Reliable Multicast for Publish/Subscribe Systems

by

Qixiang Sun

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

May 2000

Author . . . .
Department of Electrical Engineering and Computer Science
May 19, 2000

Certified by . . . .
Nancy Lynch
Professor
Massachusetts Institute of Technology
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Reliable Multicast for Publish/Subscribe Systems

by

Qixiang Sun

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Group-based reliable multicast is an important building block for distributed applications. For large systems, however, traditional approaches do not scale well due to centralized recovery mechanisms and excessive message overhead. In this paper, we present a reliable probabilistic multicast, *rpbcast*, that is a hybrid of the centralized and gossip-based approaches. In particular, rpbcast extends previous work by supporting high packet rates and many active senders. Rpbcast uses gossip as the primary retransmission mechanism and only contacts loggers if gossips fail. Large groups of active senders are supported using *negative gossip* which describes what a receiver lacks instead of what it has. Moreover, negative gossip allows *pull* based recovery and converges faster than *push* based recovery. Rpbcast also applies hashing techniques to reduce message overhead. Garbage collection in rpbcast is stability oriented. The approximate membership protocol in rpbcast exploits some garbage collection flexibilities to avoid expensive join/leave operations.

Thesis Supervisor: Nancy Lynch
Title: Professor
Massachusetts Institute of Technology

Acknowledgments

First, I would like to thank Daniel Sturman and Professor Nancy Lynch for their guidance. Their emphasis on both the theoretical aspect and the system aspect of my research helped me a great deal in understanding and evaluating my work.

Second, I would like to thank Idit Keider and Mark Astley for their input on the design and testing of the hybrid protocol. Discussions with them were very helpful in the development of the protocol. I would also like to thank Professor Ken Birman and Oznur Ozkasap from Cornell for their assistance in understanding *pbcast*.

Lastly, I would like to thank my parents Yufeng and Ying for their unwavering support throughout my career at M.I.T. Without them, none of this would have been possible.

To My Parents

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many applications today will benefit from a robust, scalable, and reliable multicast protocol. In this thesis, we leverage existing research results and propose a new reliable multicast protocol that is more scalable. Section 1.1 presents the motivation behind this work and the target application scenarios. Sections 1.2 and 1.3 give some background information related to our target application and previous work. We conclude in Section 1.4 with an brief summary of the protocol and an outline for the rest of the thesis.

1.1 Motivation

Publish/subscribe systems (pub/sub) offer a data distribution service where new information, *events*, are published into the system and routed to all interested subscribers. These systems provide a paradigm for integrating applications in distributed environments. Reliable event delivery is an important property in pub/sub systems. Traditionally, multicast based protocols are used for mass information distribution. However, reliable multicasts in pub/sub systems differ from other settings, such as group communication or multicast data transfer, in four distinct aspects. First, pub/sub systems must support intermittently connected subscribers. Second, high availability to applications in terms of steady throughput and forward progress is essential: stalling is consider a fault. Third, large numbers of subscribers with diverse interests require a multicast protocol to be scalable in terms of size of the group and number of groups operating simultaneously. And fourth, multicast must operate efficiently under changing membership.

Subscribers in pub/sub systems may have intermittent connectivity. Therefore a reliable multicast in this context must also guarantee that events are delivered to disconnected subscribers when they reconnect. This reliability criteria differs from group communication setting such as ISIS [8] in that disconnected members are treated as faulty members in ISIS and are not covered under the "all or none" delivery semantics characterized in [16]. Also, group communication services are designed for collaborative applications with strong state consistency requirements. In contrast, subscribers in a pub/sub system are generally non-collaborative and manipulate events independent of each other. Consequently, recovery through state transfer is not applicable.

The networking community treats disconnected members in a multicast data transfer in a similar manner as ISIS. Disconnected members are simply excluded from the reliability guarantee. In short, many systems do not extend reliable delivery guarantees to members with changing status. This type of guarantee is insufficient for pub/sub systems with mobile agents. Hence a multicast protocol for pub/sub systems must provide support for intermittently connected members.

Pub/sub systems also require continuous forward progress. Due to unstable events that has been received by some but not all members, most reliable multicast protocols to date will stall progress via flow control mechanisms when their buffers are full. Without using stable storage, stalling is necessary to enforce reliability. Unlike pub/sub, in most systems stalling is not considered a fault for maintaining state consistency. Practically speaking, stalling may not be a concern if the send rates in a system is sufficiently slow. Unfortunately, pub/sub systems have fast send rates. Moreover the need for high availability precludes stalling as an option. For pub/sub systems, having inconsistency among some subscribers is more desirable than lack of progress in the entire system. For example, the latest stock quote should not be delayed simply because one subscriber lags behind. This suggests that End-to-End reliability [33] is preferable.

Diverse subscriber interests, where each subscriber is only interested in a small subset of all events, complicate the design of reliable multicast in pub/sub systems. One potential approach is to divide subscribers dynamically into different multicast groups based on their interests, but this approach produces hundreds, if not thousands, of multicast groups operating simultaneously. Thus any multicast protocols for such a pub/sub system must also scale well with respect to the number of groups. Existing protocols are inefficient in this aspect because of high protocol overhead associated with periodic updates within each group.

Large number of publishers and subscribers also presents a challenge for reducing protocol overhead, maintaining membership information, and detecting failure. Due to the sheer size of the system, membership changes occur often in pub/sub systems. The Virtual Synchrony model, used in group communication to maintain precise membership and strict message ordering, will block sends while membership is changing. This constraint can quickly stall a system, which is strongly undesirable. Pub/sub systems can operate on a weaker model because a membership change implies that a subscriber has changed its interest in some events. Since each subscriber has a filter for what they are interested in, an approximate membership that includes everyone who might possibly want an event is sufficient. Extraneous and uninteresting events can be filtered out at the receiver ends in accordance with the End-to-End argument. This approach is similar to CONGRESS[4] and Maestro[7].

1.2 Background

Pub/sub systems are middleware for “gluing” together heterogeneous applications that operate on the same information space. These system are divided into three broad classes, based on their information filtering and routing mechanisms. The first class of pub/sub systems is *channel-based*. These systems have a number of predefined channels. Each channel has a designated topic. Information publishers send new events to the appropriate channels. All interested subscribers listen directly on each channel for new events. The CORBA Event Service [15] is an example of a channel-based pub/sub system.

The second class of pub/sub systems is *subject-based*. Instead of predefined channels, events are published into the system with specific topics in the subject field. Subscribers provide subscriptions that filter against the subject field to extract interested events. A classic example is the newsgroup. Another example is TIBCO Rendezvous [19] system. In TIBCO, the subject field consists of a string such as `thesis.sun.qixiang`. A subscription filter is a regular expression. For example, the subscription filter `thesis.*.*` will select all theses.

The third class of pub/sub systems is *content-based*. These systems not only allow filtering based on the subject but also on the entire content of an event. The content of an event is specified with a collection of attributes, e.g. {`type=thesis, author=sun`}. Through the introduction of attributes, content-based pub/sub systems allow individual subscribers more expressiveness in selecting the type of events they are interested in and migrate the burden of event matching to the underlying middleware. Examples of content-based pub/sub systems include Elvin [34], SIENA [10], and Gryphon [2, 5].

A related field to pub/sub systems is group communication, pioneered by ISIS [8]. These systems focus on providing a general distributed framework for processes to collaborate on a common task. To simplify correctness analysis of a distributed algorithm or application, ISIS introduces the virtual synchrony (VS) model. Under VS, group membership changes are ordered along with regular messages. Moreover, if two processes proceed together from one view to another, then they deliver the same messages in the first view. These constraints allow ISIS to enforce total and/or casual ordering of messages in faulty environments. In the event of a network partitioning, ISIS allows only the primary partition to continue operating.

Transis [3] extends ISIS by supporting partitionable operations. Instead of only permitting the primary partition to continue, Transis allows each partition to proceed together and "merge" the results when network partition heals. The nature of the merge operation is obviously application dependent. In some cases, the merge operation implies exchange missing messages. In others, merging might involve combining partial but independent results. Transis also introduces the notion of hidden views to assist in merging and totally order past view history. Totem [1] imposes an even stronger consistency model than Transis through extended virtual synchrony (EVS) [27]. EVS guarantees that if messages are delivered to multiple components of a partitioned network, then the message ordering is consistent in all of these components. The added constraint simplifies the recovery process for maintaining exact replicas.

Though group communication's emphasis on consistency and replicated data items is orthogonal to the functions of pub/sub systems, there are many similarities in the underlying multicast transport protocols. For instance, system scalability and low message delivery latencies are common concerns in designing multicast protocols for group communications and publish/subscribe systems. Techniques such as vector timestamp and randomized gossip are useful in both cases.

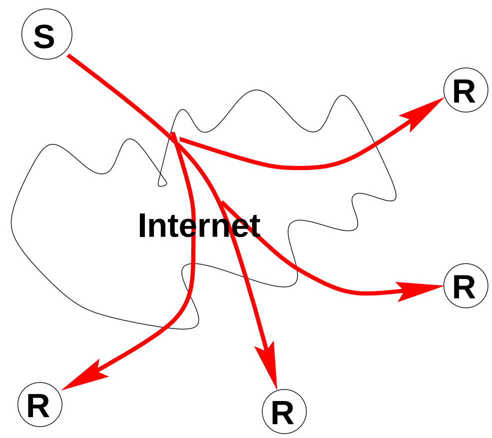

Figure 1-1: Multicast

1.3 Related Work

The primary goal of a reliable multicast protocol is to deliver a message from a sender to all receivers, see figure 1. Typically, these protocols run on top of an unreliable communication service, such as the Internet using IP multicast. Two general approaches exist for building reliable multicast. The first approach uses *loggers*: centralized servers with stable storage that archive packets and handle retransmission requests. The scalability of this approach depends on the availability of logger resources such as logger processing speed and network link bandwidth. With large groups or high traffic multicast groups, logger resources are easily exhausted.

One example of the centralized approach is Log-Based Receiver-reliable Multicast (*LBRM*) [18]. *LBRM* is *receiver-reliable* because individual receivers are responsible for detecting missing packets and requesting the appropriate retransmissions from dedicated loggers. In order to detect missing packets, senders include a sequence number in each packet. Hence any gap in the sequence number implies that there are missing packets. However, during idle periods, a receiver may not realize that it is missing the latest packet. Thus receiver-reliable multicast also requires senders to periodically send *heartbeat* messages to notify receivers of the latest sequence number. *LBRM* uses a variable heartbeat scheme to reduce heartbeat overhead.

An alternative to receiver-reliable protocols are *sender-reliable* protocols where senders require packet arrival acknowledgments (ACK) from each receiver and retransmit any unacknowledged packets. Reliable Multicast Transport Protocol (*RMTP*) [22] is a sender-reliable protocol. One drawback in sender-reliable multicast is the ACK implosion problem, which occurs when many ACK messages converge on the sender site. *RMTP* uses a hierarchical structure to reduce the number of ACKs at the sender site. For large groups, receiver-reliable protocols introduce less network and processing overhead than sender-reliable protocols. Furthermore, receiver-reliable protocols are ideal for enforcing application-level end-to-end reliability.

A second reliable multicast approach relies on peer-based recovery mechanisms. Instead of dedicated loggers, peer-based mechanisms use all members in a multicast group, both senders and receivers, as retransmission sources. When a particular receiver is missing a packet, any group member may process and service retransmission requests. An example of the peer-based approach is Scalable Reliable Multicast (*SRM*) [13]. Servicing rate is no longer a constraining factor in *SRM* because any member may handle a retransmission request. However, the message overhead introduced by peer-based requests and repairs is substantial. In *SRM*, requests and repairs are multicast, making the protocol less scalable from a network bandwidth perspective. Moreover, a retransmission request in *SRM* may cause redundant retransmissions from different members. *SRM* resolves this problem by randomizing the delays in retransmissions and suppressing retransmissions when other members have already serviced them.

A more scalable peer-based protocol is Bimodal multicast (*pbcast*) [9]. *Pbcast* uses point-to-point *gossip* to reduce excessive message traffic. In *pbcast*, each member in the multicast group periodically gossips to other members. During each gossip round, each receiver selects a random target in the multicast group and sends a digest of the current buffer to the gossip target. Upon receiving the digest, the target receiver can determine if the gossiper has a packet which the target does not and request retransmission via point-to-point channels. This approach avoids *SRM*'s redundant retransmission problem. Thus, *pbcast* is more stable than *SRM* in terms of throughput under varying network conditions [30]. Other gossip style protocols include replicated database maintenance [11], group membership [14], resource discovery [17], and failure detector [35].

Without loggers to archive old packets, peer-based approaches usually do not guarantee reliable delivery. For example, suppose some members disconnect for a long period. Then the entire multicast group has to stall progress when each member's message buffer is full, or exclude disconnected members from the multicast group in order to release buffer space. In large-scale information dissemination applications, neither stalling nor excluding members is desirable. Ozkasap et al. [29] proposes selectively archiving a packet for a longer duration than the garbage collection limit. However, this approach does not fully guarantee reliable delivery either since network congestion may cause a packet to be dropped before arriving at designated archiving sites, even after many rounds of gossip.

1.4 Our Contributions

In this thesis, we build a hybrid multicast protocol that combines the logger-based recovery of *LBRM* and gossip-based recovery of *pbcast*. In this hybrid protocol, instead of waiting for or excluding temporarily disconnected members, we guarantee continuous forward progress for all connected members and allow disconnected members to recover using loggers' stable storage. We also use gossip-based recovery for reducing logger workload. Since we use stable storage, our protocol guarantees reliable event delivery to all subscribers if an event arrives at a stable storage site. Moreover, if a publisher does not crash, we guarantee that each event will eventually reach some logger's stable storage. To efficiently manage rapidly changing membership without sacrificing reliability guarantees or performance, our protocol also maintains an approximate membership that introduces very little overhead.

Besides strengthening delivery guarantees, we introduce several optimizations in the gossip messages to make the protocol more scalable in terms of send rates and group size. We exploit situations where most members in the gossip phase have the same information. In these situations, we first verify, using small hash signatures, that two members indeed have different information before actually sending out large gossip related messages. Our second optimization involves sending out only retransmission requests in the gossip messages. This approach reduces the gossip message size compared to *pbcast* where a digest of a member's buffer is sent in each gossip message.

To verify our delivery guarantees, we give a detailed description of our protocol and its correctness proof. We also present some theoretical models for analyzing gossip efficiency. To substantiate our performance claims, we compare our simulation performance results with our two parent protocols: Log Based Receiver-reliable Multicast (LBRM) [18], designed specifically for distributed interactive simulation (DIS), and Bimodal Multicast (pbcast) [9], designed for group communication.

