such as a leader or a token, as an all-to-all broadcast. The key to achieving global order on the message delivery when using an all-to-all broadcast is the algorithm used to deliver messages. We consider two different algorithms for message delivery; the first results in a delivery order that is neither global nor causal and the second achieves global order on the message delivery.

We first consider the effect of using all-to-all broadcast in combination with a very simple delivery rule in which a process delivers a message as soon as it receives it[6]. Consider the scenario depicted in Figure 2. A:1 is the event of A sending a message (call this message a_1) to the other processes. B:2 is the event of receiving message a_1; B:3 corresponds to the sending of a reply message (call it b_3) to the message a_1. However, b_3 reaches C before a_1 reaches C, so C will deliver b_3 before a_1. Processes A and B will deliver a_1 before b_3. Therefore the delivery order violates global order, and the delivery order at C violates causal order (since the message a_1 causally affects the message b_3).

[5] This version of the leader-based method is one of the most common algorithms used for chat programs. Many COTS chat implementations use this version with a simple failover mechanism (send messages to a predefined alternate leader) in case the leader is down.
[6] Henceforth we will refer to this delivery rule as the FIFO delivery rule.

Next we show a delivery algorithm that achieves a global order; this delivery algorithm was introduced in [4]. This algorithm makes use of Lamport timestamps maintained according to the rules outlined in 2.1.1. In this algorithm, processes buffer received messages in a totally ordered list. The list is sorted according to the timestamp of the message; ties are broken by ordering according to the process ID of the sender of the message. Additionally, each process tracks the greatest timestamp it has received from every other process.

The delivery process proceeds as follows. Messages are inspected for delivery according to their position in the totally ordered list; messages with smaller timestamps are inspected first. A message with timestamp n is only delivered if the process knows that the Lamport clock of every other process has advanced to be greater than or equal to n. Process X knows that process Y's Lamport clock has advanced to be greater than or equal to n when it receives a message from process Y with timestamp greater than or equal to n.

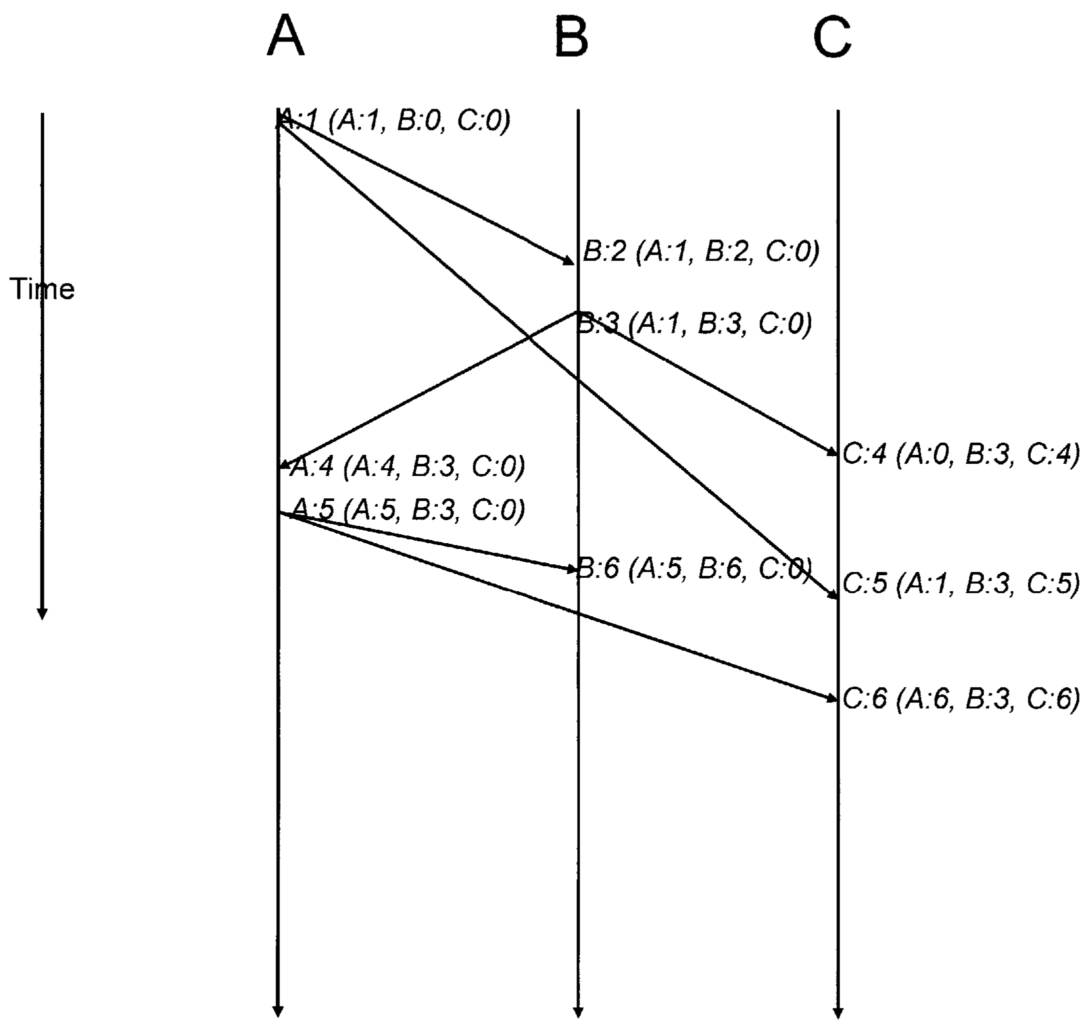


Figure 3: The same process-time diagram as Figure 2 with the additional clock-state-per-process required by the Lamport delivery algorithm.

Figure 3 depicts the same scenario as Figure 2 with the additional clock-state-per-process information required by the algorithm just described. When process C receives message b_3, it cannot deliver it, since the last message it heard from process A had timestamp 0, and $0 < 3$. As soon as process C receives a_1, it can deliver it, since at that point, it knows that the Lamport clock of every other process has advanced to be greater than or equal to 1. Furthermore, as soon as process C receives the message corresponding to A:5 (call it a_5), it can deliver b_3, since it knows that the Lamport clock of every other process has advanced to be greater than or equal to 3. This delivery algorithm, known as the Lamport delivery algorithm, achieves a global order on the message delivery.

There are various optimizations to this algorithm that serve to decrease the latency of message delivery. For example, since in the example process C is not sending any messages, processes A and B cannot deliver any messages, since they can never update their knowledge of C's timestamp. Therefore, it is useful to have each process send out periodic heartbeats so that every process can advance its knowledge of the clock states of other processes.

2.3 Group Communication Services: Existing Implementations

Group communication is defined as a means for providing multi-point to multi-point communication by organizing processes into groups [6]. A chat service is a simple group communication service; messages are broadcast to a group, and the users in a particular group see all of the messages broadcast to that group (including their own). There are various existing implementations of group communication services implementing broadcast that achieve different orderings ranging from unordered broadcasts to a globally ordered broadcast.

Systems that implement group communication primitives use a number of different methods for achieving delivery semantics such as global order. An overview of group communications systems and semantics is presented in [6]. Here we describe a sampling of systems that demonstrate the variety of options for achieving global and causally ordered multicast.

One of the first examples of a system that provides causal multicast, as well as a group membership service, is ISIS [7]. Causal multicast is implemented by piggy-backing causally related messages onto each message so that the necessary delivery order is explicitly known at the destination. ISIS suffers from performance degradation when causal order is respected due to the high message overhead. The authors also note that their implementation of causal order does not scale well to large numbers of sites.

Horus, a successor to ISIS, is introduced in [8]. Horus uses Lamport time and vector timestamps as the mechanism for achieving causal order, and depends on a multicast transport layer that provides FIFO-ordered multicast message delivery. This implementation significantly improves message delivery latency.

Ring-based ordering protocols are described in [5]. Ring-based ordering was implemented in the Totem system [9]. Performance measurements indicate that in the Totem system, for nodes that generate messages according to a Poisson process and in a network with very low loss rates, a significant number of messages are delivered with higher latencies than the network round-trip time, even for small network sizes.

A practical method for ensuring global ordering while minimizing the latency of message delivery is presented in [10]. The protocol introduced in [10] assumes that all nodes have access to a global time source. It uses global time and measurements of minimum channel latencies to set lower bounds on the difference between timestamp values of messages that are causally related. These lower bounds are used to deliver messages with minimal latency while maintaining global order.

2.4 Latency versus Global Order

There are various protocols and implementations in existence for achieving global ordered broadcast, which begs the question of why we do not simply use an existing protocol for chat in MC2C. The reason is because no existing protocol satisfies our performance objectives in our target environment. In the MC2C environment, disconnections and partitions are common events and cause message latency to grow unbounded in global order protocols. This is due to the fact that the goals of low latency message delivery and globally ordered message delivery are not complementary; guaranteeing global order usually increases the overall latency of message delivery, even when the network operating conditions are such that disconnections and partitions are unlikely[7]. Global order and low latency message delivery are conflicting goals because a global order implies the existence of a consensus, which is achieved by adding additional protocol that necessarily increases the latency of message delivery.

Disconnections and partitions exaggerate this tradeoff by making it more difficult to maintain the consensus required for the global order; only when the full required connectivity is retained can the consensus protocol work to achieve global order. For this reason, message

[7] This tradeoff is representative of a classic tradeoff in network protocol design between ordering and latency. For example, in TCP, which is a point-to-point protocol, the delivery of data is delayed until reliable FIFO order is guaranteed. The result is a greater latency of data delivery than, for example, UDP which is a point-to-point protocol with no ordering or reliability guarantees.

delivery latency in the presence of disconnections and partitions becomes unbounded. For example, in the Lamport-delivery algorithm, if a site fails then the delivery of messages at every other site is delayed until the site comes back up, resulting in unbounded message delivery latency. Similarly, in the leader-based algorithm, if a client is cut off from the leader then its message delivery latency is increased to the time it takes to reconnect to the leader. In the ring-based algorithm, if the site with the token is cut off from other sites, then the other sites cannot broadcast their messages until the token-holder is reconnected and can release the token. In each of the basic global order algorithms, the message delivery latency becomes unbounded due to network outages.

Therefore, we must develop a protocol that sacrifices global order for lower message delivery latencies while maintaining a well-understood ordering when network conditions are too harsh. In this thesis, we developed protocols satisfying the Intermittent Global Order property that achieve bounded message delivery latencies regardless of the state of network connectivity, and maintain a well-understood ordering at all times.

3 Environment

In this chapter, we give a description of the environment in which the chat prototype will be deployed. The MC2C environment consists of two to ten sites. A site is either an airplane or a ground location; each site has up to a hundred clients. Intra-site communication is reliable and high bandwidth; the intra-site network will most often be a LAN. Inter-site links are unreliable and may exhibit highly variable degrees of performance. Table 1 is a table of the various types of inter-site links in the MC2C environment and their characteristics. The MC2C environment can be modeled as a general environment in which there are clusters of well-connected nodes, and inter-cluster communication is unpredictable with varying degrees of performance.

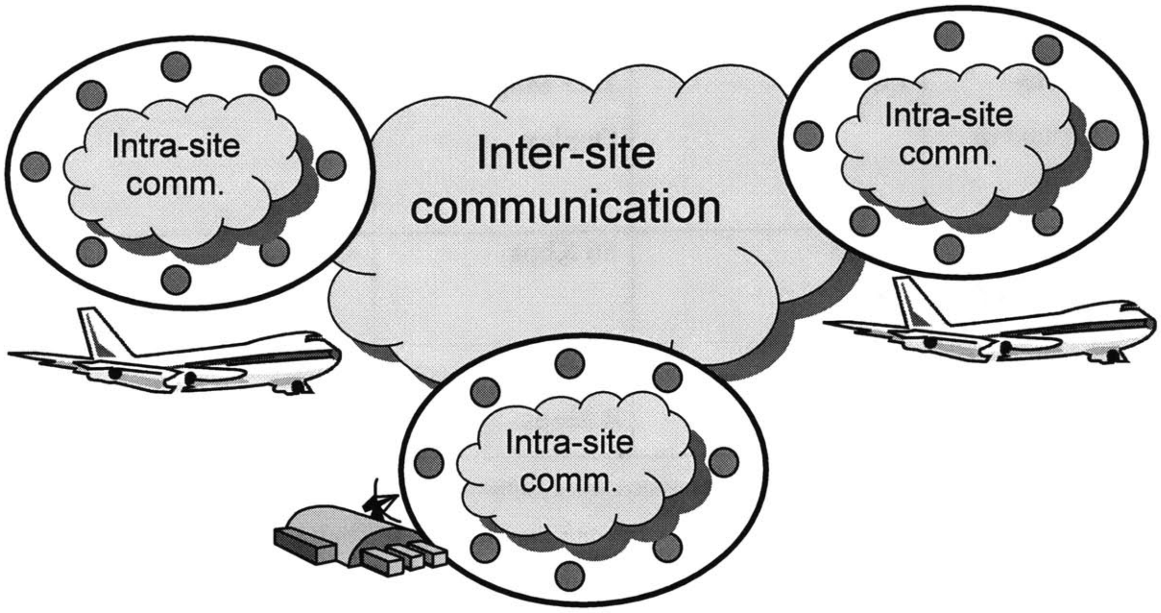


Figure 4: The MC2C environment. There are 3 sites in this figure; two planes, and a ground site. Each site has a number of end users. Notice the hierarchy inherent in the system architecture.

There are plans to develop a routing infrastructure for the MC2C environment that implements a property termed *transitive connectivity*, which we describe here. Our chat prototype does not require transitive connectivity in order to function correctly, but we describe transitive connectivity because the assumption that the routing layer underneath the chat prototype satisfies transitive connectivity simplifies the ordering properties on message delivery during times of disconnection. In section 4.3.6, we discuss the ordering properties that we achieve both with and without transitive connectivity.

Transitive connectivity is defined as follows. Let $a \leftrightarrow b$ mean that a and b are connected by a duplex link that is 'on'. Then $a \leftrightarrow b$ and $b \leftrightarrow c$ implies that $a \leftrightarrow c$. That is, suppose that there is a link between a and c that is currently 'off', perhaps due to network difficulties. Then as long as $a \leftrightarrow b$ and $b \leftrightarrow c$, then eventually $a \leftrightarrow c$, even though the direct link between a and c is down; this is because the routing layer discovers this transitive path and routes around the broken link between a and c. This property could be achieved by using a router at each site; transitive connectivity would then follow naturally from the algorithms of any common routing protocol such as OSPF [11].

Type of Link	**LOS /BLOS**	**Bandwidth (bps := bits per second)**	**Delay (ms := milliseconds)**
TCDL (point-to-point; air-ground or air-air)	LOS	10.7 Mbps Full Duplex	(Data not available)
Low-earth orbit satellite (LEO)	BLOS	56 Kbps	< 80 ms
Geo-stationary satellite (GEO)	BLOS	120 Kbps – 4 Mbps	> 250 ms

Table 1: Typical characteristics of different inter-site links. LOS := Line-of-sight. If a link is LOS, then it requires line-of-sight in order to operate. BLOS := Beyond-line-of-sight. If a link is BLOS, then it can still function when there is no line of sight. Satellite data is taken from [12].

We also include in the notion of transitive connectivity a specification of the time scale at which transitive connections are achieved. A routing protocol such as OSPF discovers the presence of a broken link through a heartbeat protocol. It expects to hear heartbeats at a regular interval. When it stops hearing heartbeats at that interval, then it considers the link to be down. In addition, an application-level protocol (e.g. our chat protocol) that operates across an inter-site link that is expected to come up and down will also generally run a heartbeat protocol in order to detect broken links in a timely manner; this application-level protocol has its own heartbeat-interval.

Transitive connectivity assumes that the router heartbeat interval is small enough relative to any application level heartbeat interval, so that the network layer detects and routes around faults (if a route is available) before the application sees any disconnect; this eliminates the need for application–level routing. We can say that we assume that the heartbeats of the underlying routing layer are at a timescale such that the connections represented by the transitive closure of the network layer are the same connections seen by the application layer at all times. This assumption also leads to the observation that the only disconnects visible to the chat application are true partitions. Transitive connectivity and some of its implications are illustrated in Figure 5.

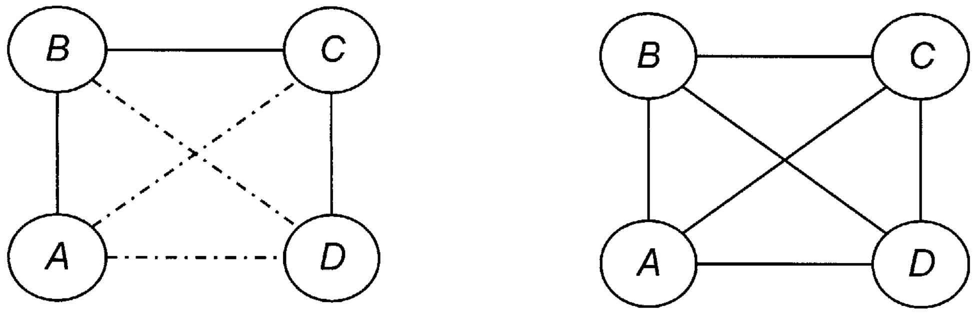


Figure 5: Illustration of the Transitive Connectivity Property. Each circle is a site, and the links represent inter-site links. The figure on the left represents the network-layer connectivity; note that the direct links A-C, B-D, and A-D are all down. The figure on the right represents the connectivity seen by the application layer. There are routers at every site that discover paths from every node to every other node, so the application layer sees full connectivity.