Though we have an implementation of our protocol, its prospects for deployment are still unclear for two reasons. First, in the publisher/subscriber setting, we must find an efficient mapping between various subscriber interests to a reasonable number of multicast groups. Second, our protocol relies on the existence of a wide-spread unreliable multicast substrate such as IP Multicast. This assumption, however, is not true at the current time. We will have to wait and see what kind of multicast capabilities will be available in the future. Fortunately, the design of our hybrid protocol is extremely flexible and modular. We can swap in and out different implementations for each individual hybrid protocol components without changing the functionalities and the reliability guarantees. Performance, on the other hand, may change significantly depending on which implementations are used.

The remainder of the thesis is organized as follows. Chapter 2 presents our desired specification for a reliable multicast protocol and an overview of our hybrid protocol. Chapter 3 details the logger-based recovery mechanism in the hybrid protocol and proves its correctness in implementing our specification. Chapter 4 discusses the gossip-based recovery mechanism, its effectiveness through mathematical analysis, and how we combine logger-based and gossip-based mechanisms in the hybrid protocol. Chapter 5 proposes several optimizations in reducing protocol overhead, garbage collection, and membership. Chapter 6 briefly outlines the necessary relaxations to the specification in order to deal with member crashes. Chapter 7 gives a comprehensive performance comparison between our protocol, *LBRM*, and *pbcast*. Chapter 8 concludes with a summary of the main contributions of this thesis and some interesting things we learned using formal specifications and descriptions. Parts of this thesis will be presented in an upcoming publication in *International Conference on Dependable Systems and Networks (DSN 2000)* titled "A gossip-based reliable multicast for large-scale high-throughput applications."

Chapter 2

Protocol Overview

In this chapter, we will give a simple specification for reliable multicast. Informally, the specification guarantees message delivery in an environment where messages can be dropped or duplicated. To simplify the discussion of our protocol, the specification in this chapter does not deal with sender or receiver crashes. We will amend this shortcoming in chapter 6 when we explore scenarios involving crashes. With the specification in mind, we will then outline the key aspects of our three phase multicast protocol *rpbcast*. *Rpbcast* is a hybrid protocol that uses both gossip-based recovery and logger-base recovery. We conclude the chapter with a specification of the underlying unreliable transport layer that our protocol assumes.

2.1 Specification of Reliable Multicast

We model a reliable multicast with a set of buffers, one for each sender and receiver pair. Our specification exports two external interfaces: *user_send* and *user_receive*. Figure 2-1 shows a graphical representation of our specification. Notice that our specification does not impose any ordering constraints on message delivery.

Informally, when a *user_send* occurs, we add a message to every receiver's buffer. When a *user_receive*

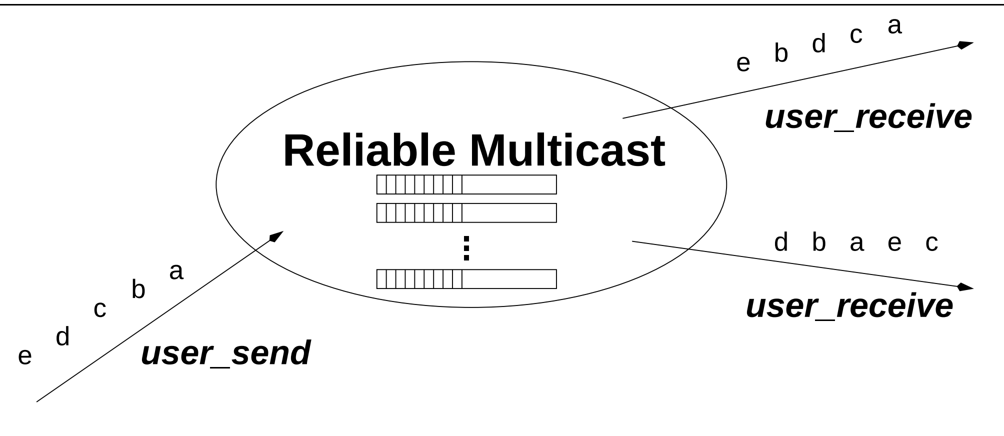

Figure 2-1: Reliable Multicast Specification

occurs, we remove any message from one of that receiver's buffers. The following I/O automaton specification [25, 24] captures this notion formally.

Reliable Multicast:

Signature:

Input:
$user\_send(m)_i, m \in M, 1 \leq i \leq n$

Output:
$user\_receive(m)_{j,i}, m \in M, 1 \leq i, j \leq n$

States:

for every $i, j \mid 1 \leq i, j \leq n$

$buffer(i, j)$, a multiset of elements of type M

Transitions:

$user\_send(m)_i$:
Effect:
for all $j, 1 \leq j \leq n$
$buffer(i, j) := buffer(i, j) \cup \{m\}$

$user\_receive(m)_{j,i}$
Precondition:
$m \in buffer(j, i)$
Effect:
$buffer(j, i) := buffer(j, i) - \{m\}$

Tasks:

Arbitrary

The specification above is insufficient without any liveness conditions. For instance, an implementation that delivers nothing satisfies the specification. Therefore, we impose the following liveness condition:

- If at any point there exists $m \in M$ such that $m \in buffer(i, j)$ for some i, j, $1 \leq i, j \leq n$, then there exists a $user\_receive(m)_{i,j}$ event at a later point of the execution.

This liveness condition guarantees eventual message delivery. With the above specification and the liveness condition, we have a formal notion of the desired behaviors of a reliable multicast. These behaviors give us the following properties of an execution of reliable multicast:

1. For each $user\_send(m)_i$ event, there exists a $user\_receive(m)_{j,i}$ event at a later time for all j, $1 \leq j \leq n$.

2. For each j, there exists an one-to-one and onto causality mapping between $user\_send(m)_i$ events and $user\_receive(m)_{j,i}$ events.

The first property, as a result of the liveness condition, states that every message is eventually delivered to all receivers. The second property, from the specification, guarantees that each receive event must correspond to a previous send event. Throughout this thesis, we will refer back to this specification when dealing with correctness.

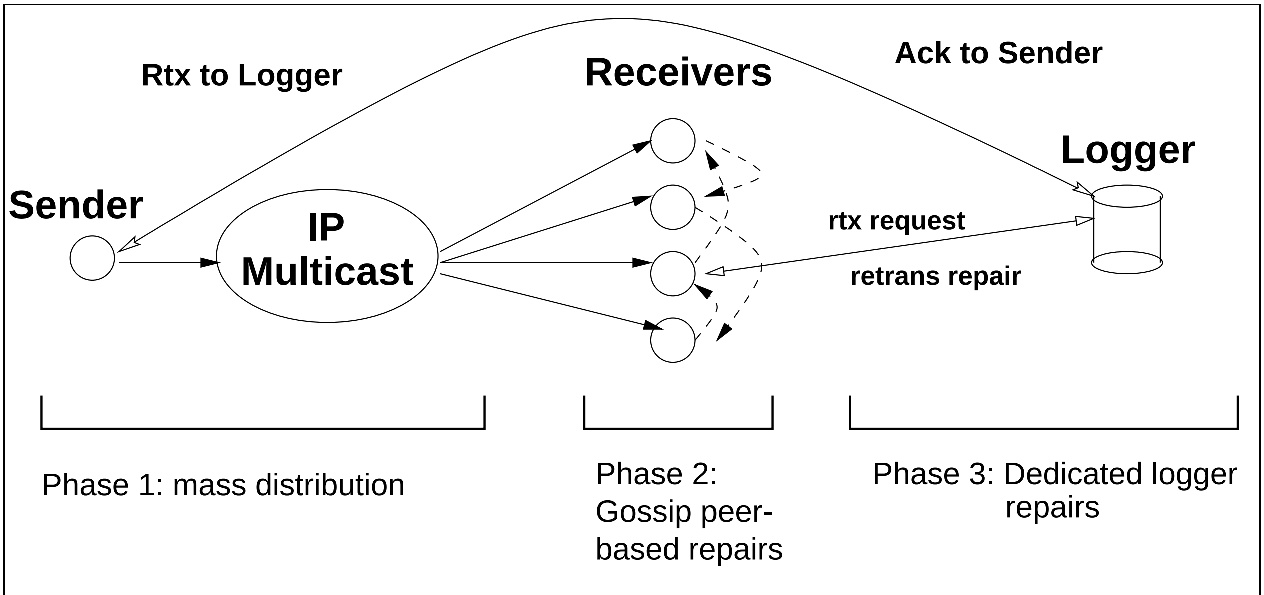

Figure 2-2: Reliable probabilistic multicast

2.2 Hybrid Protocol Overview

To implement reliable multicast, we need some members to keep a copy of a message on stable storage until everyone has received that message. Centralized approaches assign a dedicated member, either the sender or a logger, for storing a copy of the message and service retransmission requests. As noted in Section 1.2, this approach is limited by the dedicated member(s)'s service capacity. Peer-based approaches alleviate this service capacity limitation by allowing every participating member to act as a retransmission source. However, peer-based protocols typically introduce more network traffic. Moreover, these protocols relax message reliability guarantee to a probabilistic guarantee. In this thesis, we present a hybrid of these two approaches to achieve the reliability guarantee as defined in Section 2.1 and leverage many performance benefits of peer-based protocols.

Our hybrid approach, reliable probabilistic multicast (*rpbcast*), is a *receiver-reliable* [32] protocol. This means that each receiver independently detects missing messages and requests retransmissions. This approach is better than *sender-reliable* because of less network traffic, less workload on the sender, and flexibility for application End-to-End reliability. We achieve End-to-End reliability by allowing the application layer to decide whether our protocol should ask for a missing message. Though we call our protocol a multicast protocol, we only provide broadcast capabilities, hence the name *rpbcast*. We rely on the underlying unreliable multicast channel for sending messages to a subset of the entire group. Figure 2-2 shows how we combine the centralized approach with the peer-based approach.

Reliable multicast in *rpbcast* is divided into three phases. The first phase mass distributes a packet to all receivers through an unreliable multicast primitive such as IP multicast. The second phase repairs lost packets in a distributed manner using periodic gossips. If the previous two phases fail, then the last phase utilizes loggers for retransmission. Notice, phase 1 and phase 2 together are the building blocks of a peer-based recovery mechanism. Phase 1 and phase 3 together form a centralized recovery mechanism. By combining them as shown in Figure 2-2, where we first try peer-based repairs in phase 2 and then fall back to logger in phase 3, we migrate most of the logger service load in a centralized approach to the peer-based gossip phase. Moreover, we solve the reliability weakness of a peer-based approach by using loggers as a failsafe mechanism. Therefore, we get the best of both worlds: distributed recovery in peer-based approaches and reliability in centralized approach. Gossip and loggers are two components we chose to combine. A protocol designer can replace them with similar components. For instance, a designer might prefer using SRM instead of gossip for the peer-based component. We will discuss the intricacy of how to swap in and out components at the end of Chapter 4.

In designing our hybrid gossip and log-based multicast protocol, we emphasize

- guaranteeing reliable packet delivery to all receivers
- maintaining low delivery latency,
- balancing retransmission service load, and
- reducing network traffic, both per link and per node.

Chapter 3 will prove the reliability guarantee using loggers. Chapters 4 and 5 will introduce various techniques to achieve some of the above performance objectives. And Chapter 7 will give simulated performance results to validate our claims. Before we dive into the details, let us first discuss the underlying communication channel.

2.3 Specification of Unreliable Communication Channel

In our hybrid protocol, we rely on an unreliable communication channel for routing messages between senders and receivers. Moreover, we assume that the channel has the following characteristics:

1. The channel has both multicast and unicast capability.
2. The channel may drop any message.
3. The channel may duplicate finite copies of a message.
4. The channel *cannot* create bogus messages.

Since our protocol cannot control the implementation of the underlying communication channel, we assume the implementation satisfies the following specification of an unreliable channel.

Unreliable Communication Channel:

Signature:

Input:

$multisend(m)_i, m \in M, 1 \leq i \leq n$

$unisend(m)_{i,j}, m \in M, 1 \leq i, j \leq n$

Output:

$receive(m)_{j,i}, m \in M, 1 \leq i, j \leq n$

States:

for every $i, j \mid 1 \leq i, j \leq n$

$buffer(i, j)$, a multiset of elements of type M

Transitions:

$multisend(m)_i$:

Effect:

for all $j, 1 \leq j \leq n$

add a finite number (possibly 0) of m

to $buffer(i, j)$

$unisend(m)_{i,j}$:

Effect:

add a finite number (possibly 0) of m

to $buffer(i, j)$

$receive(m)_{j,i}$

Precondition:

$m \in buffer(j, i)$

Effect:

$buffer(j, i) := buffer(j, i) - \{m\}$

Tasks:

Arbitrary

Characteristics 1, 2 and 3 are true by the construction of the specification. Characteristic 4 is an property of the specification. Stating this more formally,

- *Property 2.1* At any given point of the execution, for each $m \in buffer(i, j)$ for some $1 \leq i, j \leq n$, sender i must have invoked $multisend(m)_i$ or $unisend(m)_{i,j}$ at an earlier time.

This property is true because elements can only be added to the buffers through *multisend* or *unisend*. Therefore, any message in the buffer must be a result of one of those actions. As in the case of *Reliable Multicast* specification, we also require some liveness guarantees for the unreliable communication channel. In this setting, we not only require that a message in the channel's buffer is eventually delivered, but also the condition that if there are infinitely many *multisend* or *unisend* events, then there will be infinitely many *receive* events. Since our channel only allows finite duplication, this guarantees that with sufficient retransmissions, some messages will get through. As we will see in the next chapter, the liveness property of our hybrid protocol depends on the liveness property of the underlying unreliable channel.

Chapter 3

Logger Based Recovery

In this chapter, we will describe and show the correctness of the logger based recovery component in our hybrid protocol *rpbcast*. Chapter 4 will build on top of this correctness proof to show that *rpbcast*, with its gossip component, is still correct.

The logger based recovery considered in this chapter is a simplified version of *LBRM*, without its variable heartbeats, statistical acknowledgments, or multicast retransmissions. Let us call this simplified version L. In protocol L, we have one single dedicated logger that handles all retransmissions and sending acknowledgments to the senders. Thus a sender is responsible only for initially multicasting a message, making sure the logger gets the message, and sending out heartbeats to facilitate loss message detection. With End-to-End reliability in mind, each individual receiver handles its own missing message detection, using feedback from the application layer, and issues requests for retransmission. This external application layer interface is not modeled here. Figure 3-1 shows the overall protocol setup and its relation to the specification in Section 2.1.

For clarity, we have separated sender modules and receiver modules in our discussion. In actual implementations, a node could have both sender and receiver modules. Moreover, we assume a fixed multicast group membership with a known logger, with known id $LOGGERID$, for protocol L to simplify the presentation and proof. We can extend L to include multiple loggers by using a set of logger ids instead of a single id. We defer membership details to chapter 5. For now, assume membership consists of every member with id between 1 and n.