4 Design

In this chapter we give an overview of the salient characteristics of the design of the chat prototype. We first describe how the hierarchical nature of our base architecture takes advantage of the characteristics of the MC2C environment. This architecture defines a framework for the overall design and allows us to specify the relevant building blocks necessary to complete the chat prototype. We spend the rest of the chapter describing those basic building blocks. We describe our implementation of reliable FIFO broadcast, which is a building block of global order algorithms. We then outline the set of global order algorithms that our prototype utilizes. Finally we describe how we modify these global order algorithms to implement the Intermittent Global Order property, and give a specification of the resulting ordering.

4.1 MC2C-specific Chat Architecture: Reliable Broadcast Using Site-Based Hierarchy

There is a natural hierarchy in the MC2C system environment; there is a small number of sites (typically less than 10), but each site may have up to a hundred clients. Moreover, each site has a reliable intra-site communication infrastructure, while inter-site communication is unpredictable and unreliable. Our chat prototype takes advantage of this natural hierarchy. The environment suggests a site-based hierarchy in which there is a chat server located at every site. In this architecture, clients at each site communicate messages to their site's chat server through reliable intra-site links. The chat server then communicates messages sent by its site's clients to chat servers at other sites using the less reliable inter-site links. Chat servers receive messages from other sites, order them, and then deliver these ordered messages to the clients for display.

There are many advantages to an architecture that positions a chat server at every site. This architecture moves the function of ordering messages from the clients to the chat servers. Since there is only one chat server per site and there is a small number of sites, the number of messages exchanged as part of an ordering protocol between chat servers is much smaller than the number of messages that would be exchanged if the ordering protocol were run between every client in the system. In other words, the architecture serves to reduce protocol overhead by aggregating protocol at the site level instead of the client level. The architecture also takes advantage of the fact that sites are clusters of well-connected nodes and it isolates clients from the vagaries of inter-site communication while maintaining intra-site chat at all times. Finally, it serves as a starting point for the development of a protocol implementing the Intermittent Global Order property in the MC2C environment.

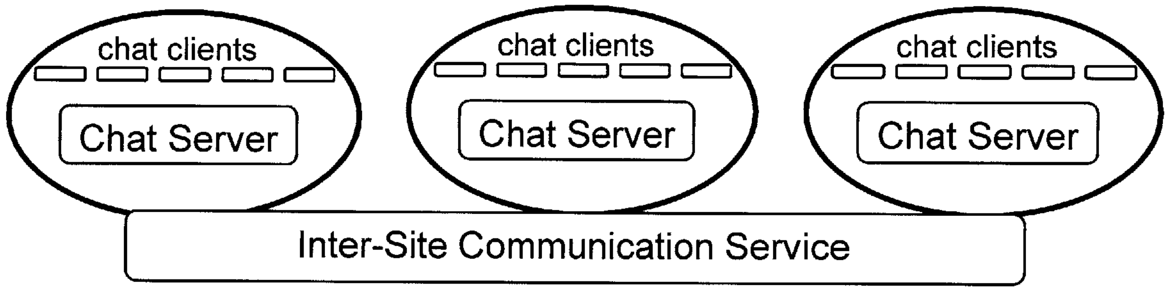


Figure 6: An architecture that takes advantage of the site-based hierarchy. There is a chat server at each site; chat clients communicate their messages to their site's chat server through the intra-site communication mechanism. Chat servers exchange messages using the inter-site communication service. The chat clients are 'thin'; they perform only the tasks of sending and receiving messages and displaying received messages. Ordering is taken care of by the chat servers.

In the following two sections, we describe concrete examples of the benefits of the site-based hierarchy by comparing the implementation of two common ordering protocols at the site-level as opposed to the client-level.

4.1.1 Leader-based ordering in the site-based hierarchy

We first consider the leader-based protocol. Implemented at the client-level, there is one leader, located at a particular site. Each client sends its message to that leader. The leader sends ordered messages back to clients for display. Consider the task of sending messages from the leader back to clients. When the communication from the leader to the clients is unicast, the leader sends multiple copies of the same message over an inter-site link with limited bandwidth and intermittent connectivity. If there are indeed up to a hundred clients per-site, then sending messages back to clients may consume up to one hundred times the minimum bandwidth necessary to communicate that information. Using multicast for leader-client communication would minimize the amount of redundant information sent over each link; however, reliable multicast protocols are not appropriate for achieving inter-site FIFO broadcast for our chat application, as discussed in section 4.2.

Another significant disadvantage to implementing the leader-based protocol at the client-level was exposed in the JEFX 2002 experiment, which utilized an off-the-shelf chat application that used a leader-based algorithm. In JEFX 2002, there were two sites; an aircraft known as the *Paul Revere* (site *A*) and a ground station (site *B*). The leader was located on site *B*. During the experiment, the inter-site link connecting *A* to *B* went down periodically, leaving the clients at site *A* unable to chat with anyone, since they were dependent on receiving messages from the leader, which was located on the other side of the broken *A-B* link. This is an unnecessary loss of service, since clients at *A* should have been able to chat with other clients at *A*.

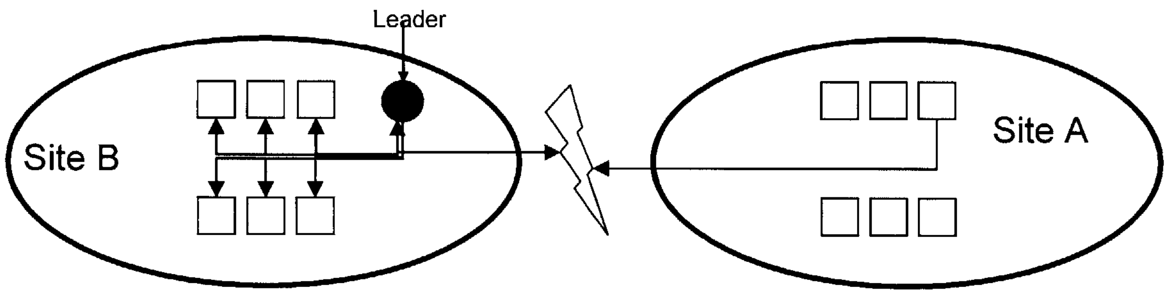


Figure 7: Scenario from Task Force Paul Revere experiment in JEFX 02. The 'leader' is located on Site B. When the A-B inter-site link breaks down, clients on A are unable to chat with anyone.

Thus, implementing the leader-based protocol at the client-level is inefficient and leaves clients exposed to the vagaries of inter-site communication. It fails to leverage the fact that sites are clusters of well-connected nodes to keep intra-site chat available at all times. Implementing a leader-based algorithm on top of the site-based hierarchy avoids these problems. There is a chat server located at each site; clients send their messages to their site's chat server, which then sends messages to a leader, which can be located at any site. The leader then sends ordered messages back to the chat servers, which deliver these messages to clients at their respective sites. Even if a chat server loses communication with the leader, then intra-site chat continues. In addition, the leader maintains inter-site chat service for all of the sites that are connected to the leader.

4.1.2 All-to-all ordering in the site-based hierarchy

Implementing the all-to-all algorithm at the client-level is even more problematic than the client-level implementation of the leader-based ordering. In all-to-all, every client sends its message to every other client; if the communication between every pair of clients is unicast, then a large amount of redundant data is sent over the links. This is compounded by the fact that the all-to-all protocol is dependent on periodic heartbeats for timely message delivery using the Lamport delivery algorithm, which increases the amount of redundant traffic flowing over sensitive links. Furthermore, there are potentially hundreds of clients across all sites, and each client must keep timestamp-per-client for every other client in order to implement the Lamport delivery algorithm. Implementing all-to-all at the site-level is much more reasonable. The number of sites in the MC2C environment is small (less than ten), so keeping a timestamp per chat server for the Lamport delivery algorithm is not unreasonable.

The conclusion is that the site-based hierarchy is an appropriate design for a chat application that targets the MC2C environment. It reduces protocol overhead through aggregation and it takes advantage of the fact that sites are clusters of well-connected nodes thus isolating clients from the vagaries of inter-site communication while maintaining intra-site chat at all times. It also serves as a starting point for the development of the Intermittent Global Order property in the MC2C environment.

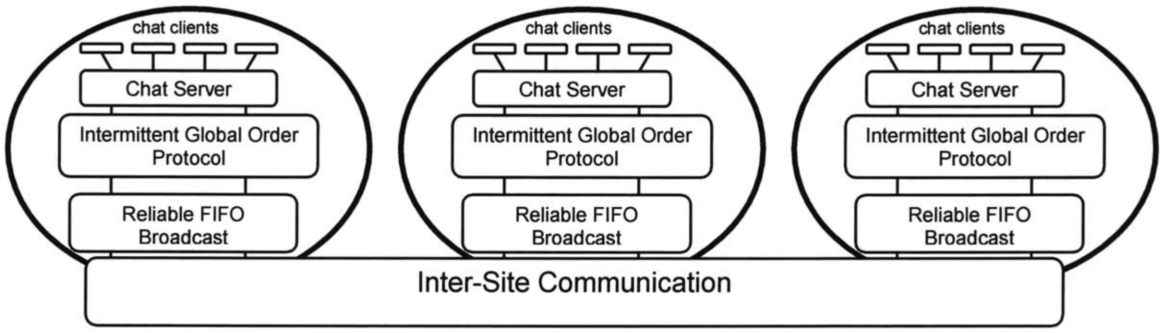


Figure 8: A more detailed view of the system architecture. The chat server at each site uses the Intermittent Global Order Protocol to broadcast messages, which in turn uses Reliable FIFO Broadcast as part of its protocol.

Having established the site-based hierarchy as our basic architecture, it is easy to see how to proceed with the design of the chat prototype. The chat server at each site uses the Intermittent Global Order Protocol to broadcast its messages; the IGO protocol uses a

Reliable FIFO broadcast, which is built on top of the inter-site communication mechanisms. In the following section we describe our design of Reliable FIFO broadcast.

4.2 Implementing Inter-site Reliable FIFO broadcast in MC2C

As described in section 2.2, reliable FIFO broadcast is an important building block for algorithms that achieve globally ordered message delivery. An important question is how to implement reliable FIFO broadcast between sites in the MC2C environment. Recall the definition of reliable FIFO broadcast; a broadcast in which no messages are lost and each destination receives the messages in the same order that the messages were sent[8]. Since we use a site-based hierarchy, we require an inter-site reliable FIFO broadcast in which the source and destinations are servers located at different sites. Inter-site reliable FIFO broadcast is a building block for global order algorithms that run at the site-level, as described in the previous section.

4.2.1 Requirements

It is useful to translate the requirements of our chat application into requirements for the inter-site reliable FIFO broadcast protocol. The chat application requires that destinations be decoupled since a poor connection between a source and one particular destination should not slow down the broadcast to other sites. Consider the effect on the reliable FIFO broadcast protocol of a long disconnection of a critical link connecting the source and a destination. During the disconnection, the protocol buffers data at the source for the purpose of resending that data to the destination as soon as the link comes back up. At the same time, the connection between the source and other destinations may remain healthy. Since destinations must be decoupled, the protocol must continue sending data to functioning destinations. Therefore, the amount of data that is buffered by the protocol for resending is unbounded. This is clearly a problem, since memory is finite and flushing the buffered data to disk will result in lower performance and a more complicated protocol. Even if buffer space is unbounded, there are problems with buffering too much data. Once the faulty link comes back up, the buffered data will be sent and received but the total delivery latency will be at

[8] A reliable FIFO broadcast is necessarily a 'connection-oriented' protocol; the protocol must effectively maintain state per destination, such as buffered data, sequence numbers, etc. in order to ensure that the data stream between source and every destination remains reliable and FIFO.

least as long as the period of the disconnection. Furthermore, since the stream must be FIFO, the resending of the buffered data increases the latency of delivery of more recent real-time data sent after the buffered data.

One of the requirements for the chat application is that the latency of message delivery is bounded. However, if the function of weathering disconnects and resending buffered data is located at the sender in a reliable FIFO broadcast protocol, then as explained above, the latency of message delivery is unbounded due to disconnections. Furthermore, the protocol indiscriminately buffers data regardless of the length of the disconnect, which results in greater memory use as well as longer latencies for real-time data sent after a reconnection.

We would like our FIFO broadcast protocol to weather short disconnections and lost data by buffering data for retransmission. However for longer disconnections (on the order of multiple seconds) we would like to close the connections to the destinations affected by the disconnection. Specifically, when the connection between source and one destination exhibits a throughput below a certain threshold, its connection should be dropped by the reliable FIFO broadcast and higher layers should be notified of this dropped connection. The chat application is responsible for reconciling lost data, since it can decide whether message reconciliation is necessary, and possibly prioritize the transmission of reconciliation data and real-time data thus controlling the latency of data delivery. We explain our strategy for handling disconnections and message reconciliation in section 5.1.3.

We therefore have two requirements on the FIFO broadcast. The first is that destinations should be decoupled; a destination should be removed from the broadcast when throughput to that destination is below a certain threshold. The second is that the FIFO broadcast should be network-friendly with proper flow-control mechanisms.

4.2.2 Implementation

Reliable FIFO broadcast between site-servers can be achieved in two ways. One option is to use a reliable FIFO multicast protocol running over IP multicast. The other possibility is to build reliable FIFO broadcast using reliable FIFO point-to-point connections between the source and every destination.

There are various reliable FIFO multicast protocols that use IP multicast and provide flow control mechanisms and the ability to adapt sending rates to handle links of varying qualities [13]. The advantage to using such a reliable FIFO multicast protocol is efficiency. IP multicast minimizes the amount of redundant data sent over a particular link. However, the service model presented by reliable multicast protocols does not meet our requirements. The multicast service model uses one destination address to represent the entire set of destinations; control data per destination is kept by the protocol and hidden from the application. The sender has limited control (perhaps none at all) over whether or not a particular destination remains in the set of destinations. We require that the sender have the ability to remove a destination from the reliable FIFO broadcast when the connection to that destination is poor; a reliable multicast protocol that hides per-state data from the application and prevents the sender from removing a destination from the group does not fit our requirements at all.

We decided to use reliable FIFO point-to-point connections between the source and every destination in order to achieve reliable FIFO broadcast. We use TCP to achieve reliable FIFO point-to-point connections. We take advantage of the fact that TCP has flow-control and is optimized for use over various types of links, including the links expected to be present in the MC2C environment. Maintaining point-to-point connections between all sources and destinations also allows us to explicitly keep track of the rate at which we are sending to each destination, which fits well with our requirements.

In addition, the gain in efficiency by using IP multicast as the underlying transport is only a marginal benefit. Our goal is to design a reliable FIFO broadcast between sites in the MC2C environment, so the number of destinations that our FIFO multicast protocol needs to handle is limited by the number of sites in the MC2C environment, which is at most ten. Opening point-to-point connections to each destination may necessarily result in some loss of efficiency due to redundant data being sent over shared links. However, since the number of destinations is at most 10, efficiency is reduced to 10 percent in the worst case. It is also unlikely that this worst-case loss in efficiency will be observed because the MC2C environment consists of planes as well as ground sites, and therefore many of the connections between sites are characterized by dedicated links; IP multicast does not increase the efficiency of link utilization when there are dedicated links between sources and destinations.

In Figure 9 we show the layered design of the chat prototype that exposes the implementation of reliable FIFO multicast.

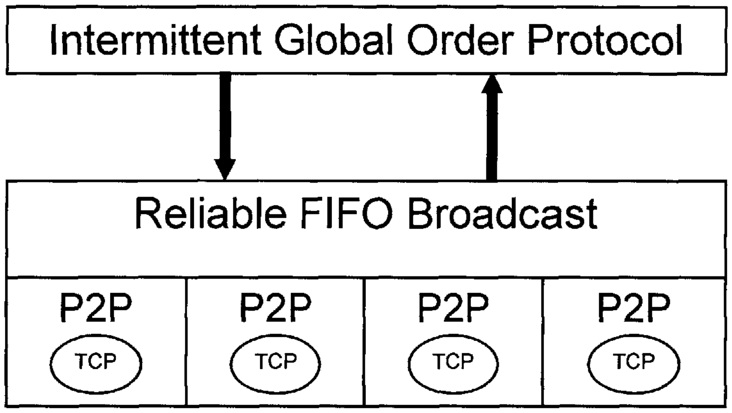


Figure 9: Reliable FIFO Broadcast. We implement Reliable FIFO multicast using TCP connections from source to every destination. When the throughput to a particular destination drops below a threshold, that connection is dropped from the broadcast, and the higher layer is notified. We use TCP since it is a reliable FIFO unicast connection; our protocol will work with any TCP-like service.

4.3 INTERMITTENT GLOBAL ORDER PROTOCOL

In this section, we describe the Intermittent Global Order Protocol, which implements the Intermittent Global Order property. This protocol is built on top of the reliable FIFO broadcast service. Intermittent global order is achieved by starting out with a traditional global order protocol and then modifying it to satisfy an intermittent ordering property that imposes global order whenever possible and relaxes this order when necessary in order to achieve bounded message delivery latencies.

4.3.1 Ordered Broadcast using Leader-based and All-to-all

The global order protocols that we use and modify are leader-based and all-to-all. We chose leader-based because it is a simple protocol to implement and because it is likely that at least one of the sites in the MC2C environment (presumably a site located on the ground) would be well-connected to the other sites. Such a well-connected site could be made the leader with the resulting performance benefits.

However, using the leader-based protocol, a site maintains communication with other sites only through the leader. If the link between a site and the leader site fails, then that site maintains only intra-site chat, and cannot chat with other sites. With the all-to-all protocol, if two sites are disconnected, they can still chat with other sites to which they remain connected, since there is direct communication between sites. In the case of the failure of one inter-site link, the all-to-all protocol can still continue communication with other sites connected via healthy inter-site links. One should note that in this case, if the Lamport delivery algorithm is used in order to achieve global order, then the delivery of messages will be delayed until the link comes back up, so message delivery latencies may grow unbounded. However, the key is that communication to other sites remains possible because all-to-all maintains direct connections between every site, and so messages from other sites continue to be received through the functioning direct connections. This gives us the option of modifying the delivery algorithm and sacrificing global order in order to continue message delivery. A disadvantage of all-to-all is that it maintains a point-to-point connection between every pair of sites; some of these point-to-point connections may have undesirable performance attributes, which may affect the protocol's performance.