Sections 3.1, 3.2, and 3.3 discuss sender, logger, and receiver implementations respectively. Useful invariants and properties that will assist in the correctness proof are also mentioned and argued in the three sections. Section 3.4 will give a simulation relation between L and the specification and prove that L indeed implements our desired specification. Section 3.5 argues about the liveness property of L. Parts of the work in this chapter have also been used for a 6.826 final group project[1]. I acknowledge my collaboration with Michael Feng and Victor Hernendez on those portions and would like to thank them for their contributions.

[1]Spring 2000, taught by Professor Bulter Lampson and Professor Martin Rinard

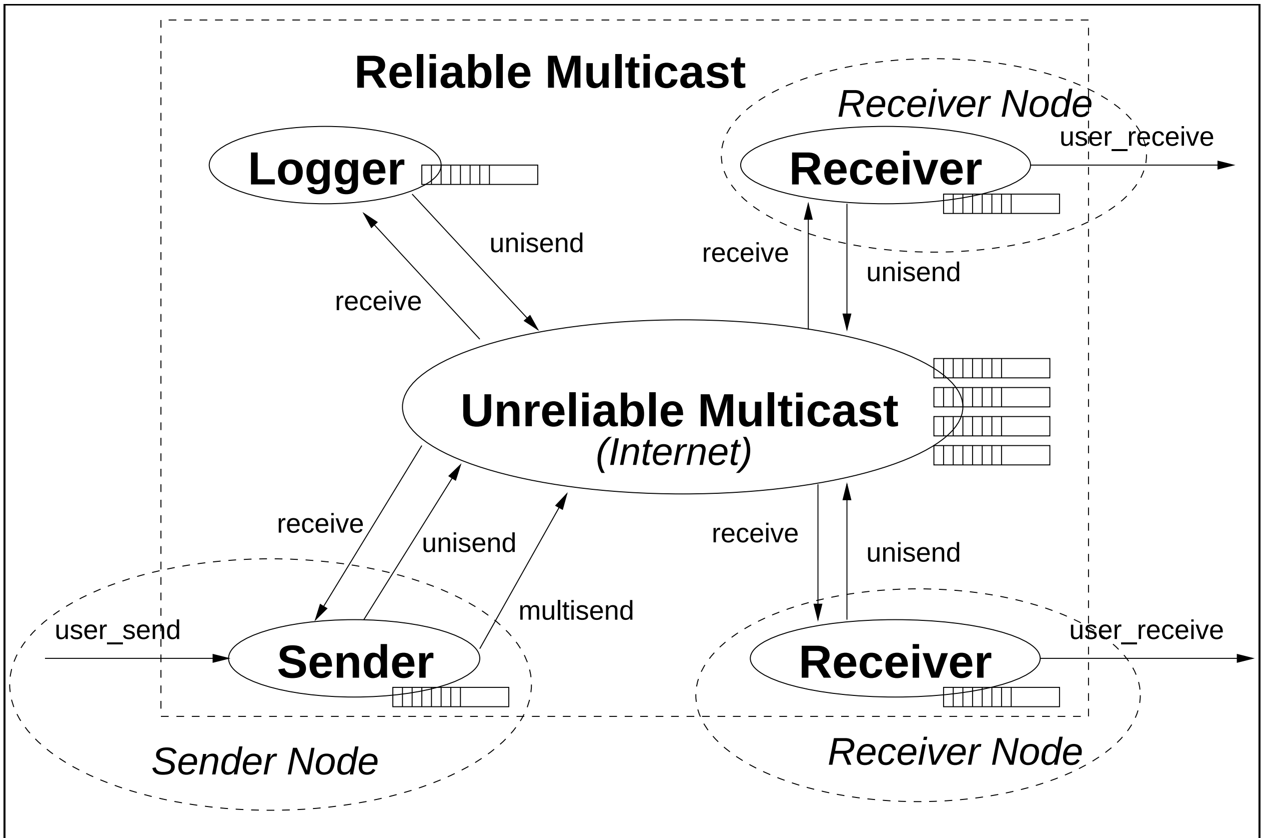

Figure 3-1: Logger based recovery

3.1 Sender Module

The main function of the sender module is to buffer messages until acknowledgments arrive from the logger. A secondary function is to periodically multicast heartbeats, latest sequence numbers, when idle. The heartbeats are necessary in order for the receivers to find out whether they are missing the last message before the idle period. The following is an I/O automaton description of the sender module.

$Sender_i$:

Types:

$ID = \{Int : senderid,\ Int : seqno\}$
$PAYLOAD = M \cup \{ACK, HEARTBEAT\}$

Signature:

Input:
$user\_send(m)_i, m \in M$
$receive(\{id, ACK\})_{j,i}, id \in ID, 1 \leq j \leq n$
$tick_i$

Output:
$unisend(\{id, m\})_{i,j}, id \in ID, m \in PAYLOAD, 1 \leq j \leq n$
$multisend(\{id, m\})_i, id \in ID, m \in PAYLOAD$

States:

seqno, integer sequence number for keeping track of the latest id, initially 0

sendbuf, a set of unacknowledged messages of type {id, M : m}, initially empty

unibuf, a set of outgoing messages of type {Int : to, {id, PAYLOAD : m}} through unisend, iniitally empty

multibuf, a set of outgoing messages of type {id, PAYLOAD : m} through multisend, initially empty

isidle, boolean flag for indicating idle status, initially TRUE

loggerid, identification of the dedicated logger. Set to default LOGGERID

Transitions:

$tick_i$:
  Effect:
    if $isidle$ then
      $multibuf := multibuf \cup$
        $\{ID\{i, seqno-1\}, HEARTBEAT\}$
    $isidle := TRUE$
    for each $entry \in sendbuf$
      $unibuf := unibuf \cup \{loggerid, entry\}$

$user\_send(m)_i$:
  Effect:
    $isidle := FALSE$
    $newid := ID\{i, seqno\}$
    $multibuf := multibuf \cup \{newid, m\}$
    $sendbuf := sendbuf \cup \{newid, m\}$
    $seqno := seqno + 1$

$unisend(\{id, m\})_{i,j}$:
  Precondition:
    $\{j, \{id, m\}\} \in unibuf$
  Effect:
    $unibuf := unibuf - \{j, \{id, m\}\}$

$multisend(\{id, m\})_{i,j}$:
  Precondition:
    $\{id, m\} \in multibuf$
  Effect:
    $multibuf := multibuf - \{id, m\}$

$receive(id, ACK)_{j,i}$:
  Precondition:
    $j = loggerid$
  Effect:
    if $\exists$ $entry = \{mid, m\} \in sendbuf$
      and $mid = id$ then
        $sendbuf := sendbuf - entry$

Tasks:

Arbitrary

$user\_send$ is called when the application wants to send a message m. The sender module assigns a unique sender-based sequence number to the message m and adds it to the multicast outgoing buffer and its own send buffer $sendbuf$. Message m will remain in $sendbuf$ until the sender module receives an acknowledgment through $receive(id, ACK)$ where id matches the unique sequence number assigned to m, denote it by $ID(m)$. Thus, two properties for the sender module are

- *Property 3.1* (Unique ID) If $user\_send(m)_i$ and $user\_send(m')$ are two distinct send events, then $ID(m) \neq ID(m')$.
- *Property 3.2* (Garbage Collection) At any point, message $m \in sendbuf_i$ if and only if there was a $user\_send(m)_i$ event and that no corresponding $receive(ID(m), ACK)_{loggerid,i}$ event has arrived yet.

$tick$ is our mechanism for doing periodic activities. In the case of the sender module, we will multicast a heartbeat if the sender is idle. We will also retransmit messages to the logger if no ACKs have came back yet.

3.2 Logger Module

The dedicated logger has two functions: 1) reply to senders after receiving their messages and 2) send repair messages to other receivers. To perform these two tasks, the logger maintains a log buffer that contains *all* known messages. Chapter 5 will discuss how we perform garbage collection on the log buffer. The following I/O automaton description formalizes the logger behavior, without any garbage collection.

$Logger_i$:

Types:

$ID = \{Int : senderid,\ Int : seqno\}$

$PAYLOAD = M \cup \{ACK, RTX\}$

Signature:

Input:

$receive(\{id, m\})_{j,i}, id \in ID, m \in M, 1 \leq j \leq n$

$receive(\{id, RTX\})_{j,i}, id \in ID, 1 \leq j \leq n$

Output:

$unisend(\{id, P\})_{i,j}, id \in ID, P \in PAYLOAD, 1 \leq j \leq n$

States:

$unibuf$, a set of outgoing messages of type {Int : to, {id, PAYLOAD : m}} through unisend, initially empty

$logbuf$, a set of messages of type {id, M : m}, initially empty

Transitions:

$unisend(\{id, P\})_{i,j}$:

Precondition:

$\{j, \{id, P\}\} \in unibuf$

Effect:

$unibuf := unibuf - \{j, \{id, P\}\}$

$receive(id, RTX)_{j,i}$:

Effect:

if $\exists \{id, m\} \in logbuf$ then

$unibuf := unibuf \cup \{j, \{id, m\}\}$

$receive(id, m)_{j,I}$:

Effect:

$unibuf := unibuf \cup \{j, \{id, ACK\}\}$

$logbuf := logbuf \cup \{id, m\}$

Tasks:

Arbitrary

The *receive* routines are event driven. If the logger receives a retransmission request (RTX), it looks up the message id in its $logbuf$ and sends a repair if the message is found. Notice that the logger does not queue RTX requests when the requested message is not in $logbuf$. We rely on each receiver to continue sending RTX requests periodically, as long as that receiver is interested in the message. This design decision gives flexibility to the end application in determining its own reliability criteria. The trade-off is the extra RTX requests.

The second type of messages a logger might receive are actual data messages from a sender. When a data message arrives, the logger adds to its buffer and returns an acknowledgment (ACK). Since ACKs could be lost, the logger might receive the same message multiple times. In order for the sender to garbage collect its *sendbuf*, the logger must generate an ACK each time, in this simple L protocol. Chapter 5 gives an alternative to this resend/ack approach by deriving acknowledgments from gossip.

3.3 Receiver Module

The receiver module, in the simple logger-based recovery, is responsible for delivering messages to the end application, detecting missing messages, and requesting retransmissions from the dedicated logger. We will omit the external interface of soliciting application layer's reliability criteria in the I/O automaton description below. This external interface should be called before deciding to request retransmission.

$Receiver_i$:

Types:

$ID = \{Int : senderid,\ Int : seqno\}$
$PAYLOAD = M \cup \{HEARTBEAT, RTX\}$

Signature:

Input:
$receive(\{id, m\})_{j,i}, id \in ID, m \in M, 1 \leq j \leq n$
$receive(\{id, HEARTBEAT\})_{j,i}, id \in ID, 1 \leq j \leq n$
$tick_i$

Output:
$unisend(\{id, P\})_{i,j}, id \in ID, P \in PAYLOAD, 1 \leq j \leq n$
$user\_receive(m)_{j,i}, m \in M, 1 \leq j \leq n$

States:

unibuf, a set of outgoing messages of type {Int : to, {id, PAYLOAD : m}} through unisend, initially empty
deliverbuf, a set of messages of type {id, M : m}, waiting to be delivered to the application, initially empty
lastseen[j], an array of last seen sequence number for each sender j, initially all -1
missing, a set of IDs indicating all currently missing messages with no duplicates, initially empty
loggerid, identifies the dedicated logger. Initially set to default LOGGERID

Transitions:

$user\_receive(m)_{j,i}$:
  Precondition:
    $\{j, m\} \in deliverbuf$
  Effect:
    $deliverbuf := deliverbuf - \{j, m\}$

$tick_i$:
  Effect:
    for each $entry \in missing$
      $unibuf := unibuf \cup \{loggerid, \{entry, RTX\}\}$

$receive(id, m)_{j,i}$:
  Effect:
    if $id \in missing$ then
      $deliverbuf := deliverbuf \cup \{id, m\}$
      $missing := missing - \{id\}$
    else if $id.seqno > lastseen[id.senderid]$ then
      for each k, $lastseen[id.senderid] < k < id.seqno$
        $missing := missing \cup \{id.senderid, k\}$
      $lastseen[id.senderid] = id.seqno$
      $deliverbuf := deliverbuf \cup \{id, m\}$

$unisend(\{id, P\})_{i,j}$:

Precondition:

$\{j, \{id, P\}\} \in unibuf$

Effect:

$unibuf := unibuf - \{j, \{id, P\}\}$

$receive(id, HEARTBEAT)_{j,i}$:

Effect:

for each k, $lastseen[id.senderid] < k \leq id.seqno$

$missing := missing \cup \{id.senderid, k\}$

$lastseen[id.senderid] =$

$\max\{id.seqno, lastseen[id.senderid]\}$

Tasks:

Arbitrary

---

user_receive is the external interface to the application. In each call, the application consumes one message from the *deliverbuf*. If *deliverbuf* is empty, then no *user_receive* action is possible. In practice, *user_receive* will block the application progress until something is available in *deliverbuf*.

The retransmission requests (RTX) are sent to the dedicated logger during periodic calls to *tick*. *tick* is responsible for consulting application reliability criteria before issuing RTX requests. In the above description, we have hidden this extra layer and forced the receiver module to always issue RTX requests for all known missing messages in the *tick* routine.

The main processing of the receiver module is done in *receive*. There are two kinds of possible messages coming out the communication channel. The first kind is a heartbeat message from a sender j. In this case, we use the heartbeat to detect new gaps in the sequence number for missing messages. Specifically, we first classify all messages with sequence numbers between $lastseen[j]$ and the heartbeat sequence number as missing. We then update the $lastseen[j]$. The other possible message type is data message. To ensure that we do not deliver a message multiple times, we only add the data to *deliverbuf* if the message was previous missing or we are seeing this message for the first time. In the latter case, we perform missing message detection similar to the heartbeat case.

There are one invariant and two properties that are useful for the correctness proof in the receiver module.

- *Invariant 3.3* Let D be the set of message ids in *deliverbuf*, then $D \cap missing = \emptyset$.
- *Property 3.4* For each message m, receiver i delivers m to the application *at most once*.
- *Property 3.5* Given a sender j. For each sequence number $k < lastseen[j]$, exactly one of the following three conditions is true:

    1. Receiver i already delivered message k to the application.
    2. Message k is in receiver i's *deliverbuf*.
    3. Id k is in receiver i's *missing* set.

Invariant 3.3 is true by construction. When a data message is added to *deliverbuf*, its id is always first removed from the *missing* set. As discussed previously, Property 3.4 holds because we never add a message

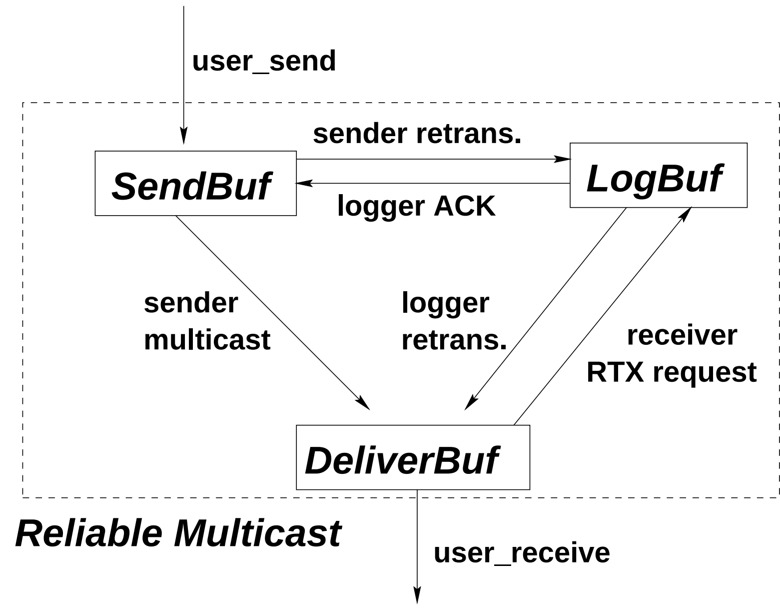

Figure 3-2: Data movement in the logger based recovery

to *deliverbuf* unless it is missing or new. To show Property 3.5 holds during any execution, we must show the three cases are disjoint and that the three cases cover all possible messages. We first apply Invariant 3.3 and Property 3.4 to show that the set of IDs in the three cases are indeed disjoint. Case 1 and 2 are disjoint because Property 3.4 states any message can only be delivered at most once — an ID in both case 1 and 2 contradicts Property 3.4. By Invariant 3.3, we also know that the *missing* set in case 3 is a disjoint set from both the delivered messages, case 1, and the waiting to be delivered messages, case 2. To show that the three cases covers all possible messages, we will show that when $lastseen[j]$ is updated, all the IDs in between will fall in either case 2 or case 3. By construction, $lastseen[j]$ is updated by either a heartbeat or a data message. In the case of heartbeats, all ids in the gap plus $lastseen[j]$ are added to *missing*, hence case 3 holds. If it was a data message, ids in the gap are added to *missing* (case 3), and $lastseen[j]$ is added to *deliverbuf* (case 2). Therefore, each message ID is accounted for in one of the three cases. Properties 3.4 and 3.5 are used in the next two sections for proving the safety and liveness conditions.

3.4 Safety

In this section, we will prove that the composition of the sender modules, the logger module, the receiver modules, and the unreliable communication channel satisfies the safety conditions of our specification of the reliable multicast in Section 2.1. We provide the liveness arguments in the next section.

To simplify the proof, consider the abstraction of the composed system as depicted in Figure 3-2. In this abstraction, senders collectively contribute to *sendbuf*. The dedicated logger maintains *logbuf*. And all the receivers form *deliverbuf*. The arrows in the figure indicate data movements between the various

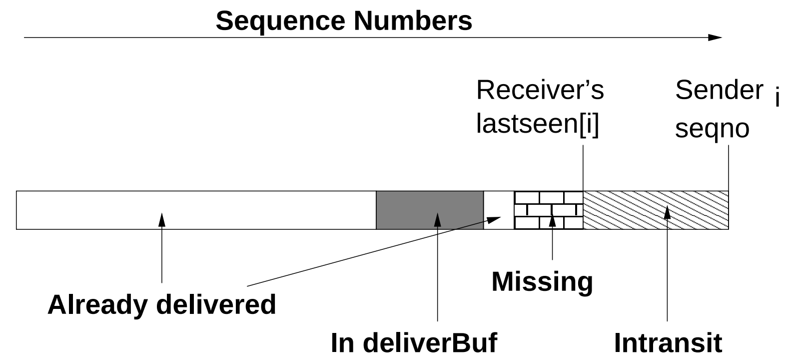

Figure 3-3: Where are the messages

components. This abstraction is valid because Property 2.1 ensures that there are no bogus messages in the unreliable communication channel. Therefore, data movements between the various buffers correspond to the composed transitions between different I/O automata in the system. Consequently, the correctness of this higher level abstraction is equivalent to the correctness of the actual implementation.

In this abstraction, messages are added to $sendbuf$ via $user\_send$ events. Once in $sendbuf$, messages are "moved" into receivers' $deliverbuf$ in two possible paths: directly through multicast or through logger retransmissions. The first path corresponds to successful initial multicast. The second path is the logger based recovery.

In order to formally define the simulation relation, we must find out where a message in the specification is located in our buffer abstraction. We will consider this question for each sender i and receiver j pair. From Property 3.5, we know that messages with ID lower than $lastseen_j[i]$ are divided into 3 cases at receiver j. However in the composed system, we have a fourth case: intransit — any messages with ID between $lastseen_j[i]$ and $seqno_i$. Let's denote the set of message IDs in this fourth case by $intransit(i, j)$. Figure 3-3 shows these four cases.

Since the logger module in Section 3.2 never garbage collects, we can always find a message either in $sendbuf$ or $logbuf$ depending on whether the sender has received an ACK or not. Let us denote the union of $sendbuf$ and $logbuf$ as $AllBuf$. Thus intuitively, we can always locate a copy of a missing message or a message in transit in $AllBuf$.

Armed with this intuition, we can define a simulation relation f that relates $sendbuf$, $logbuf$, and $deliverbuf$ in our higher level abstraction of the composed system to $buffer$ in the specification. The use of simulation relations for mapping between states in proving correctness is formally described in [25, 21, 28, 31, 20]. Let S be the state of the specification and I be the state of the implementation, then $(S, I) \in f$ if the following holds,

- For all m, $m \in S.buffer(i, j)$ if and only if $m \in I.deliverbuf_j$ or $m \in I.AllBuf \wedge (ID(m) \in I.missing_j \cup I.intransit(i, j))$.

This constraint essential states that for each undelivered message m in the specification's $buffer$, we can find m in the corresponding $deliverbuf$ or m is missing at the receiver and can be found in $AllBuf$. The constraint also states that the converse must be true as well. To show f is indeed a simulation relation, we need to show f maps initial states to initial states and all transitions preserve f (Theorem 8.12 in [24]). Initially, all the buffers are empty, the same as the specification. Therefore, f holds trivially for the initial case. Now let us consider individual transitions.

1. $\pi = user\_send(m)_i$.

   In the specification, one copy of m is added to each receiver's buffer. In the implementation, m is added to $sendbuf_i$ and $seqno_i$ is incremented. Since m has a new sequence number and no communication has been made, m is, by construction, in $intransit(i, j)$ for all receivers j. m is also in $AllBuf$ because $sendbuf \subseteq AllBuf$. Thus f still holds.

2. $\pi = user\_receive(m)_{i,j}$.

   In the implementation, message m is removed from $deliverbuf_j$. This removal corresponds to removing m from $buffer(i, j)$. Hence f also holds.

3. There are several internal actions in our composed implementation. These actions involve messages moving between buffers in the implementation. These actions should correspond to 0 steps in the specification and no changes in state. We need to consider three cases.

   (a) Message m moves from $sendbuf$ to $logbuf$.

   This case happens when logger ACKs message m. Since $AllBuf = sendbuf \cup logbuf$, f holds.

   (b) Message m arrives at $deliver_j$.

   This case happens when a missing or intransit message arrives at receiver j. After the arrival, the set $missing_j \cup intransit(ID(m).senderid, j)$ will no longer contain the message ID of m. But $deliverbuf_j$ will now include m. Thus f still holds.

   (c) Heartbeat from sender i arrives at receiver j.

   This case will cause message IDs move from $intransit(i, j)$ to $missing_j$. Since f considers $intransit(i, j) \cup missing_j$, the IDs movement does not effect f.

It follows then that our abstraction of the composed system satisfies the safety conditions of the specification. Therefore, the actual implementation also satisfies the specification.

3.5 Liveness

The one remaining detail is the liveness property of our implementation. We will give an informal argument here. There are two "eventual" properties that we must consider,

1. For each message m, each receiver will *eventually* learn the existence of $ID(m)$.

2. For each missing message m at some receiver j, receiver j will *eventually* recover m from the dedicated logger.

Property 1 essentially states that each receiver will eventually *detect* all the missing messages, either through heartbeats or a later message. This is true because during any *tick* period, a sender will either send a message or a heartbeat. Therefore, as long as the unreliable communication channel's liveness property guarantees some message will get through, we can detect all the missing messages.

Property 2 states that we will eventually *recover* all missing messages. This also holds by similar argument. Sender periodically retransmits a message to the logger until the logger ACKs. Hence a message will eventually arrive at $logbuf$. Now that each receiver also periodically solicits retransmission, logger's repair message will eventually get through and reach $deliverbuf$. Therefore, as long as we can detect missing messages, we can also recover them.

The combination of these two liveness properties guarantees eventual delivery of every message to each receiver. Using the at most once property (3.4), we get exactly once delivery and eventual delivery as required by the specification's safety and liveness properties.

Chapter 4

Combining with Gossip Based Recovery

In our *rpbcast*, we use gossip based recovery to reduce logger service load, for performance and scalability reasons. During each gossip round, a receiver, the *gossiper*, randomly selects a target member, the *gossip target*, in the same multicast group and exchanges information. In these exchanges, missing messages are repaired. There are two types of repairs: *gossiper-pull* and *gossiper-push*. Figure 4-1 illustrates the two cases.

In the gossiper-push mechanism, the gossip target first finds out which messages are available at the gossiper and then requests retransmission. In contrast, the gossiper-pull mechanism does exactly the opposite — the gossiper asks the gossip target for retransmissions blindly. Our hybrid protocol *rpbcast* uses the gossiper-pull mechanism because of lower network traffic, simpler gossip messages, and faster convergence. Gossiper-pull has lower network traffic because it only sends out two messages instead of the three sent in gossiper-push. Since gossiper is asking for retransmission, gossip messages only need to contain missing message IDs instead of the description of the gossiper's buffer. This results in simpler gossip messages. We will defer the discussion of convergence rate to Section 4.2.

In the remaining of the chapter, we will first give an I/O automaton implementation of a gossip based recovery using the gossiper-pull mechanism in Section 4.1. We then discuss probabilistic convergence issues in Section 4.2. We conclude the chapter in Section 4.3 by showing how to combine the gossip based recovery with the logger based recovery to create the hybrid protocol *rpbcast*.

4.1 Generic Gossip Implementation

In this section, we give a generic gossiper-pull based implementation. The receiver in the gossip based recovery is similar to the receiver module for logger based recovery in Section 3.3. Instead of contacting the logger

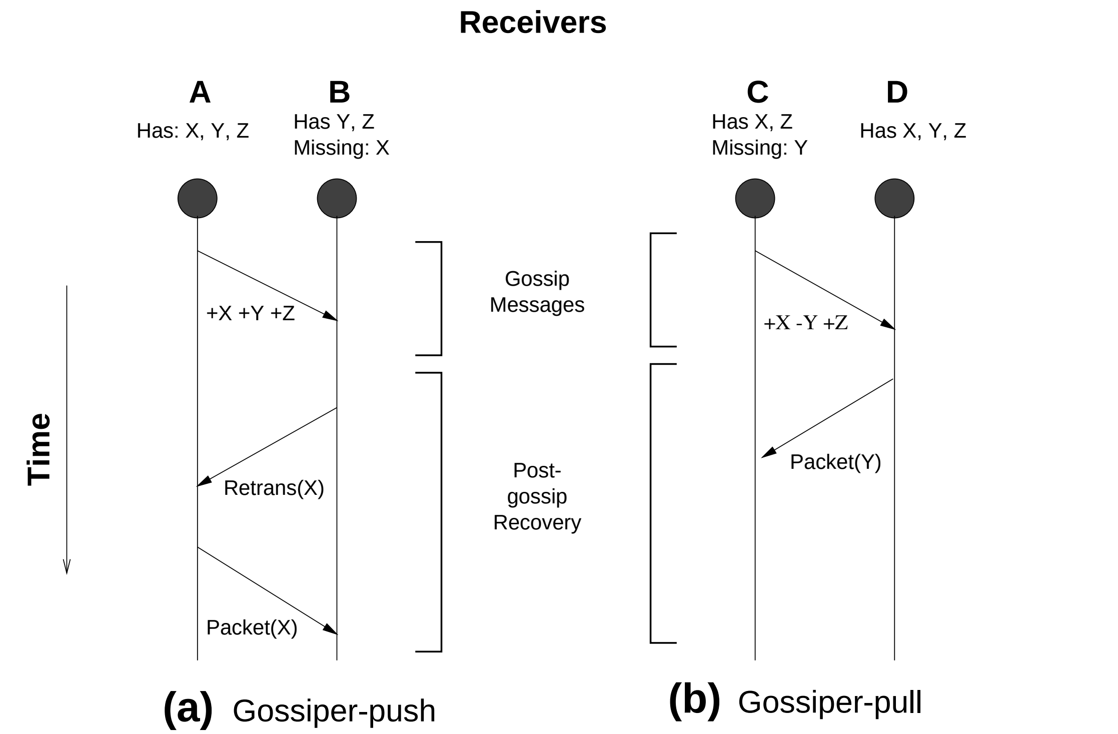

Figure 4-1: Gossip recovery examples: In (a), gossiper A pushes out a missing packet X to B. In (b), gossiper C pulls in a missing packet Y from D.

for retransmission during each *tick*, each individual receiver selects a random member and ask that member for retransmissions. In this implementation, we use an additional buffer $gossipbuf$ for buffering recent messages and servicing retransmission requests. Messages in $gossipbuf$ are garbage collected periodically, without any restrictions. Consequently, a retransmission request for m may never succeed if all other members have already garbage collected m. The I/O Automaton below describes the generic implementation. We highlight the changes and additions to the receiver module in bold.