We decided not to use a token ring-based protocol because the performance of the protocol degrades and its complexity increases when there are partitions. If there is a partition, then only one side of the partition contains the token-holder. The other side of the partition needs to elect a new token-holder, which complicates the protocol. Additionally, when the partitions merge, only one token-holder should exist, which amounts to another election. We felt that implementing such behaviors would be unnecessarily complicated and chose only to implement all-to-all and leader-based.

4.3.2 Intermittent Global Order Property

Recall the performance requirement from section 1.2.1 which calls for the Intermittent Global Order property: "… a reasonable performance goal is to maintain global order as long as the resulting message delivery latency is smaller than a certain defined threshold. This threshold is defined so as to facilitate normal human conversation using the chat application. If that threshold is in danger of being exceeded, then messages should be delivered with a weaker but well-understood ordering, and the global ordering should be re-achieved as soon as possible."

The weaker ordering properties on the message delivery implemented during times of disconnectivity are slightly different for the leader-based and the all-to-all protocols.

However, both algorithms have the same general idea. A key observation is that the desired behavior of global order with bounded message delivery latencies is only possible when certain critical connections are in place. For example, in the all-to-all algorithm, for any server *S*, the set of critical connections consists of the connections between *S* and every other chat server. If *S* is unable to communicate with a server *T*, then *S* can no longer achieve global order with the desired message delivery latencies. In particular, the Lamport delivery algorithm at *S* will wait for timestamp updates from *T*. Until those updates are received from *T*, *S* cannot update latest timestamp field of *T*. This in turn delays *S* from delivering any message with timestamp greater than the last timestamp received from *T*, so delivery latencies for these messages increase without bound until the faltering *S*-*T* connection is reestablished and timestamp updates from *T* resume. In the leader-based algorithm for any server *S*, the critical connection is the link between *S* and the leader; if this connection falters, then global order with bounded message delivery latencies cannot be maintained. When these critical connections are not in place, then there is imminent danger of message delivery latencies increasing beyond the threshold. At this point, an alternative, relaxed ordering that produces latencies within the threshold should be considered.

Therefore, the first step in implementing the desired intermittent ordering property is to define a performance threshold below which a connection is considered to have failed. We abstract the range of possible behaviors of the connection between two sites into a simple binary encoding: two sites can either be connected, or disconnected. A site considers itself to be disconnected from another site when it receives no data from the other site for a specified amount of time, or when the outgoing throughput to the other site is below a predefined threshold; the implementation details of these criteria are explained in section 5.1.3.

Once a disconnection is detected, the protocol attempts to 'weather' this disconnect for a short period known as the *disconnect-interval.* For this period of time, the disconnection is classified as a *short term disconnect.* During a short term disconnect, the protocol may attempt to maintain strong ordering properties on the message delivery despite the presence of a disconnection. If the previously disconnected link 'comes back up' before the disconnect-interval elapses, a reconciliation occurs in which each site asks for messages sent by the other site during the period of disconnection. If the disconnect duration exceeds the disconnect-interval, then the protocol classifies the disconnection as a long term disconnect. During a long term disconnection the protocol uses weaker ordering properties so that it can provide shorter, tolerable latencies of message delivery. Note that message reconciliation only occurs if a reconnection is made within the disconnect-interval; if reconnection occurs after this threshold, then no reconciliation takes place, and the protocol puts in place weaker ordering properties.

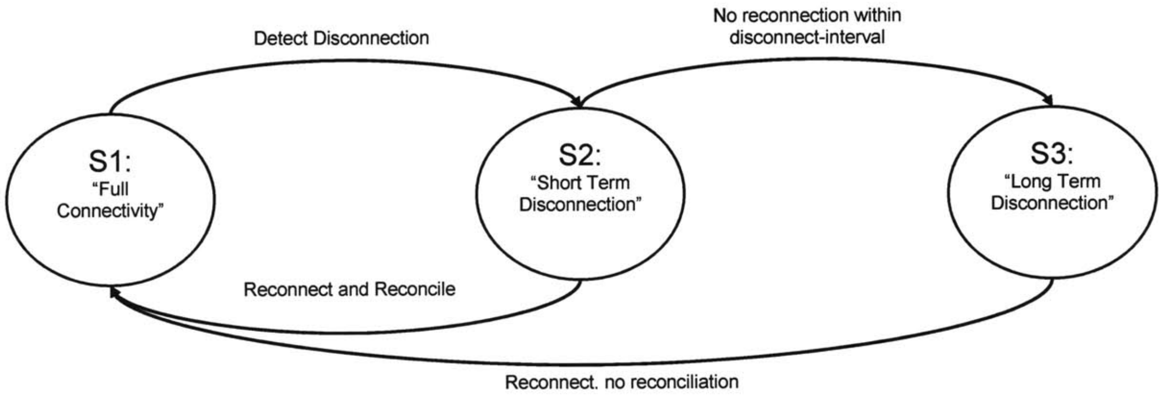

Detect Disconnection
No reconnection within disconnect-interval
S1: "Full Connectivity"
S2: "Short Term Disconnection"
S3: "Long Term Disconnection"
Reconnect and Reconcile
Reconnect. no reconciliation

Figure 10: A state machine representation of the Intermittent Global Order protocol

Figure 10 shows a state machine representation of the Intermittent Global Order protocol. State *S1* achieves global ordering with normal message delivery latencies; all critical links are functioning normally. The detection of a disconnection produces a state transition into *S2*. *S2* corresponds to a short term disconnection; the behavior of the protocol during short term disconnections is described in the next section. If a reconnection occurs before the disconnect-interval elapses, then the protocol transitions from *S2* to *S1*, and message reconciliation takes place. If no reconnection occurs within the disconnect-interval, the protocol transitions into *S3*, which is the long term disconnection state. In *S3*, the protocol exhibits relaxed ordering properties and returns to low latency message delivery.

4.3.3 Short term disconnects

The presence of a short term disconnection places the protocol in state *S2* of the IGO state machine. In this section, we describe the behavior of the protocol during the presence of a short term disconnect. We implement two different behaviors during the short term disconnect-interval. The first behavior delays the delivery of messages during the period of disconnection. This is 'wishful thinking' by the protocol; it delays message delivery in the hope that a reconnection will be made before the disconnect-interval is exceeded. If a reconnection occurs before the disconnect-interval is exceeded, then reconciliation occurs and the delayed messages arc 'mixed' with the reconciled messages and the entire set of messages is delivered in a global order. We refer to this method of handling short term disconnects as the *delay-delivery method.*

The second behavior delivers messages without delay (with weaker ordering properties) even when sites are disconnected. If reconnection occurs before the disconnect-interval is exceeded, then reconciliation occurs and the reconciled messages may be displayed out of order with respect to the other messages. In this behavior, a weaker-than-global order is achieved during the disconnection period. Sacrificing global order allows this method to achieve smaller message delivery latencies. We refer to this method of handling short term disconnects as the *hasten-delivery method.*

It is possible to implement both behaviors as modifications to both the leader-based protocol and the all-to-all protocol. We ended up implementing both the delay-delivery and hasten-delivery behaviors in the all-to-all protocol, and the hasten-delivery method as the default behavior in the leader-based protocol[9]. In the following sections, we illustrate the two methods with pictorial examples. In each example, there are two sites; the CAOC and the MC2A. They are connected by one inter-site link, and there are two clients located at each site.

4.3.4 Example: Short term disconnect using All-to-all protocol

Figure 11-Figure 14 illustrate the delay-delivery behavior in the all-to-all protocol.

[9] We initially wanted the chat application to dynamically detect the presence of a "flaky" connection; that is, a connection that flickers on and off at a frequency exceeding a certain threshold, and switch from the delay-delivery mode to the hasten-delivery mode in the presence of a flaky link. This is because continually delaying the message delivery until a reconnection occurs in the presence of a "flaky" link would lead to chronically delayed message delivery; switching to hasten-delivery mode in the presence of a flaky link should achieve lower latencies of message deliveries. However, due to time constraints, we did not incorporate this feature into the implementation.

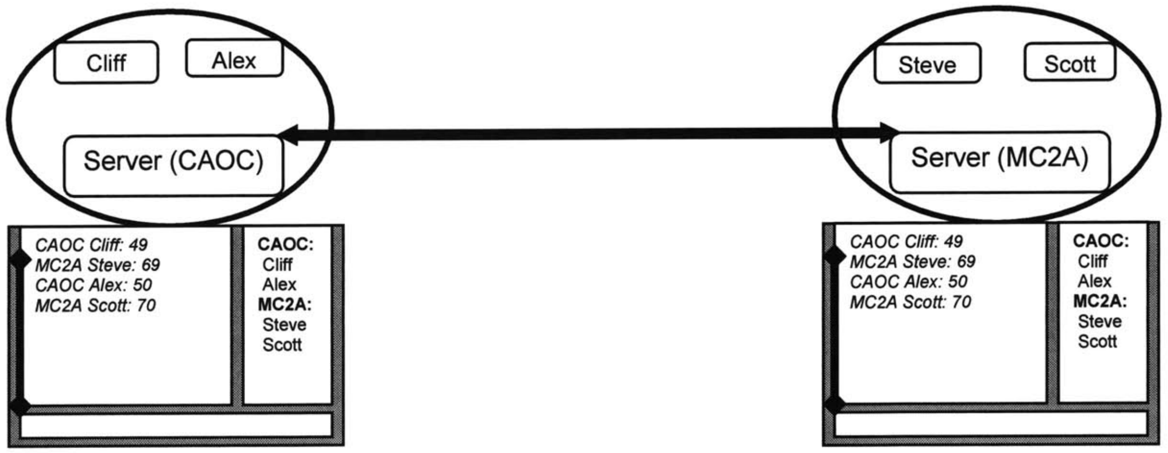


Figure 11: Short term disconnect using all-to-all: normal operation. Initially the two sites are well-connected. Below each site is a picture of the chat display seen by clients at each site. (Each client at a particular site sees the same chat display, since they are all served by the same site server). The box on the left-hand side of the chat display displays the conversation in the chat room, and the box on the right-hand side of the chat display displays the members currently logged into the chat room. Note that the message order is initially a global order; it is the same across both sites. For the purposes of this example, we assume that clients at each site send sequential numbers as messages; clients at CAOC have sent messages up to 50, and clients at MC2A have sent messages up to 70.

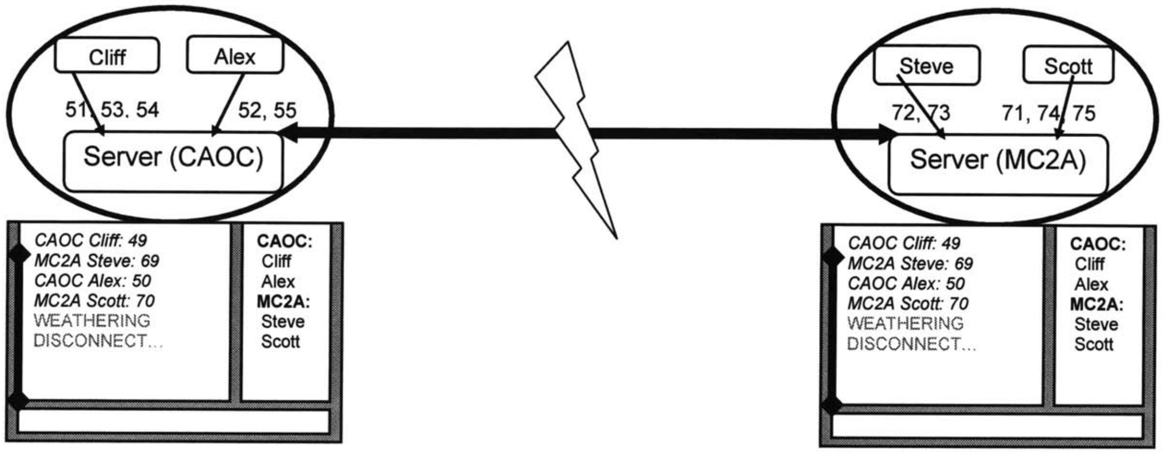


Figure 12: Short term disconnect using all-to-all: disconnection. A disconnect occurs between CAOC and MC2A. During the time of disconnection, the clients at CAOC send messages up to 55, and clients at MC2A send messages up to 75. These messages are received by their respective site servers, since they are sent over the reliable intra-site links, which are unaffected by the inter-site link outage. However, the servers do not display the messages; since they are each operating in the delay-delivery short term disconnect mode, they are both waiting to see if they can reconnect, reconcile messages, and maintain a global order on the message delivery.

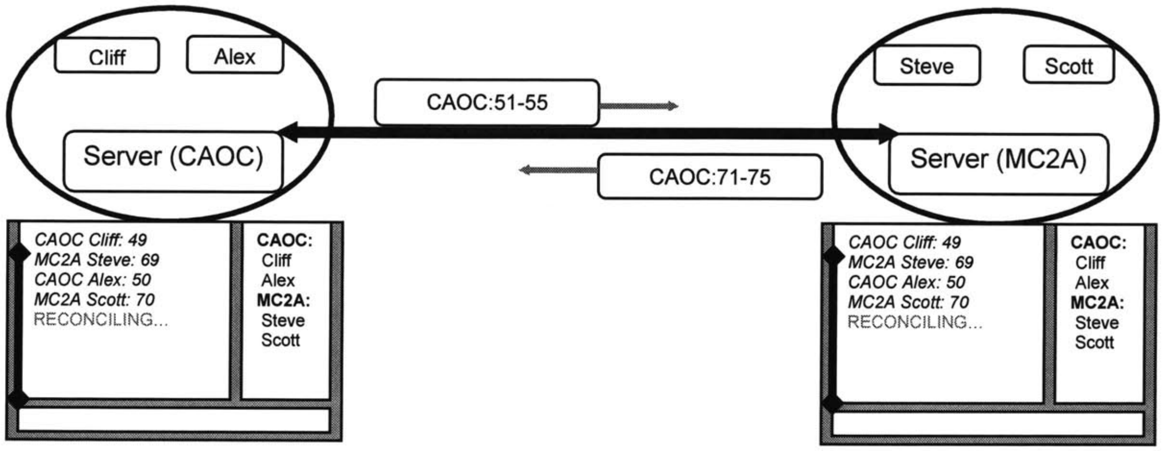


Figure 13: Short term disconnect using all-to-all: reconnection. A reconnection occurs while the connection is still a short term disconnect. The two sites reconcile messages sent during the period of disconnection; CAOC sends over messages 71-75, and MC2A sends over messages 71-75.

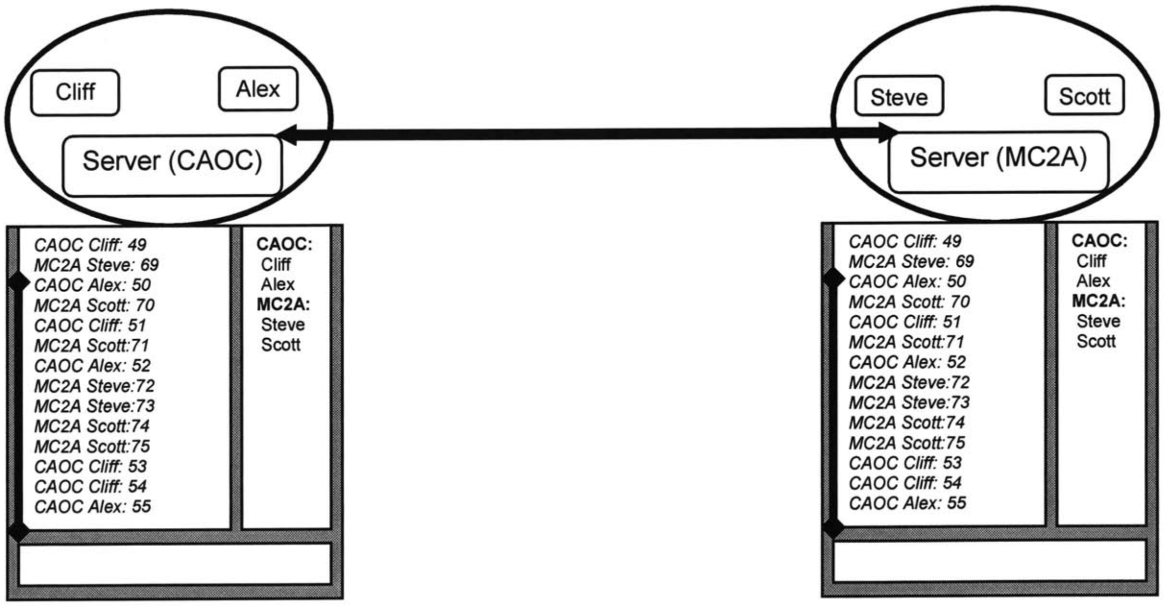


Figure 14: Short term disconnect using all-to-all: reconciliation. The messages are reconciled and displayed. Note that the two sites have the same order on the message display. This is because the reconciled messages are processed using the Lamport delivery algorithm at each site, which operates on the timestamps of the same set of messages at each site; it therefore produces the same message order at each site. Note that these messages are displayed with a greater latency since their display was delayed during the period of disconnection and reconciliation. These messages are displayed in a burst on the screen at the end of reconciliation.

4.3.5 Example: Short term disconnect using Leader-based protocol

Figure 15-Figure 18 illustrate the hasten-delivery behavior in the leader-based protocol.

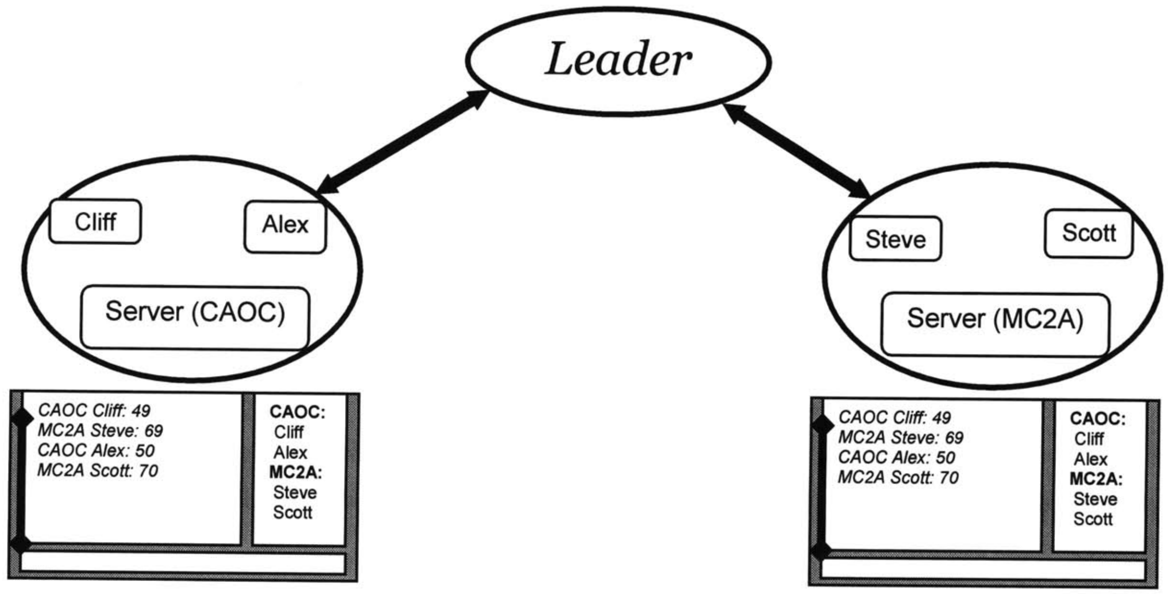


Figure 15: Short term disconnect using leader-based: normal operation. Initially both sites are well-connected to the leader. Note that initially the messages are displayed in the same order for clients at each site, and this message order is a global order; it is the same across both sites. For the purposes of this example, we assume that clients at each site send sequential numbers as messages; clients at CAOC have sent messages up to 50, and clients at MC2A have sent messages up to 70.

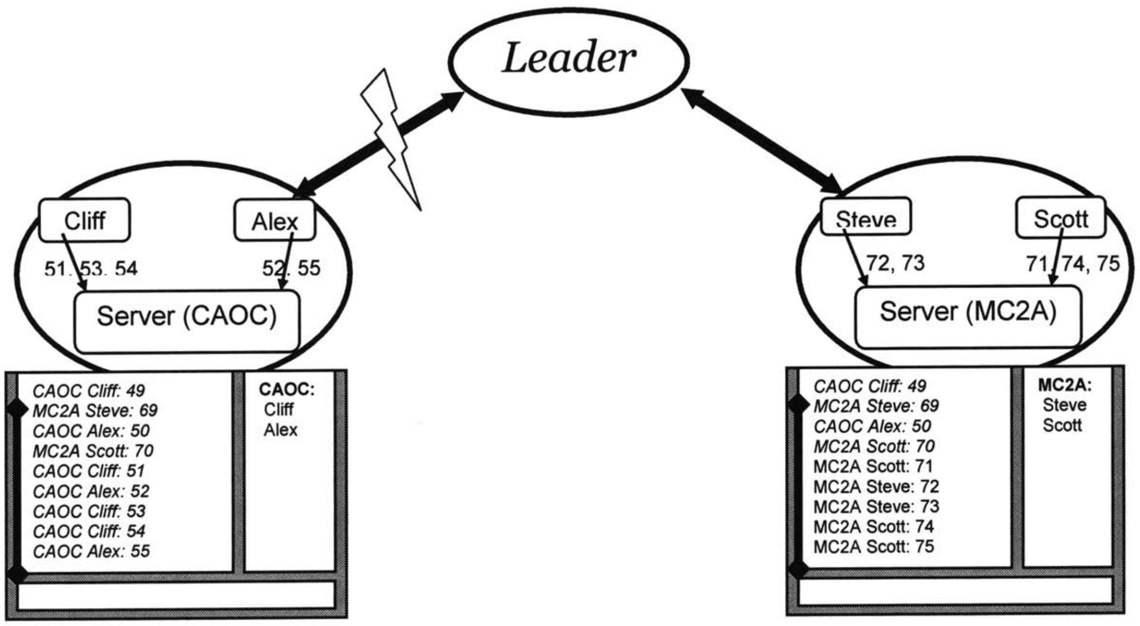
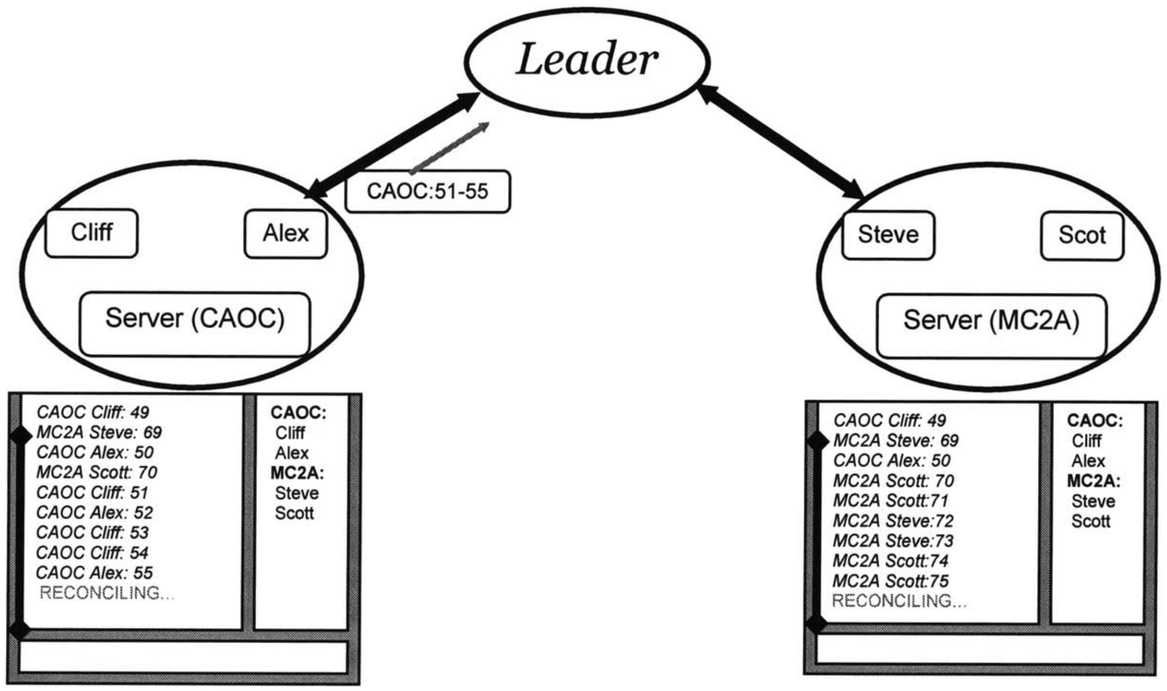


Figure 16: Short term disconnect using leader-based: disconnection. A disconnect occurs between CAOC and the leader. During the time of disconnection, the clients at CAOC send messages up to 55, and clients at MC2A send messages up to 75. CAOC receives these messages and displays them since it has lost its connection to the leader. MC2A receives these messages and relays them to the leader, which in turn relays them back to the MC2A for display. Note the membership display at each site; each site correctly informs its clients that they are chatting only with other clients located at its own site. Message delivery violates global order, since we are operating in hasten-delivery mode.



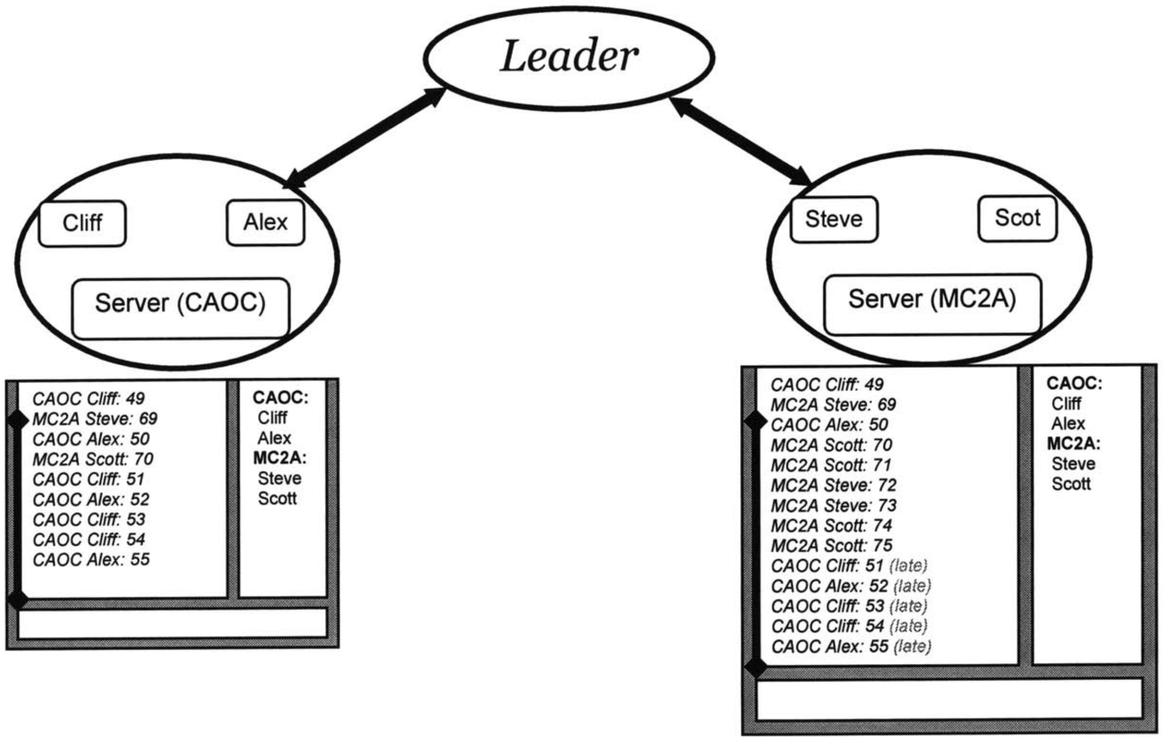
CAOC Cliff: 49
MC2A Steve: 69
CAOC Alex: 50
MC2A Scott: 70
CAOC Cliff: 51
CAOC Alex: 52
CAOC Cliff: 53
CAOC Cliff: 54
CAOC Alex: 55
RECONCILING...

CAOC:
Cliff
Alex
MC2A:
Steve
Scott

CAOC Cliff: 49
MC2A Steve: 69
CAOC Alex: 50
MC2A Scott: 70
MC2A Scott:71
MC2A Steve:72
MC2A Steve:73
MC2A Scott:74
MC2A Scott:75
RECONCILING...

CAOC:
Cliff
Alex
MC2A:
Steve
Scott

Figure 17: Short term disconnect using leader-based: reconnection. Reconnection occurs within the disconnect-interval, so reconciliation occurs. Note that only the site that is disconnected from the leader sends messages for reconciliation. We implement the protocol in this manner so that a site that is reconnecting to the leader is not deluged by messages from the possibly numerous other sites that remained connected to the leader. Note that the membership display at each site reflects the new connectivity.



```
CAOC Cliff: 49
MC2A Steve: 69
CAOC Alex: 50
MC2A Scott: 70
CAOC Cliff: 51
CAOC Alex: 52
CAOC Cliff: 53
CAOC Cliff: 54
CAOC Alex: 55
```

```
CAOC:
 Cliff
 Alex
MC2A:
 Steve
 Scott
```

```
CAOC Cliff: 49
MC2A Steve: 69
CAOC Alex: 50
MC2A Scott: 70
MC2A Scott: 71
MC2A Steve: 72
MC2A Steve: 73
MC2A Scott: 74
MC2A Scott: 75
CAOC Cliff: 51 (late)
CAOC Alex: 52 (late)
CAOC Cliff: 53 (late)
CAOC Cliff: 54 (late)
CAOC Alex: 55 (late)
```

```
CAOC:
 Cliff
 Alex
MC2A:
 Steve
 Scott
```

Figure 18: Short term disconnect using leader-based: reconciliation. Reconciled messages are displayed out of order at the site that remained connected to the leader. The messages are displayed with the caveat that they are late.

4.3.6 Ordering properties

4.3.6.1 Ordering properties during short term disconnections

When the delay-delivery behavior is used and a short term disconnection occurs, a global order on the message delivery is trivially maintained, since no messages are delivered while the protocol waits to see if a reconnection is possible during the disconnect-interval. If a reconnection takes place, global order is maintained; if not, then the disconnection is classified as a long term disconnection, and the delivery order assumes the ordering properties guaranteed for long term disconnects.

If the hasten-delivery behavior is used, then the delivery order assumes the ordering properties that are guaranteed for long term disconnects even during the time that the disconnection is classified as a short term disconnection; this is because message delivery is not delayed in the hasten-delivery method.

4.3.6.2 Ordering properties during long term disconnections

Delivery order during long term disconnections assumes slightly different properties in the all-to-all and leader-based protocols. The ordering properties are also different depending on whether or not we admit the assumption of transitive connectivity on the underlying network.

We first describe the ordering properties in the leader-based protocol, as it is not affected by the transitive connectivity assumption. In the leader-based protocol, the set of sites that remain connected to the leader have the same delivery order (that is the order defined by the leader). Each site that is disconnected from the leader maintains only intra-site chat, and so maintains only an intra-site order on the message delivery.

In the all-to-all protocol, if we do not assume transitive connectivity, then the ordering property is that pairs of sites that are connected impose the same ordering on each other's messages. The ordering property assuming transitive connectivity is a composition of this basic property. Recall that under transitive connectivity, a disconnection that does not result in a partition will be discovered and hidden from the application by the routing layer, and the application will be aware only of complete partitions. Therefore, a disconnection visible by the application is necessarily a partition. The ordering property maintained by all-to-all is that messages are delivered in the same order in each partition. This property can be 'derived' from the basic pair-wise ordering property of the all-to-all algorithm; all sites in a partition are connected to each other and to no one else, so applying the pair-wise ordering property to every pair, we get that the ordering must be the same at every site in a partition.

The ordering property guaranteed by the all-to-all algorithm is achieved by modifying the Lamport delivery algorithm. Suppose we are running the Lamport delivery algorithm at site *A*, and it is connected to a site *B*. Recall that in the Lamport delivery algorithm, each process keeps a mapping that maps every process to the last timestamp received from that process. A message is deliverable when the last timestamp of every process in the mapping has advanced past the timestamp of that message. In the traditional Lamport delivery algorithm, every process in the broadcast is kept in the mapping at al times. We modify the traditional Lamport delivery algorithm as follows. As long as *A* is connected to *B*, or *B* is classified as a short term disconnection and *A* is running the delay-delivery method of handling short term disconnections, then *B* remains in the process $\rightarrow$ timestamp mapping at

A. Once the disconnection is classified as a long term disconnection, then *A* removes B from the mapping. The result is that messages that are considered for delivery no longer have to wait for timestamp updates from disconnected sites. As long as *A* and *B* are connected, then they impose the same ordering on each other's messages. If transitive connectivity holds, then during a disconnection, every site that is in *A*'s partition will remove *B* from the mapping at approximately the same time. The result is that all sites in *A*'s partition contain the same sites in the site → timestamp mapping. Therefore, every site in *A*'s partition achieves the same ordering. The same reasoning applies to *B*'s partition.

These properties are illustrated in Figure 19 and Figure 20.

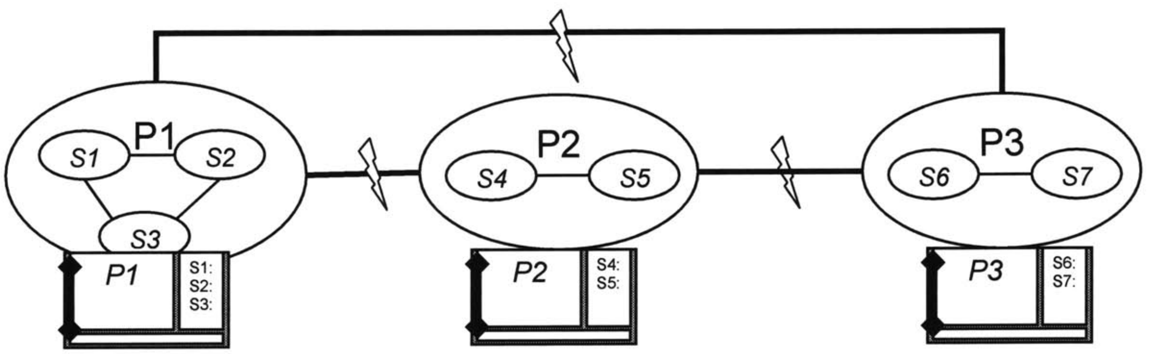


Figure 19: Long term ordering property for the all-to-all protocol, assuming transitive connectivity holds. There are three partitions; P1, P2, and P3. Every site within a particular partition has the same ordering on the message delivery. The membership display at each site reflects the partition to which it belongs.