$Receiver_i$:

Types:

$ID = \{Int : senderid,\ Int : seqno\}$

$PAYLOAD = M \cup \{HEARTBEAT, RTX\}$

Signature:

Input:

receive({id, RTX})$_{j,i}, id \in ID, 1 \leq j \leq n$

$receive(\{id, HEARTBEAT\})_{j,i}, id \in ID, 1 \leq j \leq n$

$receive(\{id, m\})_{j,i}, id \in ID, m \in M, 1 \leq j \leq n$

$tick_i$

Output:

$unisend(\{id, P\})_{i,j}, id \in ID, P \in PAYLOAD, 1 \leq j \leq n$

$user\_receive(m)_{j,i}, m \in M, 1 \leq j \leq n$

Internal:

gc({id, m})$_i, id \in ID, m \in M$

States:

unibuf, a set of outgoing messages of type {Int : to, {id, PAYLOAD : m}} through unisend, initially empty

deliverbuf, a set of messages of type {id, m}, waiting to be delivered to application, initially empty

lastseen[j], an array of last seen sequence number for each sender j, initially all -1

gmissing, a set of IDs indicating all currently missing messages, initially empty

gossipbuf, a set of messages of type {age, id, m}, initially empty

loggerid, identifies the dedicated logger. Defaults to LOGGERID

Transitions:

$unisend(\{id, m\})_{i,j}$:

Precondition:

$\{j, \{id, m\}\} \in unibuf$

Effect:

$unibuf := unibuf - \{j, \{id, m\}\}$

$user\_receive(m)_{j,i}$:

Precondition:

$\{j, m\} \in deliverbuf$

Effect:

$deliverbuf := deliverbuf - \{j, m\}$

$tick_i$:

Effect:

pick random member j | $1 \leq j \leq n$ **according to application specified selection distribution**

for each $entry \in$ **gmissing**

unibuf := unibuf $\cup$ **{j, {entry, RTX}}**

for each $entry = \{age, id, m\} \in$ **gossipbuf**

increment age

gc(id, m)$_i$:

Precondition:

$\exists$ **(age, id, m)** $\in$ **gossipbuf and age** $>$ **GCLIMIT**

Effect:

gossipbuf := gossipbuf - {age, id, m}

$receive(id, HEARTBEAT)_{j,i}$:

Effect:

for each $lastseen[id.senderid] < k \leq id.seqno$

gmissing := gmissing $\cup$ **{id.senderid, k}**

$lastseen[id.senderid] =$
$\max\{id.seqno, lastseen[id.senderid]\}$

$receive(id, m)_{j,i}$:

Effect:

if $id \in$ **gmissing** then

$deliverbuf := deliverbuf \cup \{j, m\}$

gossipbuf := gossipbuf $\cup$ **{0, id, m}**

gmissing := gmissing $-\{id\}$

else if $id.seqno > lastseen[id.senderid]$ then

for each $lastseen[id.senderid] < k < id.seqno$

gmissing := gmissing $\cup$ **{id.senderid, k}**

$lastseen[id.senderid] = id.seqno$

$deliverbuf := deliverbuf \cup \{j, m\}$

gossipbuf := gossipbuf $\cup$ **{0, id, m}**

receive(id, RTX)$_{j,i}$:

Effect:

if {id, m} $\in$ **gossipbuf then**

unibuf := unibuf $\cup$ **{j, {id, m}}**

Tasks:

Arbitrary

There are two changes to the module aside from adding buffers. The first change is that *receive* now expects and processes RTX requests. Specifically, *receive* looks up the requested message in $gossipbuf$ and sends the repair if the message is in the buffer. Note that $deliverbuf$ and $gossipbuf$ are managed separately. The second change is the age based garbage collection of $gossipbuf$. During each *tick*, we increase the age of each message in $gossipbuf$. The internal *gc* call then removes a message when its age exceeds some limit.

The implementation above uses gossiper-pull mechanism for recovering messages. Its gossip messages are the individual RTX requests sent to a random member. These RTX messages can be bundled into one single gossip message to reduce packet header overheads. We call these RTX gossip messages *negative gossips*, analogous to $NACK$s. The use of gossiper-pull and negative gossips distinguishes our gossip phase from *pbcast*[9]. *Pbcast* rely on gossiper-push mechanism. Thus gossip messages in *pbcast* must reflect gossiper's current buffer status. The use of negative gossip allows our hybrid protocol *rpbcast* to have smaller gossip message size. A second benefit of gossiper-pull is lower delivery latency because we can request retransmissions immediately upon detection through negative gossips instead of waiting until the next gossip round.

The effectiveness of a gossip based recovery is closely related to this garbage collection limit. For distributing retransmission service load, we would like to keep a message in $gossipbuf$ for as long as possible. The trade-off is the enormous memory requirements, especially at high send rates. Ozkasap et al. [29] proposes one remedy by selectively archiving a message for a longer duration than the garbage collection limit.

4.2 Convergence Rate

We measure the effectiveness of gossiping a message m by how many rounds of gossip are required until every receiver has m. We call this quantitative measure the *convergence rate*. Since gossip is a randomized process, convergence rate is probabilistic. Thus a convergence statement will have the form "message m converges in k rounds with probability p." In order for gossip based recovery to significantly reduce service load at loggers, we must pick $GCLIMIT$ such that after $GCLIMIT$ rounds, a message m converges with an extremely high probability. The particular values of $GCLIMIT$ and p depend on four parameters:

1. Success of the initial multicast — This parameter measures how many members in the current group received the initial multicast. This number may vary from 1 to n where n is the size of the group.

2. Message loss rate and distribution — This parameter characterizes message loss behaviors. For example, we may use uniform message loss distribution with a fixed loss probability (or loss rate) of ϵ for each message. Another common setting is bursty message loss in that the probability of a message being dropped is dependent upon the failures of previous messages.

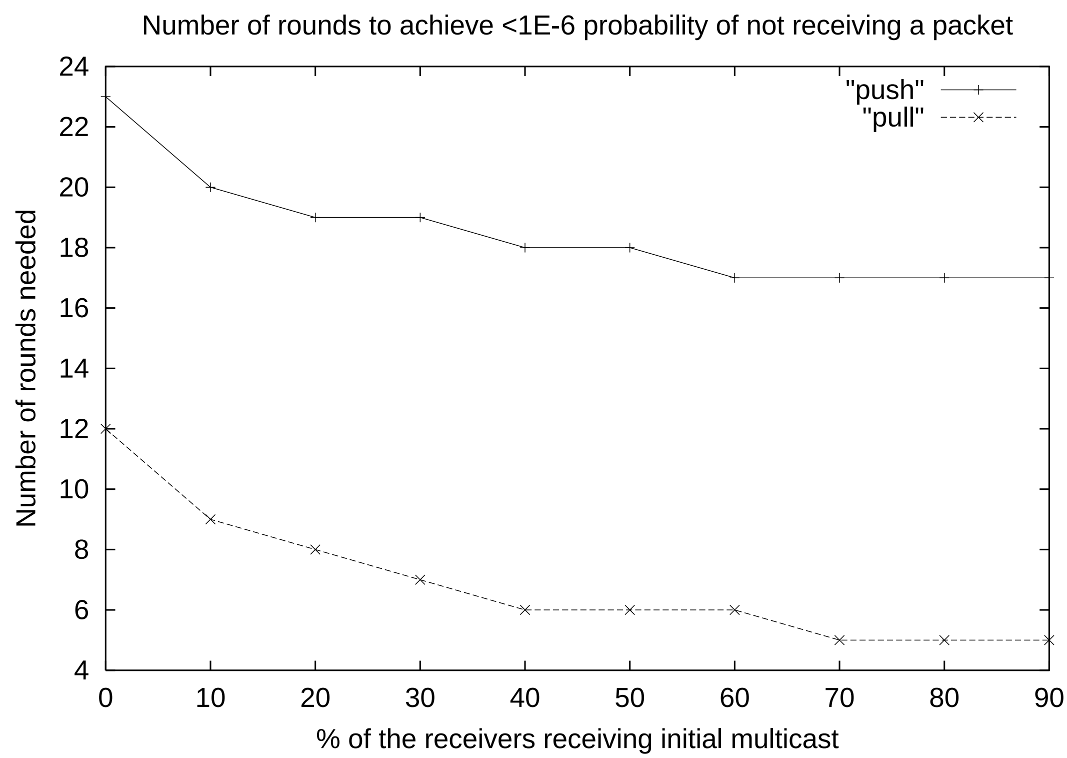

Figure 4-2: Convergence rate for 100 members, with no message loss

3. Gossiper-push or gossiper-pull mechanism — This parameter specifies which mechanism is used for gossiping.

4. Random gossip target selection distribution — this parameter determines how each member chooses whom to gossip to in the multicast group. Two examples are uniform selection distribution, where every member in the group is equally likely to be selected, or a biased distribution in favor of neighboring members, measured by the roundtrip latency delay.

Intuitively and qualitatively, if the initial multicast reaches most receivers, then the convergence rate will be quick with very high probability. Also, lower message loss rate will result in faster convergence. The effects of gossip mechanism and target selection is not immediately clear, even in the qualitative sense. Demers et al in [11] gave an intuitive argument for why gossiper-pull converges faster in database replication. The argument centers around the fact that when only a few receivers are without a message m, a pull recovery by a receiver i without m is more likely to succeed than other receivers with m pushing m out to receiver i. This observation also holds for our multicast. For the simple case of no gossip/repair message loss and uniform target selection (equally likely to pick any member), Figure 4-2 shows the number of rounds needed for 100 members to converge with error probability less than 10^{-6}, given that the initial multicast succeeds partially. From the graph, we see pull converges faster than push. We will describe how the graph is generated later in the section.

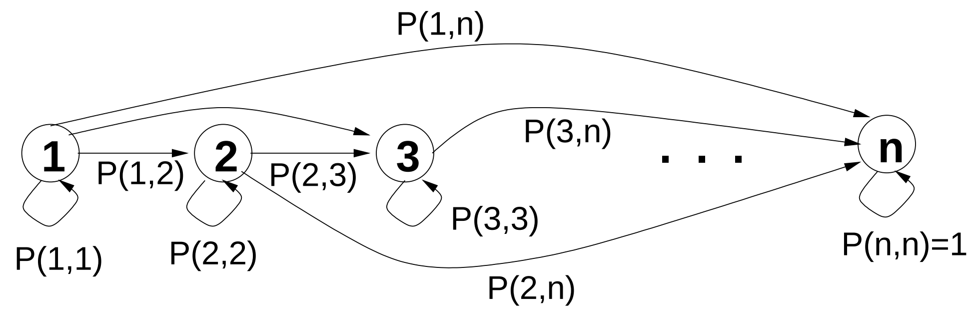

Figure 4-3: Model as a Markov Chain

The random gossip target selection is heavily dependent upon the network topology and the application needs. Generally, uniform selection results in faster convergence than biased selection. The trade-off is the extra network traffic. In practice, we observe that linear biasing in selection — probability of selecting the member drops linearly with the increase in roundtrip time — results in a good balance of convergence rate and network traffic. The simulation results in Section 7.5 suggest that latency grows with the increase in selection bias. On the other hand, network traffic through the center of the network drops with increasing bias.

As for many real world applications, simulation is the best method of isolating which parameters to adjust. For the remainder of the section, we will present two analytical approaches in determining the convergence probability. The first approach uses Markov chain to solve the simple case with uniform selection and gossiper-pull. Markov chain does not generalize nicely for the more complicated cases with biased selection or gossiper-push. For those cases, we use recurrence relation between successive rounds of gossip to calculate the convergence probability. In these discussions, we assume that every receiver contacts exactly one member during each round of gossip.

4.2.1 Markov Chain Approach

For a system with n receivers, we model it as an n states Markov chain. Let the k-th state of this n states chain represent k out of the n members have received the message. Thus, the associated transition probability $p_{i,j}$ is equivalent to one gossip round succeeding in increasing the total number of receivers with the message from i members to j members, for all $i \leq j$. Figure 4-3 shows the setup. The success of the initial multicast is model by the starting state of the Markov chain. For example starting in state 1 implies total failure of multicast while starting in state n implies complete success. Gossip/repair message losses are modeled by "adjusting" transition probabilities.

Given uniform target selection and gossiper-pull mechanism, we can compute individual transition probability $p_{i,j}$. In a simpler case, suppose no gossip/repair message losses occur, we get

$$p_{i,j} = \begin{cases} 1 & i = j = n \\ 0 & i < j \\ \binom{n-i}{j-i} \left(\frac{n-i}{n}\right)^{n-j} \left(\frac{i}{n}\right)^{j-i} & \text{otherwise} \end{cases}$$

In this Markov chain, we may transition from state i to j if and only if exactly $j - i$ members without the message contacted someone who has the message. This probability, as shown above, follows a binomial distribution.

Now that we have the transition probabilities, we can use existing tools for Markov chains to compute the expected number of transitions to reach state n (expected absorption time) or the probability of moving from state i to state j in k transitions (the k steps transition probabilities). Let $r_{i,j}(k)$ be the k steps transition probability from state i to state j, then the convergence statement, using this Markov chain model, says

- the message converges in k rounds with probability $r_{i,n}(k)$, where i is the initial success rate of the multicast.

We were not able to come up with a nice close form for $r_{i,n}(k)$ because the binomial distributions are different for different states. However, the expected number of transitions to reach state n is easily computed by solving a simultaneous system of n equations. With the help of a program for computing various probabilities for Markov chains, these computations for $r_{i,j}(k)$ should be straightforward.

4.2.2 Recurrence Approach

Markov chains are less manageable for the general case with non-uniform selection distribution or gossiper-push mechanism because we need 2^n states to denote exactly which members have a message. In the recurrence approach, we define $P_i(k)$ be the probability of receiver i have the message after k rounds of gossip. The base case of the recurrence is round 0, right after the initial multicast. We set $P_i(0)$ to be the success of the initial multicast. If the multicast completely failed, then $P_i(0) = 1$ for some i and $P_j(0) = 0$ for all $j \neq i$.

Once we have the base case of the recurrence, we compute $P_i(k)$ from $P_i(k-1)$. Informally, receiver i has the message after k rounds if receiver i has the message after $k - 1$ rounds or it got the message from some other member during round k. Let us denote the probability of receiver i get the message during round k by $get_i(k)$. Then,

$$P_i(k) = P_i(k-1) + (1 - P_i(k-1)) \cdot get_i(k) \quad \forall i, k \geq 1$$

The exact value of $get_i(k)$ depends on gossiper-pull or gossiper-push and the selection criteria. We model the selection criteria by “contact probabilities.” Let $C(i, j)$ be the probability of receiver i contacting receiver j. Then C specifies a valid set of contact probabilities if for all i, $\sum_j C(i, j) = 1$. With these contact probabilities, we compute $get_i(k)$ for the gossiper-pull as follows

$$get_i(k) = \sum_{j \neq i} C(i,j) \cdot P_j(k-1) \quad \forall i, k \geq 1$$

Intuitively, get_i succeeds if i contacts j and j already has the message. For gossiper-push, the computation is slightly more complicated. Since multiple gossipers may contact the same target, we do not want to count them multiple times. So we will compute the probability of receiver i not getting the message during round k instead and take the negation as $get_i(k)$. A receiver i will not get the message in a round if other receivers with the message all decided not to contact i. Thus, $get_i(k)$ for gossiper-push is

$$get_i(k) = 1 - \left[\prod_{j \neq i} (1 - P_j(k-1) \cdot C(j,i))\right] \quad \forall i, k \geq 1$$

With these $P_i(k)$ probabilities, the convergence statement says

- the message converges in k rounds with probability $\prod_i P_i(k)$, where $P_i(0)$ are the initial success probability of the multicast.