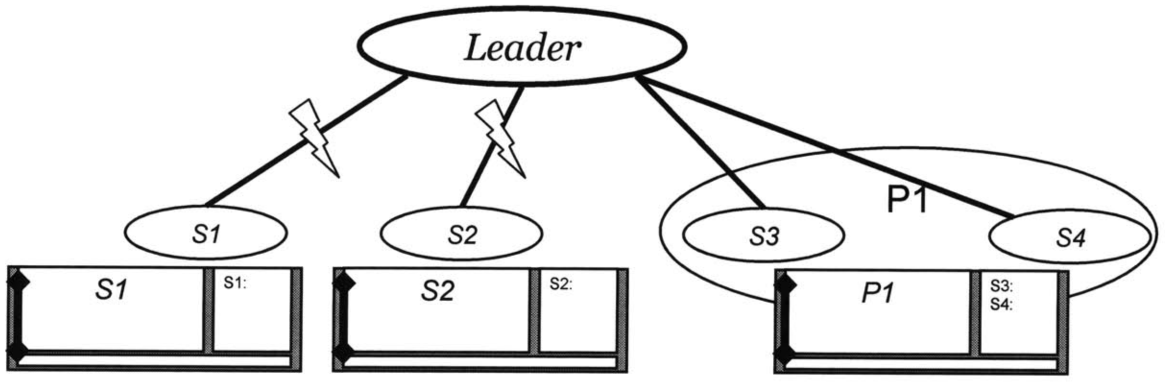


Figure 20: Long term ordering in the leader-based protocol. Sites that lose their connection to the leader maintain only intra-site chat, and cannot chat with other sites. All sites that are connected to the leader form a separate partition and see the same ordering and have the same view of the membership.

There is also an intuitive property that is maintained by the GUI of each chat client. Recall the example in Figure 16; the site CAOC is disconnected from the leader, while the site MC2A remains connected to the leader. Each site continues to display messages; CAOC maintains intra-site chat since it is disconnected from the leader, whereas MC2A maintains chat with all sites that remain connected to the leader (in this case, only itself). Note that the membership display at CAOC displays only clients at CAOC, and the membership display at MC2A displays clients in sites that remain connected to the leader (the leader itself updates connected sites with the set of sites to which it is connected). This illustrates that the membership display of a particular client always displays the universe of sites that are seen, and hence incorporated into the ordering, by that site's chat server. Assuming transitive connectivity, the membership display of a client lists the membership of the partition to which the client belongs. Message delivery order is the same in each partition. Therefore, each client can assume the following ordering property on the conversation display: each new message that appears on the conversation display is seen in the same order by every client that is visible on its own membership display.

If transitive connectivity does not hold, the same property relating ordering to the membership display holds for the leader, since it is not affected by the transitive connectivity assumption. However in this case, in the all-to-all protocol, the user cannot assume that the list of sites seen in the membership display is a partition; the user can only rely on the pair-wise ordering property on the message display.

5 Implementation

The chat prototype is written in Java™ (version 1.4); communication between servers and from client to server is through serialized Java™ objects that are sent over the network. In the following sections we describe the implementation of the chat server and client.

5.1 Chat Server

The system architecture of the server closely follows the Single Process Event Driven (SPED) architecture model [14], in which a single event-driven process concurrently handles all events and associated data processing. There are three important modules comprising the server:

- `NbServer`: a module that handles network input operations using an asynchronous I/O library. It handles all read operations, converting raw bytes read from the network into Java™ objects. Events, such as the receipt of an object or the closing of a connection, are sent to the appropriate module to be handled.
- `ChatServer`: The `ChatServer` is the interface between the chat clients and the Intermittent Global Order Protocol. The `ChatServer` hands chat messages from its clients to the `BcastService`, which sends these messages to other sites. It receives ordered messages from the `BcastService` and delivers the ordered chat messages back to clients for display.
- `BcastService`: Implements the Intermittent Global Order protocol and reliable FIFO broadcast.

The following is a basic flow of control, from the point of receiving a message from the chat client to delivering ordered messages back to the chat client for display. Chat clients log in to the `ChatServer` and send chat messages over this connection. The `ChatServer` receives these messages from the `NbServer` (which handles all incoming I/O) and hands these messages, along with updates to the membership of the global chat room, over to the `BcastService` for sending. The `BcastService` exchanges its site's messages with `BcastService` modules located at other sites. Received messages are ordered according to the Intermittent Global Order protocol and delivered to the `ChatServer`. The `ChatServer` in turn delivers these messages to its clients.

The basic system architecture is outlined in Figure 21.

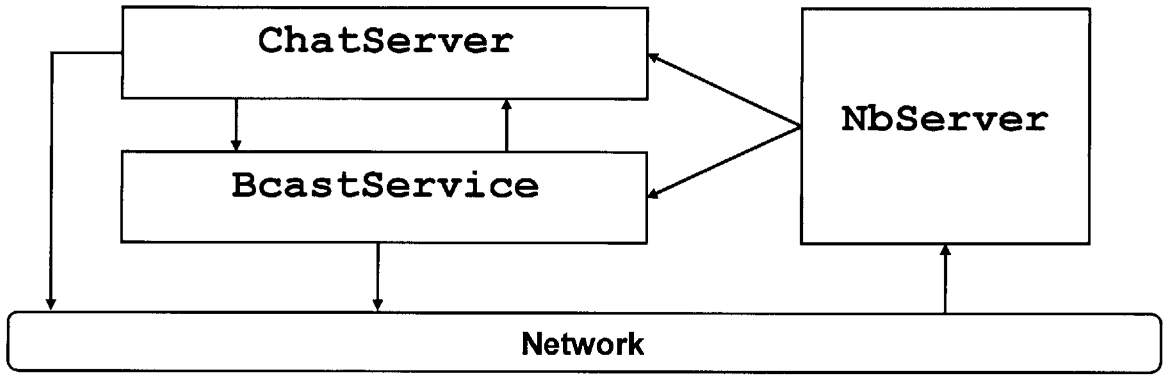


Figure 21: The basic system architecture of the chat server. In the figure, the network represents both inter-site and intra-site links. The `NbServer` handles all read operations and reports events to the `ChatServer` or the `BcastService`. Chat messages received on intra-site links from clients are reported to the `ChatServer`, which hands them over to the `BcastService`. The `BcastService` sends messages over inter-site links to be received by other `BcastService` objects. Messages received on inter-site links from other sites are handed over to the `BcastService`, which orders them and then hands them up to the `ChatServer`, which delivers the ordered messages over intra-site links back to clients for display.

5.1.1 NbServer

The SPED architecture of the chat server is made possible by the `NbServer`[10] module, which was one of the first modules written as part of this thesis. This module was written using the Java™ NIO library, which is new to Java™ 1.4. It uses non-blocking system calls to perform asynchronous I/O through calls to a Java™ `Selector` object[11].

The `NbServer` module works as follows. Clients of the `NbServer` register interest on network I/O operations with the `NbServer`. Clients may either register their interest in connections made on a specified port (`registerForAccept`), or in objects received on a particular network connection (`registerForRead`).

[10] NbServer stands for Non-Blocking Server.

[11] The `java.nio.Selector` multiplexes I/O events, providing functionality that is equivalent to a BSD UNIX `select` operation. For example, an application can register interest in a read operation on a particular network connection with the `Selector`. A call to `Selector.select()` returns the set of registered I/O events that are completed and ready to be handled.

Completed network events are reported via callback functions to a class satisfying the `ServerHandler` interface, which is in charge of handling the event. There are three functions in the `ServerHandler` interface.

- `handleAccept`: Reports a connection that is accepted on a port that was registered by a `registerForAccept` function call.
- `handleObject`: Reports an object that is received on a connection registered by a `registerForRead` function call.
- `handleEOF`: Reports an End-Of-File (EOF) that is received on a connection registered by a `registerForRead` function call.

The `NbServer` module is used in our chat application as follows. The task of listening for client connections on a specified port is handled by the `NbServer`; completed connections are reported to the `ChatServer`. The `BcastService` assigns the task of listening for connections made from `BcastService` modules located at other sites to the `NbServer`, and completed connections are reported to the `BcastService`. Additionally, the `ChatServer` and the `BcastService` delegate the task of reading data objects from their connections (to clients and to other `BcastService` modules, respectively) to the `NbServer`. Received objects and closed connections are reported to the appropriate module, which handle these events accordingly.

The SPED architecture, which is enabled by the `NbServer` module, allows the chat server to overlap network I/O functions with application level functions performed by the `BcastService` and `ChatServer` modules, all in a single thread of control. Overlapping network I/O functions with application level functions could also be achieved in a multi-threaded architecture, in which a thread is assigned to read data from every network connection. However, the cost of context-switching becomes prohibitive as the number of threads increases; if our server resides on a site with up to a hundred clients, a multi-threaded architecture might suffer from performance problems. Additionally, the use of multiple threads introduces synchronization issues as multiple threads may contest for shared data objects; managing this synchronization may produce potential performance bottlenecks.

We should note that the only function required to achieve a SPED architecture is a `select` system call; in that case the "event" reported to the application is the availability of bytes on a particular network connection. Wrapping the NbServer around a Selector increases the granularity of the "events" reported to our application from the level of bytes to the level of objects. Interested modules are informed only of the receipt of entire objects using callback functions; the partial receipt of an object over a network connection is hidden from the application[12]. This greatly simplifies the task of writing an object-oriented network protocol implemented by a high-performance SPED architecture.

5.1.2 ChatServer

The ChatServer interfaces between the chat clients and the Intermittent Global Order Protocol. When the ChatServer starts up, it initializes the BcastService which establishes connections to other sites and informs the ChatServer of these connections. The ChatServer maintains knowledge of its connectivity to other sites through updates passed to it by the BcastService.

The ChatService also listens on a known port for client connections; clients log into the ChatServer and establish TCP connections over which they send messages to the ChatServer. The ChatServer receives messages sent by clients and passes them to the BcastService. When a client logs into the chat server, the ChatServer module passes a membership update to the BcastService. Membership updates are treated as chat messages by the BcastService. Connectivity updates from the BcastService are also translated into membership updates by the ChatServer, which sends them to its clients so that the appropriate membership is displayed on the screen.

Figure 22 summarizes the logical relationships and the types of messages passed between clients, the ChatServer, and the BcastService.

[12] The Java™ I/O class java.io.ObjectInputStream achieves the same function when wrapped around the raw byte stream. However, the java.io.ObjectInputStream.readObject() function call, which returns an entire object, is a blocking I/O operation. Blocking I/O operations are by nature synchronous; they cannot be registered for an asynchronous select operation.

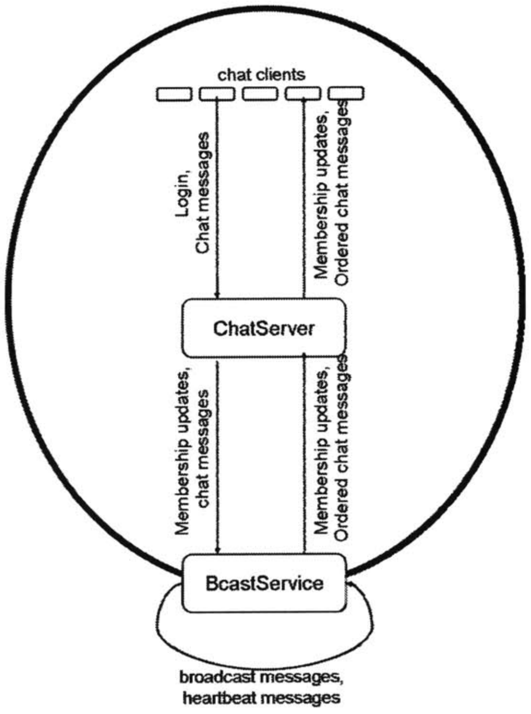

chat clients
Login, Chat messages
Membership updates, Ordered chat messages
ChatServer
Membership updates, chat messages
Membership updates, Ordered chat messages
BcastService
broadcast messages, heartbeat messages

Figure 22: A summary of the logical relationships between chat clients, the `BcastService`, and the `ChatServer`. Note that the BcastService module communicates with other BcastService modules using broadcast messages, which encapsulate chat messages and membership updates.

5.1.3 BcastService

The `BcastService` implements the Intermittent Global Order protocol and contains the implementation of inter-site reliable FIFO broadcast. We have two different implementations of the `BcastService`: one that uses the all-to-all protocol and one that uses the leader-based protocol. Both versions of the `BcastService` assign sequence numbers to messages and keep message histories for the purposes of reconciliation. Sequence numbers are distinct from Lamport timestamps and are used to request messages during reconciliation.

The `ChatService` starts up the `BcastService`; depending on the mode in which it is started, the `BcastService` then establishes the connections necessary in order to broadcast messages. In the leader-based case, it establishes a connection to the leader, which is at a known IP address and port. In the all-to-all case, it establishes connections to `BcastService` modules running at other sites. In all-to-all, the `BcastService` is given the set of sites to which it has to connect during initialization; this set of sites is static. The

BcastService module probes for connections to sites with higher IP addresses, and listens for connections from sites with lower IP addresses; this is simpler than having both sites probe each other and potentially establish TCP connections at approximately the same time.

When the BcastService is handed a message by the ChatServer for sending, it first assigns the message a sequence number based on the value of the local sequence number counter and then increments the counter. In all-to-all mode, if it is has inter-site connectivity, it performs a reliable FIFO broadcast of the message to other sites. In leader-based mode, it performs a reliable FIFO send to the leader. If those critical inter-site connections are not available, then the message is ordered and delivered back to its own clients. If the BcastService is running in all-to-all mode, messages are assigned Lamport timestamps in addition to sequence numbers for the purposes of the Lamport delivery algorithm.

5.1.3.1 Reliable FIFO Broadcast

Reliable FIFO multicast is implemented in the BcastService module as a single TCP connection to every site. Broadcasting a message consists of serializing the message object and sending it over each TCP connection.

We use non-blocking sockets and set the local send socket buffer size for each connection to be as large as the maximum "burst" of data that is expected from the chat clients during a short term disconnect. A connection is classified as "off" if a write to the corresponding TCP socket fails because the local socket buffer cannot handle the additional load. Under normal operation, the rate of chat messages should not exceed the bandwidth of the underlying link, so the send socket buffers should not fill up. However, if the underlying link is "down", then the socket buffer will fill up; a failed write is therefore a reliable indication of a problem in the underlying link, and it is reasonable to assume that the link is down and to close the connection.

We want to set the send socket buffer sizes to handle the maximum burst expected, so that failed writes accurately signal a broken link. The maximum expected burst of data is the amount of data sent by a site's clients during a short term disconnection; during reconciliation this data will be sent in one burst upon reconnection. The disconnect-interval is 15 seconds, we conservatively assume that the set of clients at a site send at an aggregate rate of 2 messages per second, and each message is 1 kilobyte. The largest burst size is therefore 15*2*1 = 30 kilobytes. Therefore, the send socket buffer size is set to be at least 30 kilobytes.

5.1.3.2 Heartbeat Protocol

On the receive side, the `BcastService` module implements a heartbeat protocol in order to detect disconnections. In our experience, disconnections are almost always detected with the heartbeat protocol before detection by the sending side, since our estimation of the socket buffer size required to handle the largest burst is conservative. The heartbeat protocol keeps track of the most recent time when a message from each particular site is received. It sends out periodic heartbeats to connected sites every *heartbeat-interval* seconds; the heartbeat-interval is set to be two seconds in our implementation. If a site is not heard from for two consecutive heartbeat-intervals, then in the third heartbeat-interval, the TCP connection to that site is closed, and the disconnection is classified as a short term disconnection.

Once the connection to a site is closed, the `BcastService` dedicates a separate thread that probes to reestablish a connection to the disconnected site. Here again, the site with the lower IP address probes the site with the higher IP address, so that only one connection is established. If a reconnection occurs to that site within (*disconnect-interval – heartbeat-interval)* seconds (that is, before the disconnection is classified as a long term disconnection), then reconciliation occurs over the newly established TCP connection and the sites exchange the messages that they missed.

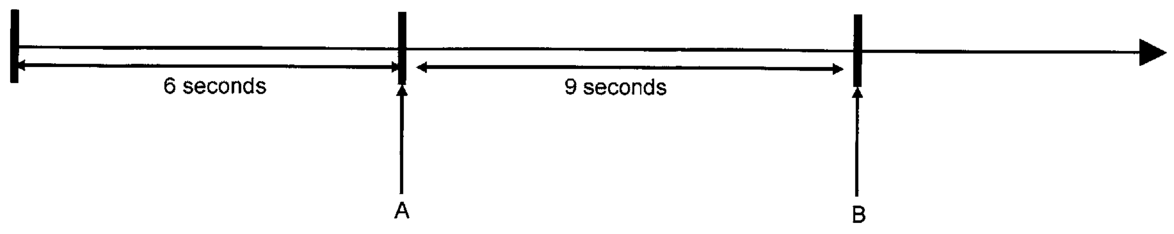


Figure 23: The disconnect timeline. If no data is received by point A, which corresponds to 3 heartbeat-intervals (6 seconds), then the TCP connection is closed, and the `BcastService` classifies this connection as a short term disconnection. Point B marks the disconnect-interval. If a reconnection occurs between A and B, then reconciliation occurs. If no reconnection is made by point B, then the connection is classified as a long term disconnection. Reconnections made after point B do not result in reconciliation.

5.1.3.3 Reconciliation

Reconciliation occurs only when a reconnection is made before the disconnection is classified as a long term disconnection. This is because as the length of a disconnection grows, the number messages to be reconciled increases. Delivering all of these messages to the user may inundate the user with irrelevant messages and make the chat output unreadable[13].

When a disconnection is detected either by the sending side or the heartbeat protocol, the TCP connection is closed, so data buffered by this connection is lost. If sites reconnect quickly enough to reconcile, the reconnected sites do not know where in the sequence of messages to restart sending to the other side, due the loss of state caused by the broken TCP connection. The purpose of reconciliation is to mask the underlying disconnection from the chat server, and to make sequence of messages between the reconnected sites reliable and FIFO. Reconciliation therefore takes the form of a request and reply; each site requests the other site to send messages starting at the last sequence number that it missed from the other site; this set of messages sent in bulk as a reply.