Note that the probabilities $P_i(k)$ are not probability mass functions (PMF). Instead, $P_i(k)$ for a fixed i form a cumulative probability distribution functions (CDF) over the number of rounds k. Therefore, given the gossip parameters and the initial condition, we can compute the CDF. From the CDFs, we can take the derivative, with respect to k, to find out the exact PMF of receiver i. Once we have the PMF, we can compute expected value, variance, or other useful numbers. However, the derived PMF is only valid for a particular set of parameters and initial conditions. To include possible message losses, we need to tweak the $get_i(k)$ probability appropriately. We used this recurrence formulation to generate Figure 4-2.

Using the recurrence approach, we can derive a crude lower bound on the number of rounds required to achieve a certain convergence probability for the simple case with uniform selection, gossiper-pull, and initial multicast reaching n_0 members. In this case, the approach gives us the following quadratic recurrence for members who did not receive the message initially.

$$P_k = \frac{n_0}{n} + \frac{2(n-n_0)-1}{n} P_{k-1} - \frac{n-n_0-1}{n} P_{k-1}^2$$

where $P_0 = 0$. We do not know how to solve this quadratic recurrence. However, note that $P_k \geq 1 - e^{-k}$ if $n_0 > \frac{1}{2}n$. Therefore, the number of rounds for $P_k < \delta$ is $k \geq -\log(\delta)$. The convergence probability for this case is $1 - P_k^{n-n_0}$. If $P_k < \delta$, then $1 - P_k^{n-n_0} < (n-n_0)\delta$. Therefore, bounding $1 - P_k^{n-n_0} < \epsilon$ is equivalent to bounding $P_k < \frac{\epsilon}{n-n_0}$. Thus to achieve convergence probability of ϵ with $n_0 > \frac{1}{2}n$, we get

$$k \geq -\log\left(\frac{\epsilon}{n-n_0}\right)$$

For the case with $n_0 < \frac{1}{2}n$, we can figure out how many rounds we need to reach half of the receivers. As it turns out, $P_k \geq \frac{n_0}{n}e^{k-1}$ while $P_k < \frac{1}{2}$. Thus it takes at most $1+\log(\frac{n}{2n_0})$ rounds to reach half the receivers. Combining these two halves, we arrive at the crude lower bound for achieving convergence probability of ϵ for a given n_0 as

$$k \geq 1 - \log\left(\frac{2n_0 \cdot \epsilon}{n(n-n_0)}\right)$$

4.3 Rpbcast - Integrating Logger/Gossip Based Recovery

Our hybrid protocol, *rpbcast*, uses the same sender and logger modules as described in Section 3.1 and 3.2. However, the receiver module in *rpbcast* is the combination in functionalities of the two receiver modules described in Section 3.3 and 4.1. Luckily, the two receiver modules have very little overlap in their functionalities. The only overlap is adding a newly arrived message to $deliverbuf$. Therefore, the receiver module in *rpbcast* simply combines the two receiver modules into one big module. The only necessary addition to the big module is a mechanism for moving missing message IDs from *gmissing* (missing ID set in the gossip based recovery) to *missing* (missing ID set in the logger based recovery). when those messages can no longer be recovered through gossip phase. There are two simple solutions. The first solution just moves message IDs from *gmissing* to *missing* if the IDs have been in *gmissing* for a “long” time. This solution can be implemented by a receiver without knowing anything about the rest of the system. The drawback is that we always pay this time delay while an ID remains in *gmissing*, even though gossip phase might have already failed to recover the message.

Our second solution uses more feedback from the gossip phase. Instead of using a timeout, we ask our gossip target to generate a *garbage collected notification* whenever the target has already garbage collected the message from its $gossipbuf$. This notification will alert the gossiper to move the missing message ID from *gmissing* to *missing* — in effect, send all future requests directly to a logger. Figure 4-4 illustrates the use of garbage collected notification.

When we gave an overview of our protocol in Section 2.2, we mentioned that a protocol designer can “swap” in SRM for the gossip based recovery if he or she desires. This swapping is possible because of the lack of overlap in functionality between the two types of receiver modules. Therefore, as long as missing message IDs are eventually moved from the peer-based recovery to logger based recovery, the resulting protocol will still function correctly. However, some of the optimizations in Chapter 5 are designed specifically for gossip based recovery, thus may not be applicable if SRM is swapped in.

We now give the I/O automaton description of the combined receiver module in *rpbcast* that utilizes both gossip and logger recovery with the garbage collection notification scheme. In this version, we grouped individual RTXs into one big message and send that message as our gossip.

Rpbcast_Receiver$_i$:

Types:

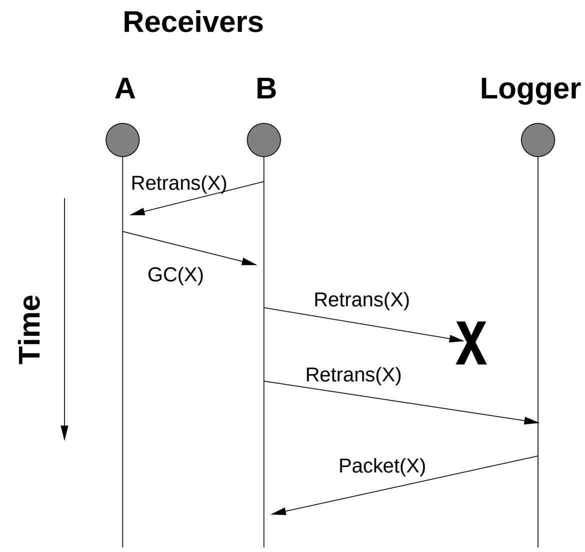

Figure 4-4: GC notification and repair example

$ID = \{Int : senderid,\ Int : seqno\}$

$IDList = array\ of\ IDs$

$PAYLOAD = M \cup \{HEARTBEAT, RTX, \{GOSSIP, IDList : rtxlist\}, GcNOTE\}$

Signature:

Input:

$receive(\{id, HEARTBEAT\})_{j,i}, id \in ID, 1 \leq j \leq n$

$receive(\{id, m\})_{j,i}, id \in ID, m \in M, 1 \leq j \leq n$

$receive(\{id, GcNOTE\})_{j,i}, id \in ID, 1 \leq j \leq n$

$receive(\{\{-1,-1\}, \{GOSSIP, rtxlist\}\})_{j,i}, 1 \leq j \leq n$

$tick_i$

Output:

$unisend(\{id, P\})_{i,j}, id \in ID, P \in PAYLOAD, 1 \leq j \leq n$

$user\_receive(m)_{j,i}, m \in M, 1 \leq j \leq n$

Internal:

$gc(\{id, m\})_i, id \in ID, m \in M$

States:

unibuf, a set of outgoing messages of type {Int : to, {id, PAYLOAD : m}} through unisend, initially empty

deliverbuf, a set of messages of type {id, m}, waiting to be delivered to the application, initially empty

lastseen[j], an array of last seen sequence number for each sender j, initially all -1

missing, a set of IDs indicating missing messages in logger phase, initially empty

gmissing, a set of IDs indicating missing messages in gossip phase, initially empty

gossipbuf, a set of messages of type {age, id, m}, initially empty

loggerid, identifies the dedicated logger. Defaults to LOGGERID

Transitions:

$unisend(\{id, m\})_{i,j}$:

Precondition:

$\{j, \{id, m\}\} \in unibuf$

Effect:

$unibuf := unibuf - \{j, \{id, m\}\}$

$user\_receive(m)_{j,i}$:

Precondition:

$\{j, m\} \in deliverbuf$

Effect:

$deliverbuf := deliverbuf - \{j, m\}$

$tick_i$:

Effect:

for each $entry \in missing$

$unibuf := unibuf \cup \{loggerid, \{entry, RTX\}\}$

pick random member j, $1 \leq j \leq n$ according to application specified selection distribution

$unibuf := unibuf \cup$

$\{j, \{\{-1, -1\}, \{GOSSIP, gmissing\}\}\}$

for each $entry = \{age, id, m\} \in gossipbuf$

increment age

$gc(id, m)_i$:

Precondition:

$\exists (age, id, m) \in gossipbuf$ and $age > GCLIMIT$

Effect:

$gossipbuf := gossipbuf - \{age, id, m\}$

$receive(id, HEARTBEAT)_{j,i}$:

Effect:

for each $lastseen[id.senderid] < k \leq id.seqno$

$gmissing := gmissing \cup \{id.senderid, k\}$

$lastseen[id.senderid] =$

$\max\{id.seqno, lastseen[id.senderid]\}$

$receive(id, GcNOTE)_{j,i}$:

Effect:

$gmissing := gmissing - \{id\}$

$missing := missing + \{id\}$

$receive(\{-1, -1\}, \{GOSSIP, rtxlist\})_{j,i}$:

Effect:

for each $id \in rtxlist$

if $\{id, m\} \in gossipbuf$ then

$unibuf := unibuf \cup \{j, \{id, m\}\}$

else if $\{id, m\} \notin (gmissing \cup missing)$ and $id.seqno \leq lastseen[id.senderid]$ then

$unibuf := unibuf \cup \{j, \{id, GcNOTE\}\}$

$receive(id, m)_{j,i}$:

Effect:

if $id \in (gmissing \cup missing)$ then

$deliverbuf := deliverbuf \cup \{id, m\}$

$gossipbuf := gossipbuf \cup \{0, id, m\}$

$gmissing := gmissing - \{id\}$

$missing := missing - \{id\}$

else if $id.seqno > lastseen[id.senderid]$ then

for each $lastseen[id.senderid] < k < id.seqno$

$gmissing := gmissing \cup \{id.senderid, k\}$

$lastseen[id.senderid] = id.seqno$

$deliverbuf := deliverbuf \cup \{id, m\}$

$gossipbuf := gossipbuf \cup \{0, id, m\}$

Tasks:

Arbitrary

In the combined implementation, whenever we detect a new missing message, we always add it to *gmissing* — initiating gossip recovery for that message. Missing message ID will migrate from *gmissing* to *missing*, ie. move from gossip phase to logger phase, after a garbage collection notification ($GcNOTE$).

We generate a $GcNOTE$ if and only if the requested message is not in our $gossipbuf$ and that we have received the message before. This $GcNOTE$ generation condition is true and sufficient for two reasons. First. Property 3.5 implies that we have received the message before, i.e. added to $gossipbuf$. Second, gc is the only call that removes messages from $gossipbuf$. Therefore, not in $gossipbuf$ implies local garbage collection has occurred.

To ensure that *rpbcast* still satisfies our specification and liveness, we need to augment our simulation relation f to include $gmissing$ as part of the missing message set. The remaining catch is to augment the liveness condition such that a message ID is eventually removed from $gmissing$:

- *Property 4.1* For each missing message m, m is either recovered during the gossip phase or eventually passed on to the logger phase recovery.

The first half of the claim is obvious from construction. To show the second half of the claim, we note that local garbage collection of m will eventually occur. In other words, there exists a time t in the execution such that $m \notin gossipbuf_i$ for all i and time after t. Therefore, after time t, periodic gossip messages for retransmission of m will eventually result in the arrival of a corresponding $GcNOTE$ message and removal of the ID from $gmissing$ as desired. Using Property 4.1 and the correctness of the logger based recovery shown in Chapter 3, we conclude *rpbcast* satisfies our specification and liveness condition in Section 2.1. At this point, we have completed the description of our hybrid protocol *rpbcast*. We will now procede to propose several “optimizations” and resolve some of the nagging details.

Chapter 5

Heartbeats, Membership, and Logger Garbage Collection

In this chapter, we cover details about logger garbage collection and two optimizations to our hybrid protocol *rpbcast*. We discuss two strategies for garbage collection (GC): timeout based and stability oriented GC. The two optimizations center around using gossip for distributing heartbeats and membership information. In the case of membership, we will also give a simple membership protocol that is efficient and sufficient for the purpose of logger GC. This chapter is organized in three sections. Section 5.1 talks about gossiping heartbeats. Section 5.2 describes our logger GC schemes. Section 5.3 covers various membership related issues.

5.1 Gossiping and Hashing Heartbeats

One drawback of all receiver-reliable protocols is the network traffic generated by the periodic heartbeat messages to inform all receivers about the latest message id during an idle period. In the vanilla *rpbcast* protocol described in the previous chapters, we multicast a heartbeat messages in each idle *tick* period. *LBRM* improves upon this fixed heartbeat scheme by a variable heartbeat [18] scheme using exponential back-off timer for generating heartbeats. The variable heartbeats work great when there are a few idle senders. However, this scheme will induce considerable overhead if the number of idle senders is large.

In a publisher/subscriber environment, we expect many senders to have bursty send behaviors — that is a sender outputs data in bursts and that there is a long idle period in between send bursts. Hence, these types of environment will often have many idle senders. This is the key motivation for propagating heartbeats through gossip. We will outline a naive approach first and then show how to optimize it.

In the naive approach, we include all the latest sequence numbers in every gossip message. Note that a gossip message that contains both the latest sequence numbers and negative gossips is equivalent to listing

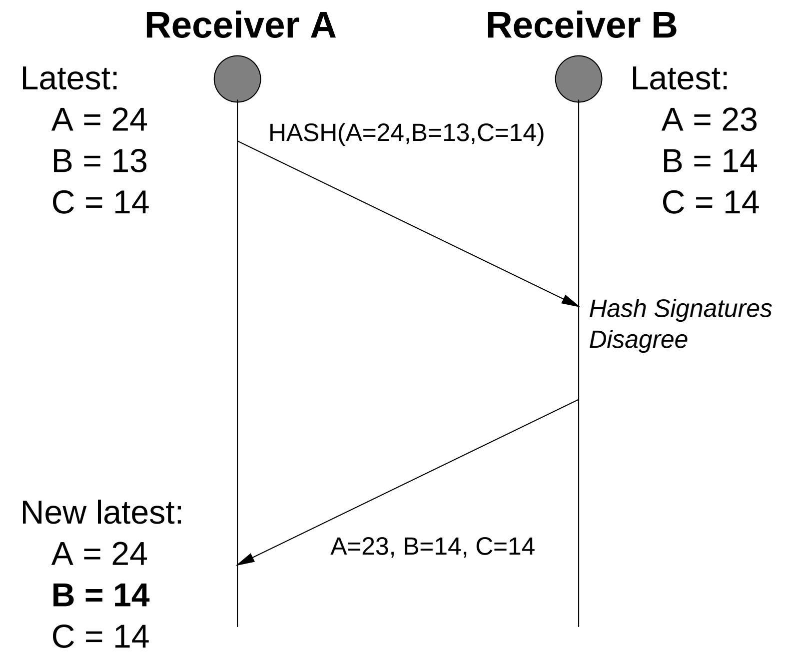

Figure 5-1: Hashing and merging heartbeat information

all previously received messages. Thus we lose the reduced message size benefit from negative gossips as claimed in Section 4.1. To improve upon the naive approach, we take advantage of that fact that we have N heartbeats in one gossip message as opposed to N separate heartbeat multicast messages in *LBRM*'s variable heartbeats scheme.

We realize this advantage by "compressing" N heartbeats into something smaller to reduce heartbeats overhead, which is not possible if each heartbeat is in a different message. More specifically, *rpbcast* compresses the heartbeats through hashing. Let HB_i be the last known sequence number from idle sender i, then *rpbcast* computes

$$hsig = HASH(\bigcup_i HB_i)$$

using some collision-free hash function $HASH$. Instead of gossiping all N heartbeats, *rpbcast* gossips the hash signature $hsig$. If the gossip target has the same signature, that is they agree on the latest sequence number with very high probability, then no further information exchange is necessary. On the other hand, if signatures differ, then the target will respond with a list of its heartbeats and let the gossiper merge the difference. Figure 5-1 demonstrates the interaction. Note that sending a hash signature and then merging the difference is essentially a gossiper-pull mechanism for finding out the latest sequence number. One additional optimization is for the gossiper to also update the target if the target is behind in some heartbeat values. This optimization is equivalent to having both pull and push in one gossip round.

Gossip with hashing as described above does not function well as the main heartbeat distribution mechanism: when a new heartbeat value appears at a particular sender, every member will see a different signature and cause a mass exchange of heartbeat values. To avoid this problem, *rpbcast* requires each sender to initially multicast its heartbeat value before entering an idle period. This initial multicast mass distributes the new heartbeat value. Consequently, the hashing optimization will only handle minor corrections. Thus the periodic heartbeat is replaced by one multicast and gossips. Another optimization is to use a variable gossip rate for hash signatures. This optimization reduces message traffic, thus lowers overhead.

The effectiveness of gossip with hashing for distributing heartbeat information depends on how often hash signatures disagree. Since a heartbeat value is unchanged during a particular idle period, disagreements can only occur immediately following a sender's transition from active to idle. If this active-to-idle transition rate and loss rate on the initial heartbeat multicast are high, then the hashing scheme will behave like the naive approach because disagreements result in full exchanges of heartbeat values. On the other hand, if these rates are low compared to the gossip period and that the number of idle senders is reasonably large, then the hashing optimization will result in significantly lower overhead than that incurred by *LBRM*'s variable heartbeats.

Since gossiping and hashing heartbeats are straight forward, we will not give the I/O automaton description of the receiver module here with this optimization. The correctness of *rpbcast* with the optimization still holds given that the liveness property guarantees eventually some of the heartbeats are delivered. This liveness condition is necessary to ensure that each receiver eventually detects all missing messages, as claimed and used in the liveness argument in Section 3.5.

5.2 Logger Garbage Collection

A simple GC mechanism uses timeouts. For example, one such scheme might be to garbage collect all messages older than 2 days in $logbuf$. Timeout based GC methods provide a deterministic mean of limiting $logbuf$ size and do not depend on any membership information. However, such methods destroys the reliable message delivery guarantee as specified in Section 2.1. For instance, the network might partition into multiple segments and remain partitioned for more than 2 days. In that case, members will not recover the message after the partitions heal because loggers have already performed garbaged collection. One remedy is to relax our specification such that we are allowed to drop messages that has been "stuck" for a while.

A more sophisticated GC mechanism, often used in group communication systems, is stability-oriented GC (S_{GC}). In S_{GC}, a logger garbage collects a message if and only if every receiver in the group, at the moment of the actual send, has acknowledged the message's arrival. Let J_i be the precise time when receiver i joined the group, then we define a garbage collection mechanism as *stability-oriented* if

- For all i, if m is sent after J_i, then m cannot be garbage collected without i's acknowledgment.

Many stability detection protocols exist in the research literature. However, they usually impose significant overhead in order to reduce detection time, especially with large groups and changing membership.

S_{GC} is also "incompatible" with receiver-reliable protocols because S_{GC} requires positive acknowledgments while receiver-reliable protocols use strictly negative acknowledgments.

In *rpbcast*, we amend this "incompatibility" between S_{GC} and receiver-reliability. Since all of our $NACK$s are sent in one gossip message (receiver module in Section 4.3), we can take the complement of the $NACK$s to derive the appropriate ACKs. However, we still have some ambiguities because we do not know the latest sequence numbers that the receiver has seen. If gossip messages also contain heartbeat information, as proposed in Section 5.1, then the ambiguities disappear. In practice, we may use the following simplified version — find the smallest missing message ID for a sender in the gossip message and garbage collect messages below that ID. If there are no missing messages for a particular sender, then we use the sequence number in the gossip message, if it is available.

This notion of derived ACKs may also be useful in reducing senders to logger retransmission traffic. Instead of constantly retransmitting a message until an ACK arrives, a sender can treat a logger as a normal receiver and derive ACKs from logger's gossip, if the initial ACK was lost. Under the assumption that message loss is low, say 0.5%, this alternative should not cause sender's buffer to fill up due to lost ACKs.

With these *derived* ACKs, our GC policy can be extremely sloppy. Below we give an I/O automaton description of our $logbuf$ garbage collection management.

$LoggerGC_i$:

Types:

$ID = \{Int : senderid,\ Int : seqno\}$

$SeqList = array\ of\ Ints$

$WaitList = array\ of\ Int$

Signature:

Input:

$new\_member(j)_i, 1 \leq j \leq n$

$remove\_member(j)_i, 1 \leq j \leq n$

$add\_message(id, m)_i, id \in ID, m \in M$

$gc(j, nackList, lastList)_i, j \in ID, nackList \in IDList, lastList \in SeqList$

Output:

States:

$current\_member$, an array of Int that reflects the current membership, initially {i}

$logbuf$, a set of elements of type {id, m, WaitList : wait}, initially empty

Transitions:

$new\_memeber(j)_i$:
  Effect:
    $current\_member := current\_member \cup \{j\}$
    for each $entry = \{id, m, wait\} \in logbuf$
      $wait := wait \cup \{j\}$

$remove\_member(j)_i$:
  Effect:
    $current\_member := current\_member - \{j\}$

$add\_message(id, m)_i$:
  Effect:
    $logbuf := logbuf \cup \{id, m, current\_member\}$

$gc(j, nackList, lastList)_i$:
  Effect:
    for each $entry = \{id, m, wait\} \in logbuf$
      such that $j \in wait$ do
      if $id \notin nackList$ and
        $id.seqno \leq lastList[id.senderid]$ then
        $wait := wait - \{j\}$
      if $wait = \emptyset$ then
        $logbuf := logbuf - entry$

Tasks:

Arbitrary

The *LoggerGC* mechanism described above has the following behavior. When the logger learns about a new member through *new_member*, the logger implicitly assumes that all messages currently in the buffer also require the new member's acknowledgment. Similarly, when a new message arrives, the logger assumes the message needs acknowledgments from everyone in *current_member*. *LoggerGC* is also sloppy in that $remove\_member(j)$ does not remove j from every message's *wait* list.

The interesting part of *LoggerGC* happens when there is a garbage collection *gc* call. In $gc(j, nackList, lastList)$, the logger finds all messages in *logbuf* that is waiting for acknowledgment from receiver j. For each of those messages, the logger makes sure that receiver j is not currently missing the message ($id \notin nackList$) and that receiver j has seen the message id ($id < lastList$). Basically, this check derives an acknowledgment from the $NACK$s and latest sequence numbers.

Note that $new\_member(j)$ is called when the logger learns about a new member j's presence. In the actual execution run, j might have joined the group at an earlier time. Let t_{join} be the moment j joins the group and t_{learn} be the moment that the logger learns about j's presence. Then, *LoggerGC* will correctly implement a stability-oriented GC as defined earlier if for each message m sent after t_{join}, one of the following two cases is true.

1. m arrives at the logger after time t_{learn}, or
2. there exists another receiver h such that GC of m depends on h and h has not sent an acknowledgment of m to the logger before time t_{learn}.

If case 1 happens, then the new member j has already been added to the *current_member* list when m arrives. Consequently, when *add_message* is called, member j will be added to m's waiting for acknowledgment list, as required by the definition of the stability-oriented GC. If case 2 happens, then m is already in $logbuf$ when the logger learns of member j's presence. Hence *new_member* will append j to the waiting for acknowledgment list of each message in $logbuf$, including m, which also satisfies the definition of the stability-oriented GC.

To show that *LoggerGC* is indeed a stability-oriented GC, we must show that either case 1 or case 2 is true for all affected messages. Unfortunately, we cannot prove this without cooperation from the membership protocol. Section 5.3 gives two "inexpensive" membership protocols that one can use to ensure case 2 always holds when a message arrives before the logger learns about a new member.

Due to the possibility of a member crashing and never returning, a practical implementation of *rpbcast* should have both timeout based and stability-oriented GC. One may argue that using a failure detector and stability-oriented GC is sufficient. This argument is true if the application does *not* have intermittently connected members. However, this assumption is not the case in publisher/subscriber systems. When there are intermittently connected members, a failure detector cannot distinguish between a crashed member and a temporarily disconnected member. Therefore, we will need a timeout based failsafe mechanism.

5.3 Approximate Membership

Rpbcast uses membership information for two purposes: selecting a gossip target and maintaining stability oriented garbage collection at loggers. For selecting gossip target, we do not need a precise and up-to-date membership all the time. As long as membership changes are eventually propagated to every member, *rpbcast* will function correctly. This observation suggests that gossip is an appropriate tool for distributing membership. In fact, an optimization, specific to *rpbcast*, may distribute membership information in the same manner as heartbeats. Current membership information is hashed and sent with each gossip message. When disagreements occur between hashed signatures, the full membership information is exchanged.

This lazy gossip approach to membership is not sufficient for logger garbage collection because loggers need to know precisely when a new member has joined the group. A simple solution is to let loggers handle all membership joins and leaves. As long as one logger knows about a join or a leave, the gossips will take care of the propagation. Again, an initial multicast of the joins and leaves will improve the gossip efficiency. This approach of using loggers as admittance control trivially satisfies case 2 set forth in Section 5.2 because $t_{join} = t_{learn}$ for at least one logger. Therefore, the simple approach is sufficient for implementing stability oriented garbage collection by the arguments in Section 5.2.

A weaker version of joins and leaves, partially based on [14], is also possible. Before we go into the details of this weaker membership, let us first define the notion of *time* in a more concrete way. Using the vector timestamp idea [12, 23, 26], we define time t by a vector of sender sequence numbers. We denote the sequence number for sender i at time t by $t[i]$. With the vector time, we can formally define the notion of *before* and *after*. We say

1. t_1 is *before* t_2 if $t_1[i] \leq t_2[i]$ for all $i \in t_1$.

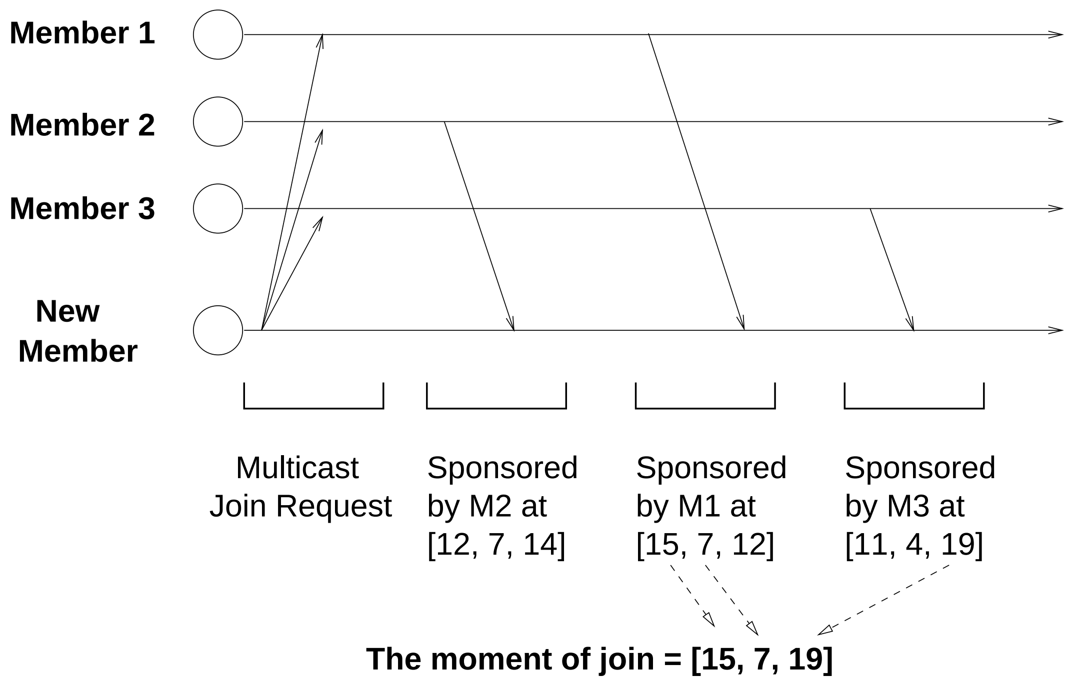

Figure 5-2: Membership: Joining

2. t_1 is *after* t_2 if $t_1[i] \geq t_2[i]$ for all $i \in t_2$.

3. Message m with ID $\{senderid, seqno\}$ is *after* t_1 if $seqno > t_1[senderid]$.

Note that *before* and *after* form only a partial order (rather than a total order) using these vector times. Fortunately, these vector times are sufficient for the purpose of determining when a new member has joined. We now give an informal description of what happens during a join. We will give an I/O Automaton later in the section.

During a join step, a new member W multicasts join messages periodically to the group until receiving k distinct gossip messages from members already in the group, i.e. from k *sponsors*. Note that each gossip message signifies that an existing member has accepted the join attempt and begun propagating the new membership information. If we have heartbeat values in the gossip message, as suggested in Section 5.1, then these sequence numbers in the gossip message define the precise moment in time when the sponsorship is established. We denote the sponsorship time from member i by TS_i. From the k sponsorship time, we define t_{join} as the earliest time such that t_{join} is *after* TS_i for all i. Clearly, $t_{join}[j] = \max\{TS_i[j]\}_{i=1}^{k}$. Figure 5-2 illustrates the process outlined above.

Intuitively, this join mechanism is correct because garbage collection of any message sent after a sponsor's acceptance of W will first require that sponsor's acknowledgment. Since the acknowledgment is derived from the sponsor's gossips to the loggers and gossip messages contain membership information, the loggers will learn about W's existence before garbage collecting, as required by case 2 in Section 5.2. Therefore, no messages after t_{join} will be garbage collected without W's acknowledgments. We use k sponsors to tolerate $k-1$ failures. Leave operations are identical to the join operations, except we do not have to compute t_{join}.