Currently the implementation keeps message histories for all sites. The message histories for each site are stored in FIFO order in an object in memory. This is egregious, since it prevents the chat servers for running too long because of memory consumption. One possible fix is to flush old messages to disk. However, a site may only want to keep message

[13] We observed from the tests that even reconciliation after short term disconnections produces rates of message delivery that could potentially exceed the user's ability to process messages; this is discussed in section 6.3.5.2

histories for the purposes of reconciliation. In this case, it can remove from its history messages that it knows have been received by every other site.

5.1.3.4 Reconciliation after long term disconnections

Originally, we planned to reconcile messages after long term disconnections, so that each site could keep a FIFO record of messages sent by every other site during a session. The fact that the implementation contains message histories for every site is an artifact of this original idea. Keeping message histories from every site is unnecessary given the reconciliation strategy that we implemented. The current strategy requires only that a site keep a history of recently sent messages.

Long term message reconciliation was planned to occur as a background thread that requested messages to fill gaps in its FIFO per-site message histories. This proved to be too difficult to implement during the course of this thesis. We did, however, think through a possible implementation of long term message reconciliation and per-site message histories. The key is to realize that long term reconciliation messages are lower priority messages, compared to real-time chat messages and short term reconciliation messages. In order to maintain the priority of real-time and short term reconciliation messages, it would be necessary to implement a priority-based sending scheme that maintains different queues for messages with different priorities. The long term message reconciliation thread would deposit reconciliation messages in lower priority queues in a priority-based sender, which would schedule and send them appropriately. Using this approach, long term message reconciliation would not consume valuable bandwidth during times of need.

5.1.3.5 Limitations with the current implementation of short term message reconciliation

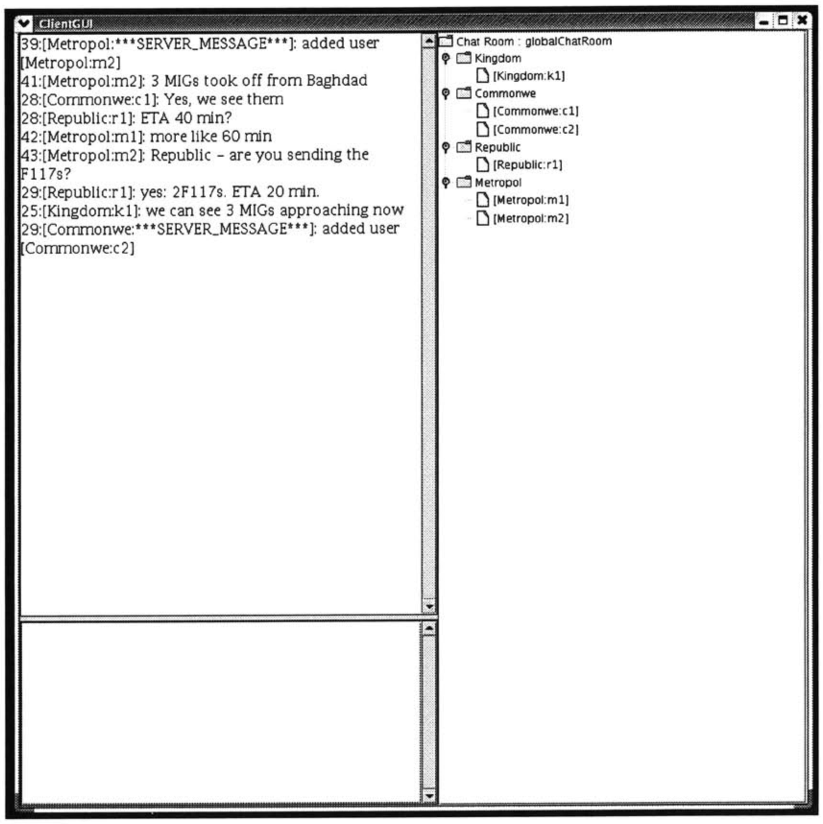
The current implementation of short term disconnects and reconciliation is unnecessarily complicated. Consider the disconnect timeline presented in Figure 23. At point *A*, a disconnection is detected; the connection is classified as a short term disconnection and the Intermittent Global Order protocol implements the delay-delivery or hasten-delivery behavior to handle the disconnection. At this point, the TCP connection is closed; if the link comes back and a reconnection occurs before point *B*, then the reconciliation protocol does a request-reply and the FIFO stream is maintained. Closing the TCP connection discards messages buffered by TCP connection; this loss of state is what necessitates the request-reply protocol upon reconnection. The closing of the TCP connection and the request-reply protocol upon reconnection is an unnecessarily complicated way to achieve a reliable FIFO stream between two sites despite a short term disconnection.

The same effect can be achieved without closing the TCP connection at point A. In this case, the heartbeat protocol continues to listen for data on the connection after point A. If data is received before point B, then this is equivalent to a reconnection. The TCP stream ensures that the resulting stream is FIFO, and no request-reply reconciliation protocol is needed; reconciliation is automatically achieved since the TCP stream is FIFO and reliable. The key is to report the detection of a short term disconnection at point A (or even sooner), so that the Intermittent Global Order protocol can act on the information and implement its short term disconnect handler; in addition, the TCP connection must be kept open so that upon reconnection, a reliable FIFO stream is maintained. If no reconnection is made by point B, then the TCP stream can be closed. Reconnection after point B does not result in reconciliation, so the application-level request-reply protocol becomes completely unnecessary.

We implemented the more complicated version because the prototype initially performed reconciliation after both short term and long term disconnections. Closing the TCP connection at point A as soon as the disconnection was detected allowed us to handle both short term and long term disconnections and reconciliations in a uniform manner. Later when we decided not to perform reconciliation after long term disconnections, the existing code for handling reconciliation was used to handle short term disconnections, resulting in the more complicated protocol.

5.2 Chat Client

The chat client is a "thin" client that allows a user to enter messages and displays messages received from the chat server. We have implemented the client both with and without a GUI, for the purposes of both demonstration and testing. The client can be configured to be used with an automated "bot" that sends messages at different rates for the purposes of automated testing. Figure 24 shows a screenshot of the client GUI and explains the three frames.



```
39:[Metropol:***SERVER_MESSAGE***]: added user
[Metropol:m2]
41:[Metropol:m2]: 3 MIGs took off from Baghdad
28:[Commonwe:c1]: Yes, we see them
28:[Republic:r1]: ETA 40 min?
42:[Metropol:m1]: more like 60 min
43:[Metropol:m2]: Republic - are you sending the
F117s?
29:[Republic:r1]: yes: 2F117s. ETA 20 min.
25:[Kingdom:k1]: we can see 3 MIGs approaching now
29:[Commonwe:***SERVER_MESSAGE***]: added user
[Commonwe:c2]
```

```
Chat Room : globalChatRoom
  Kingdom
    [Kingdom:k1]
  Commonwe
    [Commonwe:c1]
    [Commonwe:c2]
  Republic
    [Republic:r1]
  Metropol
    [Metropol:m1]
    [Metropol:m2]
```

Figure 24: The Client GUI. The upper left hand frame is the conversation display. Each message is displayed in the format [Site:username]:<message text>. The lower left hand frame is for text input. The right hand frame displays the members that are currently logged into the chat room. The membership display is site-aware; users are listed under the site to which they belong. When the connection to a site is lost, it is removed from the membership display, along with the set of users in that site.

The client communicates messages to the server through a TCP connection that is established when the client logs on to the server. Communication from the server to clients must be reliable and FIFO. This can be achieved in two ways. One approach is for the server to simply send messages to each client through the TCP connection that it has to that client. This is robust, but inefficient; clients will, in general, reside on a robust LAN along with the server. There may be up to a hundred clients, so each message is duplicated up to a hundred times on the LAN. The second approach is to use multicast. We implemented a simple reliable FIFO multicast for a LAN environment that uses UDP multicast in order to broadcast messages to each client. UDP multicast is unreliable and has no ordering guarantees, but a LAN is generally very robust and reliable, so missed or reordered messages are rare.

Nevertheless, we include a simple mechanism for handling missing or reordered messages; our protocol sends NACKs for missed or reordered messages to the server over the reliable TCP connection; the server then retransmits this data back to the client over the same TCP connection.

6 Testing and Evaluation

We performed experiments on a test-bed that was created as part of this thesis. We ran a number of experiments that tested the performance of the chat server under short and long term disconnects, using both the leader-based and all-to-all algorithms. In each experiment, we had at least one link that was disconnected for between 25-50 percent of the time, and we tested both the delay-delivery and the hasten-delivery methods for tolerating short term disconnections.

The chat servers perform as expected. Messages are delivered with low latencies (<1 second) in a global order when all sites are connected. Messages are also delivered with low latencies (< 1 second) when there are long term disconnections, or when tolerating short term disconnects in the hasten-delivery mode; however, the ordering is not global and is as specified in section 4.3.6. Message latencies are higher but still bounded while detecting link outages; latencies drop down to low levels again once the disconnection is classified as a long term disconnect, or weathered according to the method in place. Reconciliation results in higher rates of message delivery, and delivers messages with higher latencies.

The general result is that our prototype ensures chat availability regardless of the state of inter-site connectivity. Message delivery latency is bounded; latencies are low as long as the network is stable, but may increase while disconnections are detected until they are classified or weathered. A well-defined ordering is always maintained on the delivery of chat messages.

6.1 Test-bed Architecture

We implemented a test-bed with four computers. Each computer is a site and runs one chat server; chat clients are run as separate processes on the same computer as their corresponding chat server. The sites are named Metropol, Commonwe[14], Kingdom, and Republic. Data between chat servers is sent through a router, running the Linux operating

[14] Metropol is short for Metropolis, and Commonwe is short for Commonwealth. The Keyboard-Video-Mouse Switch (KVM) used for display and input to the servers in our test-bed setup limited the names of the servers to be at most 8 characters, thus the shortened names. We refer to the servers by these shortened names for the remainder of the thesis.

system that uses software[15] to control the bandwidth and availability of the connection between any two servers. Connectivity and bandwidth can be set independently for both directions of a link. We wrote scripts that support simple configuration files that control bandwidth and connectivity as a function of time during an experiment.

An important limitation of the test-bed is that we currently do not explicitly control the latencies of the test-bed links. We could vary message sizes in order to simulate latency effects; although this is possible in the chat client, we don't vary this parameter in our tests. Future improvements to the test-bed should implement variable latencies for each point-to-point link.

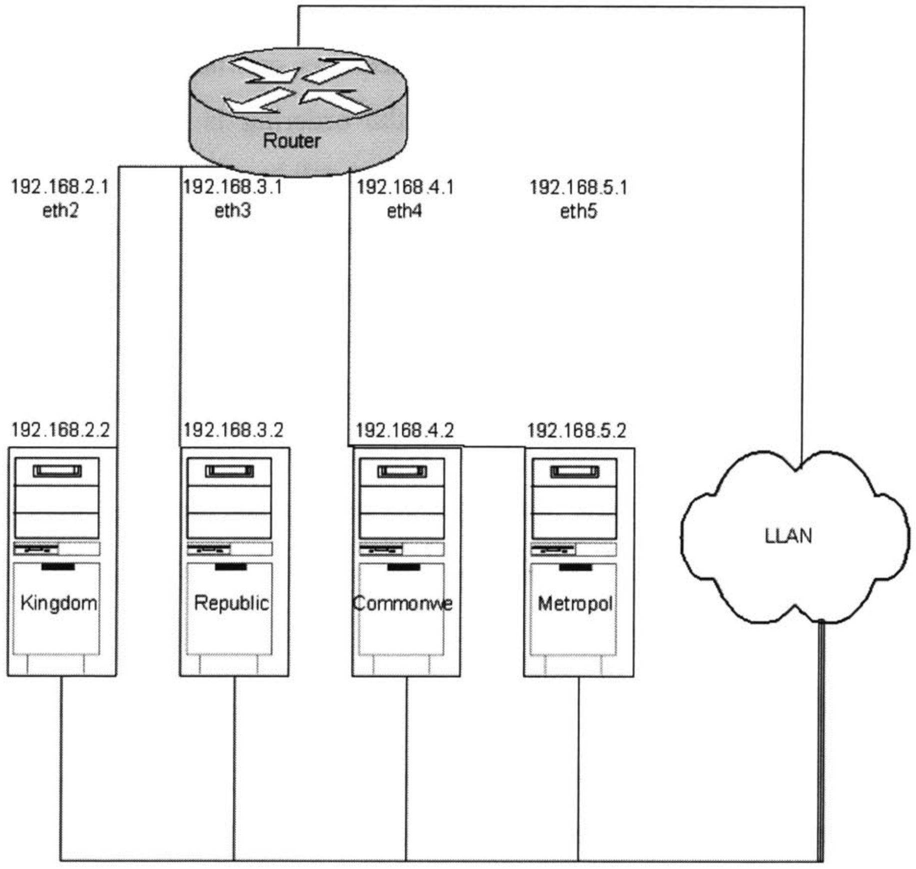


Figure 25: Test-bed Architecture. Data sent between chat servers (Kingdom, Republic, Commonwe, and Metropol), is configured to be sent through the router. Software at the router controls link availability and bandwidth. Experiment controls are sent through the LLAN network and are not affected by the connectivity/bandwidth settings of the experiment.

[15] We used iptables (see www.netfilter.org) in order to bring links up and down, and we used the Linux Advanced Routing and Traffic Control package (see www.lartc.org) to control bandwidth.

6.2 Scenarios

We ran a suite of experiments that varied the connectivity between servers as a function of time in order to test the various algorithms implemented in the prototype. There are 4 basic "disconnectivity scenarios" that we used in our experiments. These scenarios are shown in Figure 26 and explained in the caption. In each experiment, the disconnection described by the disconnectivity scenario is imposed periodically, interspersed with periods of full connectivity. Each experiment lasts for five minutes. All inter-site links are limited to 56 kilobits per second. Sending rates are set such that the inter-message interval from each site is uniformly distributed between 0 and 1 second. In other words, each site sends approximately 2 messages per second. The size of each message is approximately 1 kilobyte[16]. Table 2 presents a summary of the experiments.

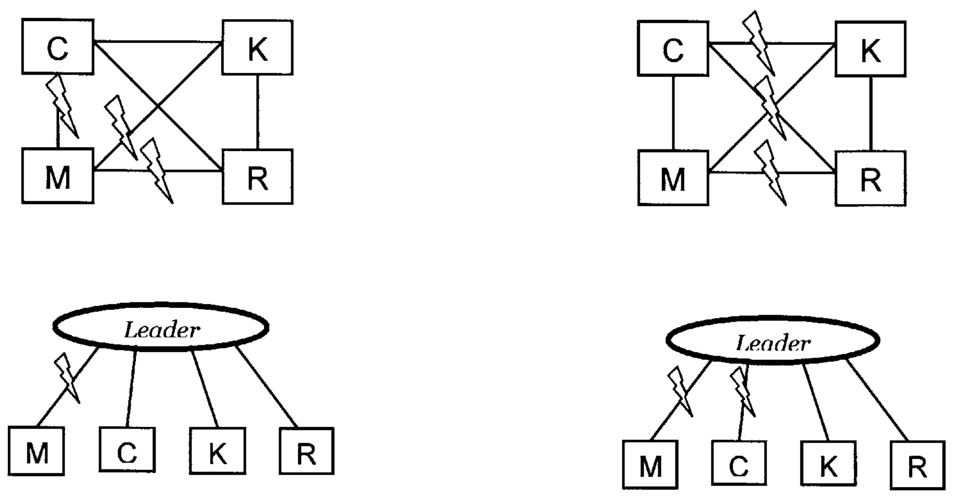


Figure 26: "Disconnectivity scenarios". The 4 sites are labeled M for Metropol, C for Commonwe, K for Kingdom, and R for Republic. During a particular experiment, one of the above four types of disconnections is imposed periodically in between periods of full connectivity. A1 and A2 are imposed when testing the all-to-all protocol, and L1 and L2 are imposed when testing the leader-based protocol. In A1, Metropol is disconnected from every other site. In A2, there is a partition; Metropol and Commonwe are connected on one side of the partition, and Kingdom and Republic are connected on the other side of the partition, and there is no communication across the partition. In L1, Metropol is disconnected from the leader. In L2, both Metropol and Commonwe are disconnected from the leader.

[16] This is the size of each message object after serialization. We estimate that we can reduce the size of each message to be 500 bytes or less after serialization after removing redundancy in our data representations, and by removing instrumentation data added to message objects for the purposes of testing, such as certain timestamps.

Test Number	Ordering Algorithm	Short term disconnect handler	Disconnectivity Scenario	Disconnect time (seconds)	Connect time (seconds)
1	All-to-all	Delay-delivery	A1	10	30
2	All-to-all	Hasten-Delivery	A1	10	30
3	All-to-all	Hasten-delivery	A1	25	25
4	All-to-all	Delay-delivery	A2	10	30
5	All-to-all	Hasten-delivery	A2	10	30
6	All-to-all	Hasten-Delivery	A2	25	25
7	Leader-based	Hasten-delivery	L1	10	30
8	Leader-based	Hasten-delivery	L2	10	30

Table 2: Experiment summary. In each experiment, the clients at each site send messages such that the average aggregate rate of messages sent from each site is 2 messages per second. Experiments last for 5 minutes. Full connectivity is maintained for 'connect time' seconds. Then the disconnectivity scenario is imposed on the network for 'disconnect time' seconds, which is then followed by full connectivity, etc., and the process is repeated for the entire test run.

6.3 Analysis

In the following sections we analyze the results of the tests. We analyze the chat server's performance when facing short term disconnections in its various modes of operation; all-to-all with hasten-delivery (Tests #2 and #5), all-to-all with delay-delivery

(Tests #1 and #4), and leader-based (Tests #7 and #8). We then analyze the performance of the chat server when handling long term disconnections (Tests #3 and #6).

We use three different types of graphs to analyze the experiments. The first type graphs the sequence number of the received message as a function of time. Since the sequence numbers of messages sent from each site are monotonically increasing, these graphs can be used in order to visualize the client's view of the chat display during the run. This graph allows us to visualize time periods in which no messages are delivered as well as gaps in sequence numbers corresponding to periods of disconnection with no reconciliation.

The other two types of graphs plot the latency and rate of received messages as a function of time. These graphs are characterized by two distinct types of regions; 'normal' regions and 'spiked' regions. 'Normal' regions correspond to times of stable connectivity; message latencies are low and messages are delivered at a steady rate. 'Spiked' regions correspond either to message delivery after the detection of a disconnection or to periods of message reconciliation. In these regions, messages are delivered with higher latencies and the rate of message delivery varies. The behavior and performance differences of the various modes of the chat server can be observed from these three types of graphs.

6.3.1 All-to-all with hasten-delivery

Tests #3 and #5 are scenarios in which the chat servers are running in all-to-all mode using the hasten-delivery method for handling short term disconnections. The run involves alternating periods of connectivity and a type "A1" or "A2" disconnection. The disconnection lasts for 10 seconds and is followed by 30 seconds of full connectivity, which is then followed by 10 seconds of disconnection, etc.

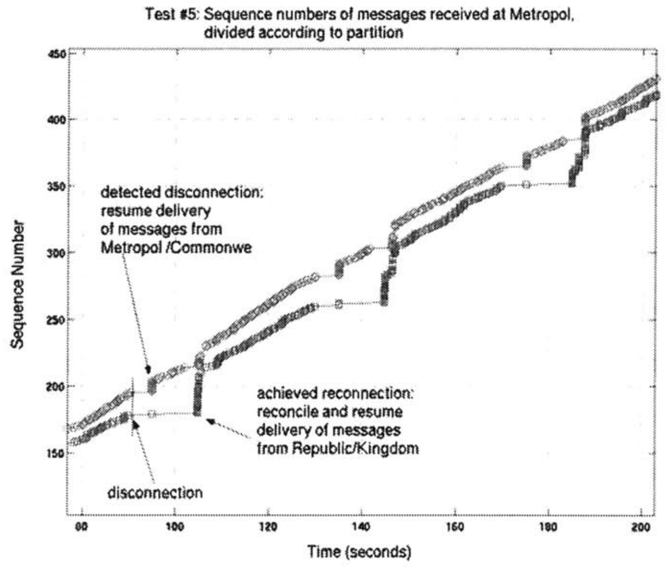


Figure 27: Test #5: Sequence numbers of messages received at Metropol, divided according to partition.

Figure 27 plots the sequence numbers of messages received at Metropol as a function of time in Test #5, in which the "A2" disconnection is imposed. Figure 28 plots the latency and rate of received messages as a function of time for messages received by clients at the site Metropol during Test #5. The figures zoom in on the period from 80 seconds to 200 seconds into the run and cover three different disconnect times; one from 90-100, one from 130-140, and one from 170-180.

We first examine Figure 27. The upper line marks messages received from Metropol and Commonwe, which are the sites in the partition to which Metropol belongs. The lower line marks messages received from Republic and Kingdom, which are the sites on the other side of the partition. Data points, which correspond to delivered messages, are marked by circles in the upper (Metropol/Commonwe) line and are marked by squares in the lower (Republic/Kingdom) line. From time 80-90, all sites are connected and Metropol receives messages from all sites. At 90 seconds, a disconnection occurs. It takes 6 seconds (3 heartbeat intervals) for Metropol to detect that there is a disconnection; during this time, its record of the latest Lamport clock times of Kingdom and Republic do not increase, so no messages are delivered by the Lamport delivery algorithm. After 6 seconds, Metropol recognizes and classifies the disconnection as a short term disconnect. The hasten-delivery method is in place to handle the short term disconnection, so messages from sites on its side of the partition continue to be delivered; note that data points on the upper line resume at approximately 96 seconds. At 100 seconds, full connectivity is resumed. It takes a few seconds for Metropol to detect the new connectivity and reestablish a connection to Republic and Commonwe. After the connection is reestablished, reconciliation occurs and Republic and Commonwe send Metropol the messages that were sent on their side of the partition between 90-100 seconds. Metropol receives and delivers the reconciled messages at approximately 104 seconds; note that in the graph, data points resume on the lower line at approximately 104 seconds. There are no breaks in the sequence numbers; Metropol receives messages sequentially from each site despite the disconnections. Although there are times when no messages are received, messages resume where they left off; this is because the disconnections are short term and reconciliation occurs. According to the hasten-delivery behavior, reconciled messages from the other side of the partition are delivered with the caveat that they are late and out-of –order with respect to other messages in the conversation display.

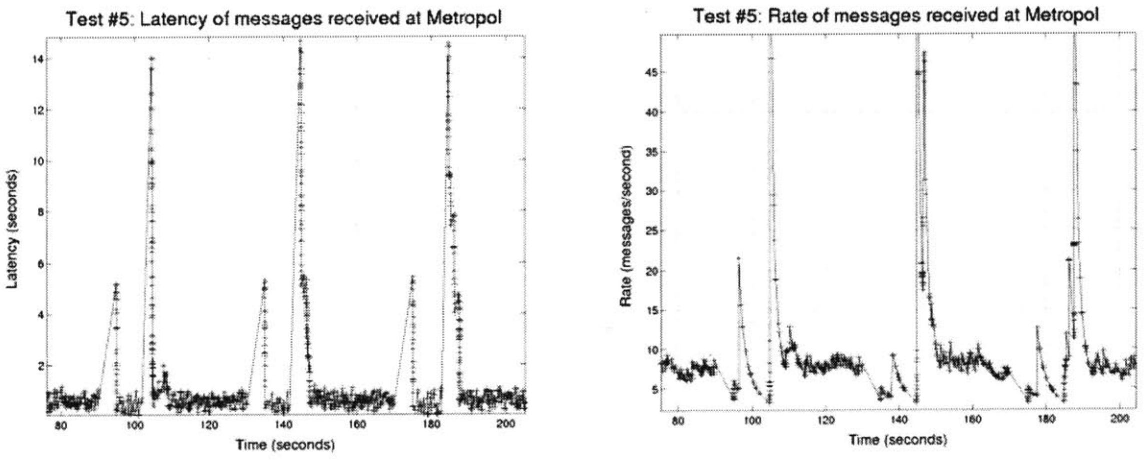


Figure 28: Test #5: Latency and rate of messages received at Metropol.

We trace the same sequence of events in both the latency and rate graphs. In the stable, connected periods, the average rate of delivery is 8 messages/second, which corresponds to 4 sites delivering messages at 2 messages per second. The latency of message delivery is approximately 0.5 seconds on average; this corresponds to the average inter-message time at each site. This agrees with the observation that the Lamport delivery algorithm cannot achieve a better average latency than the greatest average inter-message latency at any of its connected sites; since it must wait for timestamp updates from all sites before it can deliver messages.

From 90-96 seconds, no messages are delivered as the disconnection is detected and classified as a short term disconnection. At 96 seconds, message delivery resumes; the messages from the Metropol/Commonwe partition are delivered with latencies of at most 6 seconds. There is a spike in the rate of message delivery as these messages are delivered in a burst. Then, for a short time, only messages from the Metropol/Commonwe partition are delivered, so the rate falls to approximately 4 messages/second[17]. At 104 seconds, reconciliation occurs and messages from the Republic/Kingdom partition are delivered. The latency of these messages is at most 14 seconds; 10 seconds for the disconnections plus 4 seconds to reestablish the connection and reconcile messages. The rate undergoes another spike of even greater magnitude, since all messages sent by Republic/Kingdom during the disconnection are delivered. After the reconciliation, the rate and latency drops back down to the normal level. There are user interface issues associated with delivering messages at such high rates after reconciliation, as the rate of message delivery may exceed the user's ability to process information; see section 6.3.5.2 for a discussion of this.

6.3.2 All-to-all with delay-delivery

Tests #1 and #4 are scenarios in which the chat servers are running in all-to-all mode using the delay-delivery method for handling short term disconnections. The run involves alternating periods of connectivity and a type "A1" or "A2" disconnection.

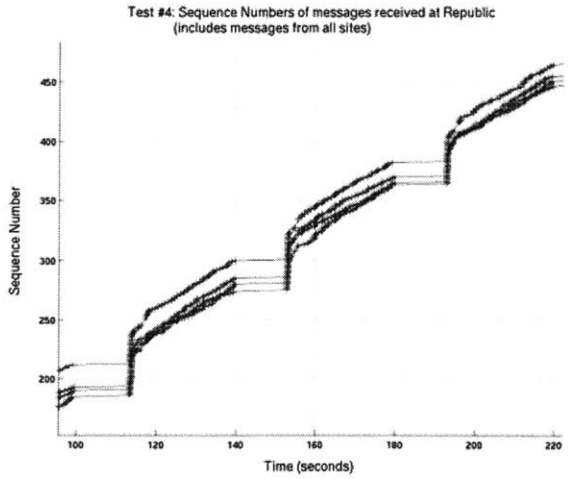


Figure 29: Test #4: Sequence numbers of messages received from all four sites at Republic during Test #4. Note that there are no gaps in the sequence numbers of messages from any site, and that delivery of messages on both sides of the partition is delayed during disconnects.

[17] This effect is lost in the graph due to the effect of the moving average.

Delay-delivery is a noteworthy mechanism because it maintains a global order on the message delivery despite the presence of disconnections. Figure 29 plots the sequence numbers of all of the messages received at Republic during Test #4, in which a type A2 disconnection is imposed. Note that there are no gaps in the sequence numbers of messages from any site. Delivery of messages from both sides of the partition is delayed during a disconnection; after reconnection, messages are reconciled and delivered in a global order.

Figure 30 plots the latency and rate of messages received at Commonwe during Test #1. Note that in hasten-delivery mode, there are two "spikes" during a disconnection in the latency graph. The first spike is from the resumption of delivery of messages in the site's own partition after the detection of a disconnection, and the second spike is from the delivery of reconciled messages from the reconnected partition. Delay-delivery mode eliminates the first spike, and incorporates all of the messages that were sent by sites during the disconnection in the second spike. The result is a greater overall latency of messages; no messages are delivered to the user during the disconnection. After reconnection, all messages are delivered to the user in a global order, so messages that a user typed at the beginning of the disconnection are displayed only after reconciliation. This is another illustration of the latency versus global order tradeoff; in order to achieve global order, the delay-delivery method tolerates greater overall latency on the message delivery.

Another effect of delay-delivery is that after reconciliation, messages are delivered at a rate that exceeds the rate of delivery during reconciliation of the hasten-delivery method; this is simply because there are more messages to be delivered after reconciliation in the delay-delivery method. The effect is not so apparent when comparing the rate graphs from Figure 30 (delay-delivery) with Figure 28 (hasten-delivery). The difference between the two graphs is that with delay-delivery, higher rates of message delivery persist for a longer time since a greater number of messages are reconciled. As was mentioned earlier, this increased rate of delivery may cause problems for the user.

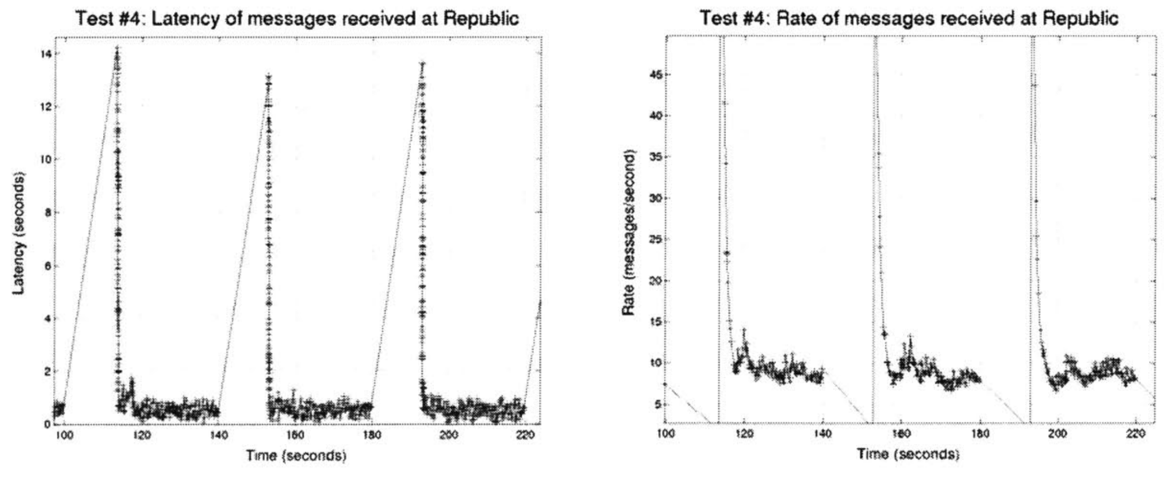


Figure 30: Test #4: Latency and rate of messages received at Commonwe.

6.3.3 Leader-based

In tests #7 and #8, the chat server is running in leader-based mode. In the test configuration, the leader is a process on the router machine. The leader could be a process running on any machine to which every site can connect.

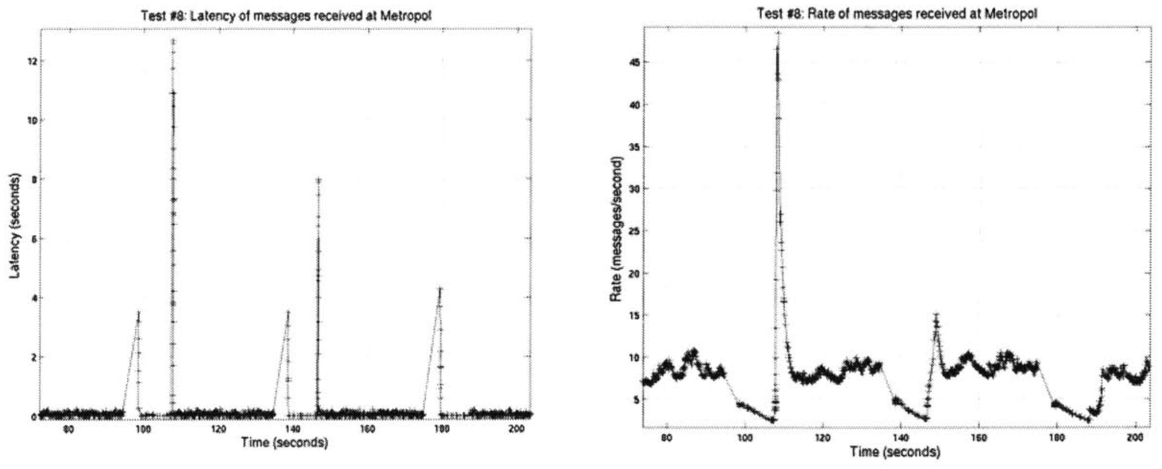


Figure 31: Test #8: Latency and rate of messages received at Metropol.

As we mentioned before, the best average latency that can be achieved by all-to-all is equal to the worst average inter-message interval of any of the sending sites. The leader-based protocol has no such restrictions on the average message delivery latency. The only lower bound on the message delivery latency is the network round trip time from the site to the leader and back. Since we don't explicitly control the latencies of links, the round trip time is very small, which results in the leader-based protocol significantly outperforming all-to-all in terms of latency; latencies for the leader-based protocol average about 100 milliseconds.

The leader-based protocol implements only the hasten-delivery behavior for handling short term disconnections. The implementation is slightly different than the implementation used for all-to-all. In the leader-based case, when a site that was disconnected from the leader reconnects and reconciles, it does not receive messages from the other sites that remained connected to the leader. The rationale for this is that the site that was disconnected would be inundated by messages from sites that remained connected to the leader, and the user would not be able to process all of the information. Sites that remained connected to the leader do receive messages from the reconnected site; the rationale for this is that since there is only a small number of reconnected sites, the number of messages received upon reconciliation would not be too high and the messages could be displayed without confusing the user. Therefore, the sites that remain connected to the leader should see no gaps in the sequence numbers of messages coming from the disconnected site. Although that same reasoning should logically lead us to implement rate-limiting behavior during reconciliation in the all-to-all protocol's implementation of hasten-delivery, we did not do so. The all-to-all implementation reconciles messages and displays them on the screen regardless of the increase in the rate of delivered/displayed messages.

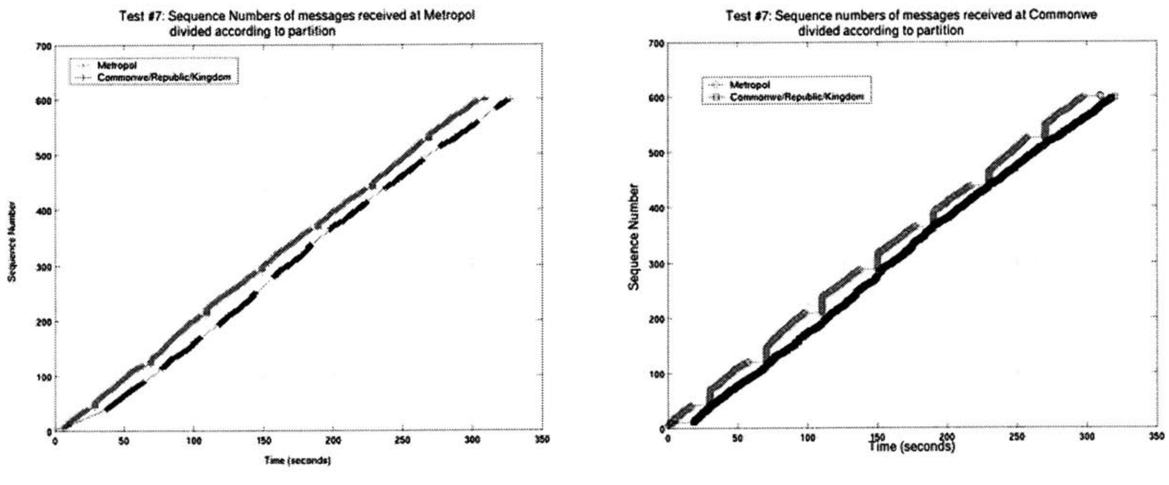


Figure 32: Test #7: The graph on the left shows sequence numbers of messages received at Metropol, divided according to partition. The graph on the right shows sequence numbers of messages received at Commonwe, divided according to partition.

In Test #7, Metropol is periodically disconnected from the leader, while Commonwe, Republic, and Kingdom remain connected to the leader. Figure 32 plots the sequence numbers of messages, from the point of view of sites on different sides of the partition imposed in Test #7. The graph on the left shows the sequence numbers of messages received at Metropol. Notice that Metropol sees gaps in the sequence numbers of messages from the other side of the partition. The graph on the right shows the sequence numbers of messages received at Commonwe. Due to reconciliation in which messages from Metropol are delivered to sites that remained connected to the leader, Commonwe sees no gaps in the messages sent by Metropol.

Figure 31 plots the latency and rate of messages at Metropol during Test #8, in which both Metropol and Commonwe are periodically disconnected from the leader. The graph traces three disconnection periods. Each disconnect is characterized by a small spike, in which the latency rises to approximately 6 seconds. Sometimes, a larger spike, in which the latency rises to approximately 12 seconds, is observed. The smaller spikes in latency are due to the delay in message delivery that occurs when the disconnection is being detected. Once the disconnection is detected, Metropol maintains only intra-site chat. The latency of message delivery drops to a very low level, since it eliminates the round-trip time to the leader. The rate drops from 8 messages/second to approximately 2 messages per second, corresponding to intra-site chat only. The optional second spike in the latency graph is due to the manner in which disconnections are performed in Test #7; both Commonwe and Metropol disconnect and reconnect to the leader at approximately the same time. In the first two disconnections, Metropol connects to the leader before Commonwe. Therefore, it receives the reconciled messages from Commonwe after Commonwe reconnects to the leader. In the third disconnection, Metropol connects to the leader after Commonwe, so it does not receive Commonwe's reconciled messages. The corresponding graphs at Commonwe exhibit one spike in the first two disconnections, and two spikes in the final disconnection, for the same reason. After reconnection and reconciliation, the message rate settles back to 8 messages per second, and the latency settles back to approximately 100 milliseconds.

6.3.4 Long term disconnections

Long term disconnections are easy to analyze and understand because there is no reconciliation. Figure 33 plots the sequence numbers of messages received at Kingdom during Test #36, in which a Type "A2" disconnection is imposed periodically as a long term disconnection. Note that there are no gaps in the sequence numbers of messages coming from the Republic/Kingdom side of the partition. However, Kingdom's view of the messages from the Metropol/Commonwe has gaps because of long term disconnections.

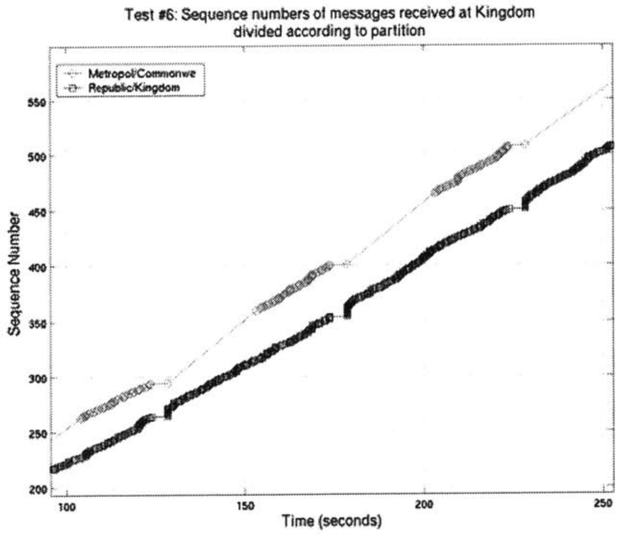


Figure 33: Test #6: Sequence Numbers of messages received at Kingdom, divided according to partition.

Figure 34 plots the latency and rate of messages received at Kingdom during Test #6. During connected times, latency is at normal levels and messages are delivered at 8 messages per second. The disconnection is detected after 6 seconds, resulting in a spike in both the latency and rate graphs. During the disconnected time, the message rate drops to 2 messages/second on average, corresponding to two sites sending at 2 messages/second. Latency remains the same, since we are in all-to-all mode, and the inter-message latency of connected sites is still 0.5 seconds. Reconnection does not result in reconciliation and message rates return to normal.

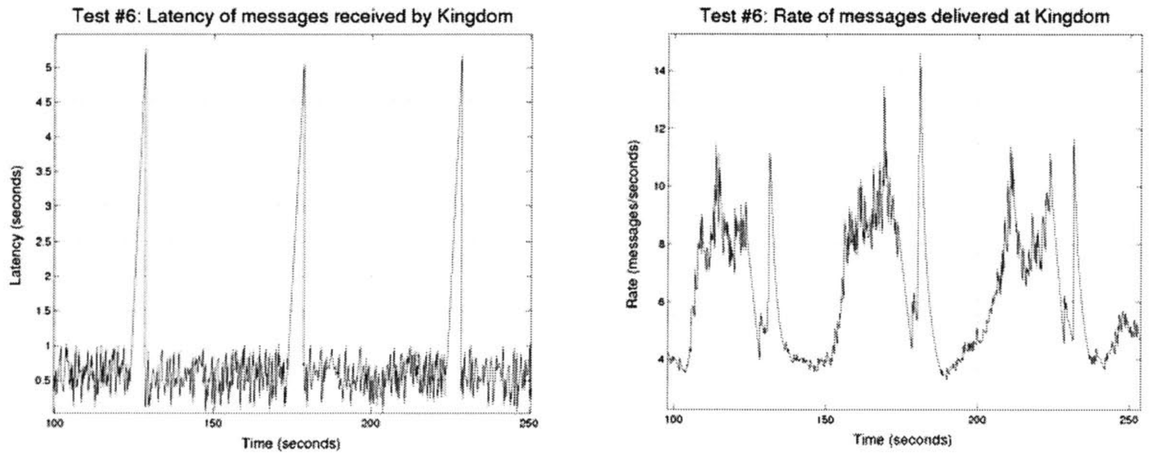


Figure 34: Test #6: Latency and rate of messages received at Kingdom.

6.3.5 Discussion

6.3.5.1 Global order versus latency

The implemented approaches differ in how they balance global order and latency. Leader-based has the best latency characteristics, but achieves only intra-site ordering when a disconnection occurs. All-to-all with hasten-delivery has worse latency characteristics than the leader-based solution, but performs better in terms of overall message delivery latency compared to all-to-all with delay-delivery. All-to-all with hasten-delivery sacrifices global order in order to achieve its improved latency characteristics, but is able to achieve ordering among connected sites when global connectivity is not available; this is better than reverting to only intra-site chat as in the leader-based case. All-to-all with delay-delivery maintains global order, but pays the price in the latency of delivered messages, as it delays delivery until global order can be achieved.

Differences in latencies of message delivery result from one of a number of reasons. During normal operation, the average latency of message delivery of the all-to-all protocol is determined by the average inter-message times of connected sites; the Lamport ordering algorithm cannot improve on this bound as long as it uses only logical timestamps in its protocol. The all-to-all protocol could potentially make use of real time in its protocol to improve its average latency; it could use real time to estimate the latencies between sites (assuming the sites are have synchronized clocks), and use this to produce better estimates of when a message is ready for delivery, such as in [10]. However, improving the average delivery latency during stable operation is not really a priority; we are only looking to maintain latencies that support human conversation.

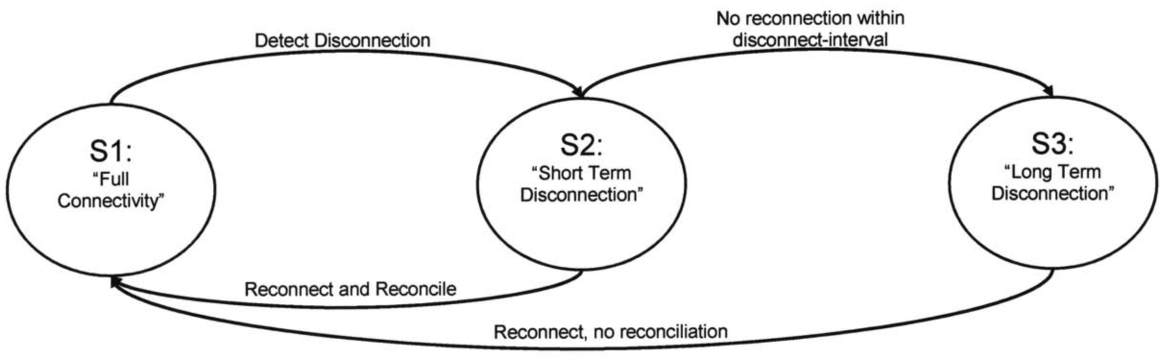

Detect Disconnection
No reconnection within disconnect-interval
S1: "Full Connectivity"
S2: "Short Term Disconnection"
S3: "Long Term Disconnection"
Reconnect and Reconcile
Reconnect, no reconciliation

Figure 35: A state machine representation of the Intermittent Global Order protocol.

Latencies and rates increase due the detection of disconnections and during reconciliation. We can understand this better by referring to the state machine representation of the Intermittent Global Order protocol, which we reproduce in Figure 35. The Intermittent Global Order protocol achieves global ordering in state S1, imposes either a relaxed ordering or global ordering in state S2 (depending on whether the hasten-delivery or delay-delivery behavior is in place), and imposes the relaxed ordering in state S3. The transition from S1 to S2 causes increased latencies and rates due to the process of detecting a disconnection; as long as the mechanism for detecting a broken link takes time, increased latencies and rates due to this transition are unavoidable. The transition from S2 to S1 produces increased latencies and rates due to reconciliation. This source of increased latency can only be avoided if there is no reconciliation, as can be seen from the results of the long term disconnect tests in section 6.3.4. There are no increases in latency or rate in the state transitions S2 → S3 and S3 → S1.

6.3.5.2 Message delivery rates

An important factor that we did not take into account when designing the prototype is the rate of message delivery when handling disconnections. Our general observation is that protocols that trade off better (i.e., closer-to-global) ordering properties for latency also experience higher rates of message delivery during detection of disconnections and reconciliation. All-to-all achieves closer-to-global ordering than the leader-based during disconnections because it includes all connected sites in its ordering, whereas the leader-based protocol achieves only intra-site chat during disconnections. Thus, when detecting disconnections in all-to-all, messages from all connected sites are delayed; once the disconnection is detected, these messages are delivered, which results in an increased rate of message delivery. All-to-all with hasten-delivery mitigates this effect through its "two-spike" behavior; there is an initial spike in the message rate when messages from connected sites are delivered after detecting the disconnection, followed by another spike in the message rate following reconnection and reconciliation. All-to-all with delay-delivery has only one rate "spike" that includes all of the messages from all connected sites after reconnection and reconciliation; this spike includes all of the messages delivered in both 'spikes' in the hasten-delivery method and is therefore more pronounced and prolonged.

The benefit of achieving a closer-to-global ordering during times of disconnection is that it improves the user experience and extends the range of communication. However, these benefits are lost if increases in the rate of message display make it impossible for the user to process the information on the screen. In the next generation chat prototype, an important issue to address is the maximum rate at which messages should be delivered to the screen. The usability of the chat application should not suffer as a result of closer-to-global message ordering and message reconciliation.

7 Future Work

We have identified a number of issues to be addressed in future versions of the chat prototype and discuss these issues in this section.

7.1.1 User interface

There are a number of user interface improvements that could be made to the chat client that could improve its usability. The manner in which messages are delivered to chat clients in the current implementation poses two problems. First, the hasten-delivery behavior results in "late" messages being delivered to the client. The client's Graphical User Interface (GUI) should display these messages in a manner that reflects their delayed delivery; for example, it could color code these messages or display them in a separate window. The second challenge is that messages are delivered at higher rates during certain periods, due to the chat server's handling of disconnections and message reconciliation. The client GUI could limit the rate of message display to a reasonable upper bound, so as not to inundate the user with messages. The client would need to handle message arrivals that exceed this rate. Possible solutions include opening up a separate window, or displaying messages at only the maximum display rate (regardless of the arrival rate). Messages that arrive at high rates could be buffered and displayed in order with some delay, and the client could 'catch up' to real-time later, assuming that message rates eventually decrease.

There are also a number of simple enhancements to the client GUI that could improve the user experience; for example, messages could be color-coded by site, and the client could notify the user of connectivity problems experienced by the site's chat server.

7.1.2 Reconciliation after long term disconnections

As was discussed in section 5.1.3.4, reconciling messages after long term disconnections would allow each site could keep a FIFO record of messages sent by every other site during a session. Long term reconciliation would be useful in order to 'replay' the messages sent during a particular session; reconciled messages could be stored in a database and the entire chat conversation could be replayed using timestamp information contained in each message.

Incorporating long term reconciliation would require changes to the design; long term reconciliation messages are lower priority messages than real-time chat messages and short term reconciliation messages. In order to maintain the priority of real-time and short term reconciliation messages, it would be necessary tu implement a priority sender, with different queues for messages of different priorities. In such a design, a long term reconciliation thread would deposit reconciliation messages in lower priority queues in the priority sender, which would schedule and send them appropriately. This approach would ensure that long term reconciliation does not steal valuable bandwidth from higher priority messages.

7.1.3 Stability Tracking

In the current implementation, sites that remain connected impose the same ordering on each other's messages; this is the pair-wise ordering property described in 4.3.6. However, two sites that are connected may deliver different sequences of messages to their clients because they may have not always share the same set of neighbors. For example, consider a scenario with three sites *A*, *B*, and *C*. Initially, all three sites are connected. After some time, *C* is disconnected from *A*. If transitive connectivity holds, then *C* should soon be reconnected to *A* through *B*. However, let us assume that transitive connectivity does not hold, or that it is not achieved quickly enough. In that case, *C* might send some messages during the time in which it is disconnected from *A* which are received and delivered by *B*. Site *A* does not receive these messages, so the sites *A* and *B* see a different sequence of messages although they are connected. The same situation could arise if *C* disconnected from *A* and, soon afterwards, also disconnected from *B*. A desirable property to be satisfied by the chat application would be that all sites that are connected see exactly the same sequence of messages, regardless of whether or not the routing layer implements transitive connectivity. This could be achieved by implementing a stability tracking module at the application layer. The stability tracking module could ensure before a message is delivered that it has been received by all connected sites (that is, that the message has become 'stable'). Stability tracking would increase message delivery latency, but would guarantee that connected sites always see the same sequence of messages. Stability tracking algorithms are described in [15].

7.1.4 Test-bed Improvements

An important limitation of the test-bed is that we currently do not explicitly control the latencies of the test-bed links. Future improvements to the test-bed should implement variable latencies for each point-to-point link. In general, a realistic test-bed should include emulation of the various types of links present in the MC2C environment.

8 References

1. Dougherty, S.T., *JEFX 2002 ends with positive results*, in *Air Force Link.* August 9, 2002.

2. Butler, A., *War Planners Talk In 'Chat Rooms' to exchange targeting data, tips*, in *Inside the Air Force.* April 25, 2003.

3. Fulghum, D.A., *Tests of MC2A Reveal Problems and Promise*, in *Aviation Week & Space Technology.* 2002.

4. Lamport, L., *Time, Clocks, and the Ordering of events in a distributed system.* Communications of the ACM, 1978. **21**(7): p. 558-565.

5. Melliar-Smith, P.M., L.E. Moser, and D.A. Agarwal, *Ring-based Ordering Protocols.* Proceedings of the IEEE International Conference on Information Engineering, 1991(Singapore): p. 882-891.

6. Chockler, G., V., I. Keidar, and R. Vitenberg, *Group Communication Specifications: A Comprehensive Survey.* ACM Computing Surveys, 2001. **33**(4): p. 1-43.

7. Birman, K.P. and T.A. Joseph, *Reliable Communication in the presence of failures.* ACM Transactions on Computer Systems, 1987. **15**(1): p. 47-76.

8. van Renesse, R., et al., *Horus: A Flexible Group Communication System.* Communications of the ACM, 1996. **39**(4): p. 76-83.

9. Moser, L.E., et al., *Totem: A Fault-tolerant Multicast Group Communication System.* Communications of the ACM, 1996. **39**(4): p. 54-63.

10. Koch, R., L.E. Moser, and P.M. Melliar-Smith, *Global Causal Ordering With Minimal Latency.* Proceedings of the 2nd International Conference on Parallel and

Distributed Computing and Networking (PDCN), 1998(Brisbane, Australia): p. 262-267.

11. Moy, J.T., *OSPF:Anatomy of an Internet Routing Protocol.* 1998: Addison-Wesley.

12. *Handbook on Satellite Communications*. 3rd Edition. 2002, New York, NY: Wiley-Interscience.

13. Rosenzweig, P., M. Kadansky, and S. Hanna, *The Java Reliable Multicast Service: A Reliable Multicast Library.* Sun Microsystems Laboratory Technical Report TR 98-68, September, 1998.

14. Pai, V.S., P. Druschel, and W. Zwaenepoel, *Flash: An efficient and portable Web server.* Annual Usenix Technical Conference, 1999.

15. Guo, K., et al., *Hierarchical message stability tracking protocols.* Department of Computer Science, Cornell University: Technical Report, CS-TR 97-1647, September 1997.

[illegible]