We now give the formal I/O automaton description of the module that distributes membership information through gossip and handles join/leaves. The hashing optimization can be applied to the distribution of membership as well. In order for this implementation to function correctly, we assume each member has access to a unique monotonic number generator. We use this unique number to differentiate between various incarnations of a member. In practice, we use the local clock time as the monotonic number generator. In the I/O Automaton, we model the number generator as a counter which increments before each join or leave attempt.

$Member_i$:

Types:

$STATUS = \{not\_member, member, pending\_join, pending\_leave\}$

$MemberElem = \{Int : id, STATUS : status, Int : freshness, Int : lastseqno\}$

$PAYLOAD = \{GOSSIP, set\ of\ MemberElem : members\} \cup \{JOIN, Int : freshness, Int : seqno\}$
$\cup \{LEAVE, Int : freshness\}$

Signature:

Input:

- $join_i$
- $leave_i$
- $tick_i$
- $receive(JOIN, freshness, seqno)_{j,i}$, freshness : Int, seqno : Int
- $receive(LEAVE, freshness, seqno)_{j,i}$, freshness : Int, seqno : Int
- $receive(GOSSIP, members)_{j,i}$, members : set of MemberElem

Output:

- $multisend(P)_i, P \in PAYLOAD$
- $unisend(j, P)_i, P \in PAYLOAD$

States:

$cmembers$, a set of MemberElem that keeps track of current membership, initially $\begin{cases} \{i, member, 0, -1\} & \text{if logger} \\ \emptyset & \text{otherwise} \end{cases}$

$state \in STATUS$, status of the member, initially $\begin{cases} member & \text{if logger} \\ not\_member & \text{otherwise} \end{cases}$

$t_{join}[i]$, an array of sequence numbers, used for determining when exactly did join succeed.

$fresh$, integer counter for uniquely identify each attempt, initially 0

$sponsors$, a set of Int, identifies all current sponsors

$multibuf$, a set of elements of type PAYLOAD, for outgoing messages

$unibuf$, a set of elements of type {to, PAYLOAD}, for outgoing messages

$loggerid$, the identifier of the logger, set to default LOGGERID

$seqnum$, sequence number of type Int. Used in the vector time stamp, initially -1

Transitions:

$unisend(j, P)_i$:

Precondition:

$\{j, P\} \in unibuf$

Effect:

$unibuf := unibuf - \{j, P\}$

$join_i$:

Precondition:

$status = not\_member \mid pending\_join$

Effect:

$fresh := fresh + 1$

$status := pending\_join$

$sponsors := \emptyset, cmembers := \emptyset$

$t_{join}[j] := -1,\ \forall 1 \leq j \leq n$

$leave_i$:

Precondition:

$status = member \mid pending\_leave$

Effect:

$fresh := fresh + 1$

$status := pending\_leave$

$sponsors := \emptyset$

$tick_i$:

Effect:

if $status = pending\_join$ then

$multibuf := multibuf \cup \{JOIN, fresh, seqnum\}$

else if $status = pending\_leave$ then

$multibuf := multibuf \cup \{LEAVE, fresh, seqnum\}$

else if $status = member$ then

pick random member $j, j \in cmembers$ and

that status of j is $member$

$unibuf := unibuf \cup \{j, \{GOSSIP, cmembers\}\}$

$multisend(P)_i$:

Precondition:

$P \in multibuf$

Effect:

$multibuf := multibuf - \{P\}$

$receive(JOIN, freshness, seqno)_{j,i}$:

Effect:

if $status = member$ then

merge $cmembers$ with

$\{j, member, freshness, seqno\}$

$receive(LEAVE, freshness, seqno)_{j,i}$:

Effect:

if $status = member$ then

merge $cmembers$ with

$\{j, not\_member, freshness, seqno\}$

$receive(GOSSIP, mem)_{j,i}$:

Effect:

merge $cmembers$ with mem

if $status = pending\_join$ and

$\{i, member, freshness, seqno\} \in mem$ and

$freshness = fresh$ then

$sponsors := sponsors \cup j$

update $t_{join}[i]$ if t_{join} is not after mem

if $sizeof(sponsors) \geq k$ or $j = loggerid$ then

$status := member$

else if $status = pending\_leave$ and

$\{i, not\_member, freshness, seqno\} \in mem$ and

$freshness = fresh$ then

$sponsors := sponsors \cup j$

if $sizeof(sponsors) \geq k$ or $j = loggerid$ then

$status := not\_member$

$cmembers := \emptyset$

Tasks:

Arbitrary

Basically, the member i enters a $pending\_join$ or $pending\_leave$ state through $join$ and $leave$ respectively. These two routines increment the $fresh$ counter so that old incarnations will not confuse us. $tick$ generates

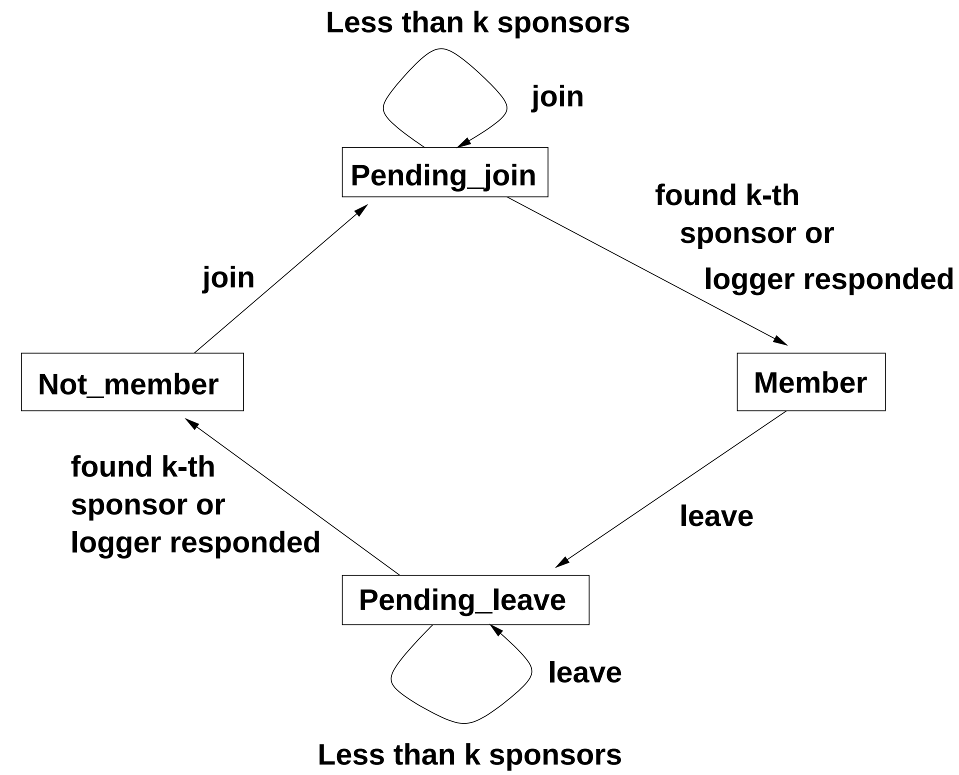

Figure 5-3: Membership: High level state transitions

periodic gossips and remulticast join/leave messages if in pending mode. The various *receive* routines are for processing join/leave requests and gossip messages. Note that we only allow current group members to process join/leave messages, i.e. pending members are not allowed to be sponsors. In *receive*, we merge the current group membership *cmembers* with the newly arrived ones. We merge them according to their freshness — honoring the most recently data (with largest freshness). When processing gossip, we also update sponsor information as well if we are in pending mode. Note that we leave the pending states if we get k sponsors or if we get a sponsorship from a logger. We use the shortcut from a logger for bootstrapping when the group size is less than k. Figure 5-5 shows more clearly how the various status states relate to each other and which actions are enabled at each state.

The property we need from this $Member$ implementation is

- *Property 5.2* If a "join" succeeds, then t_{join} is *after* TS_i for each sponsor i.

This property holds trivially if we maintain that $t_{join}[j] = \max\{TS_i[j]\}_{j=1}^{k}$. By induction on k, we can easily show that our incremental updates to $t_{join}[j]$ is correct. With the assumption that garbage collection derives acknowledgments from gossip messages that also contains membership information, we guarantee that loggers will learn about the new member's presence before garbage collecting any message m after TS_i for some sponsor i. By Property 5.2, t_{join} is after all TS_i. Therefore, no messages after t_{join} will be garbage collected before loggers learn about the new member (case 2 in Section 5.2). Hence, this membership protocol is sufficient for implementing stability-oriented garbage collection as described in Section 5.2.

The above $Member$ implementation can be easily extended to include intermittently connected members. We add one more possible state *disconnected* that a member can be in and two transitions *disconnect* and *reconnect*. *disconnect* changes a member's state from *member* to *disconnected*. *reconnect* does the opposite. We omit this extension here.

Chapter 6

Member Crashes

So far we have assumed that members, both senders and receivers, do not crash, or they have stable storage for keeping sequence numbers and buffers. In this chapter, we will show how to extend our specification and implementation to include sender/receiver crashes. We will continue to assume that loggers have stable storage for all its operations. Section 6.1 extends the specification in Section 2.1 to include sender/receiver crashes. Section 6.2 suggests several approaches in modifying the hybrid protocol to deal with receiver crashes and also explores situations with sender crashes.

6.1 Specification with Crashes

To allow member crashes, we will add two interface routines $crash_i$ and $recover_i$ in the specification. Informally, $crash_i$ will disable all *user_send* and *user_receive* events and enable $recover_i$. $recover_i$ does the opposite of $crash_i$. Moreover, the *crash* and *recover* are allowed to drop some messages. This dropping of messages is our relaxation of the specification for dealing with crashes. Below is the I/O automaton specification.

Reliable Multicast with Crashes:

Signature:

Input:

$user\_send(m)_i, m \in M, 1 \leq i \leq n$

$crash_i$

Output:

$user\_receive(m)_{j,i}, m \in M, 1 \leq i, j \leq n$

$recover_i$

Internal:

$drop(m)_{i,j}$

States:

$status(i)$ for $1 \leq i \leq n$, array of crash/recovery bits. initially, $status(i) := up$ for all i

for every $i, j, 1 \leq i, j \leq n$

$buffer(i, j)$, a multiset of elements of type {boolean, M}

Transitions:

$user\_send(m)_i$:
Precondition:
$status(i) = up$
Effect:
for all $j, 1 \leq j \leq n$
$buffer(i, j) := buffer(i, j) \cup \{FALSE, m\}$

$crash_i$:
Precondition:
$status(i) = up$
Effect:
$status(i) := down$
for each $1 \leq j \leq n$
mark some messages as $\{TRUE, m\}$
in $buffer(i, j)$

$user\_receive(m)_{j,i}$
Precondition:
$\{FALSE, m\} \in buffer(i, j)$
and $status(j) = up$
Effect:
$buffer(i, j) := buffer(i, j) - \{FALSE, m\}$

$recover_i$:
Precondition:
$status(i) = down$
Effect:
for each $1 \leq j \leq n$
mark some messages as $\{TRUE, m\}$
in $buffer(j, i)$
$status(i) = up$

$drop(m)_{i,j}$:
Precondition:
$\{TRUE, m\} \in buffer(i, j)$
Effect:
$buffer(i, j) := buffer(i, j) - \{TRUE, m\}$

Tasks:

Arbitrary

Essentially, all messages are initially marked as $FALSE$ when there are no crashes. When crash and recovery happen for member i, then some messages are marked as $TRUE$ — specifically, messages destined for receiver module i or messages sent by sender module i. Those messages marked as $TRUE$ are eventually dropped from the buffer by the internal action *drop*. This relaxation gives flexibility to the implementation for designing its recovery strategy. One would use backward simulations to show that such an implementation indeed satisfies the modified specification because the specification will usually make a nondeterministic decision on which messages are dropped before the actual implementation does.

6.2 Sender/Receiver Crashes

There are two possible approaches for dealing with receiver crashes in our implementation: either guarantee at most once delivery or at least once delivery. The specification in Section 6.1 is for an at most once delivery guarantee. The simplest way for doing at most once delivery is to rejoin the group from scratch and get a new starting point. Since our hybrid protocol is a receiver-reliable protocol, the protocol already implements at most once delivery in the event of a receiver crash. This approach is also flexible for application specific recovery. The alternative is at least once delivery. Our hybrid protocol can be easily modified to accommodate this approach as well. Instead of initiating a new join, the recovering member can contact a logger and figure out which messages it may not have been delivered to the application yet.

Sender crashes are more complicated because the sender may have difficulty in recovering the old sequence number, hence resulting in conflicting sequence numbers or a gap. Traditionally, protocols deal with this problem by forcing the sender to check out a block of sequence numbers before sending messages. This approach avoids conflicting sequence numbers when the sender crashes and recovers. However, it will result in a sequence number gap and cause receivers to think they are missing some messages. To use this approach in our hybrid protocol, we need to add extra logic into the data path for detecting gaps resulting from a sender crash. The easiest solution is to have notifications from the sender to loggers whenever a logger is requesting a message not in $sendbuf$. Loggers can then take over the responsibility of spreading that information to other receivers. We can also multicast these notifications to speed up the process.

The aforementioned candidate solutions for addressing sender/receiver crashes have all been implemented before, e.g. Belsnes's $FivePacketHandshake$ protocol [6]. We will skip their details in the interest of saving trees and move onto simulation results in the next chapter.

Chapter 7

Simulation Results

Our experiments focus on comparisons between gossip-based *pbcast*, log-base *LBRM*, and our hybrid protocol *rpbcast*. We implemented these three protocols as agents in UC Berkeley's Network Simulator NS2 using C++ and conducted test runs using NS2.

7.1 Experimental setup

Our test topology is a thirty node tier-hierarchy topology generated by Georgia Tech's Internetwork Topology Models (GT-ITM). Figure 7-1 shows the topology layout. The numbers on each link are the latencies for those links.

Of the thirty nodes, eighteen fringe nodes participate in our multicast experiments. Moreover, each receiver can also be a sender in our experiment. Node 14 on the far left side of network is the designated logger for both *LBRM* and *rpbcast*. Each link in the network is a 100Mb/sec bidirectional link.

Unfortunately, NS2 does not simulate packet servicing time at individual nodes. We approximate request service time by imposing a fixed limit on how many recovery requests a node can process in a second. In our experiments, we do not count multicast data packets toward this service limit because we are only interested in bounding overhead processing time. The default service limit is 1000 repair retransmissions per second. We also set the router queue limit to be 50 packets. In our test runs, we did not observe any packet loss due to queue overflow. Other protocol specific parameters are summarized in table 7.1.

In our experiments, each test run consists of 10 seconds of multicast traffic plus some additional lingering time to reliably deliver packets to every receiver. During this 10 seconds, each sender independently generates 1 kilobytes packets, with Poisson arrival rates. When we vary multicast rates in our experiments, we vary the expected number of packets generated by each sender.

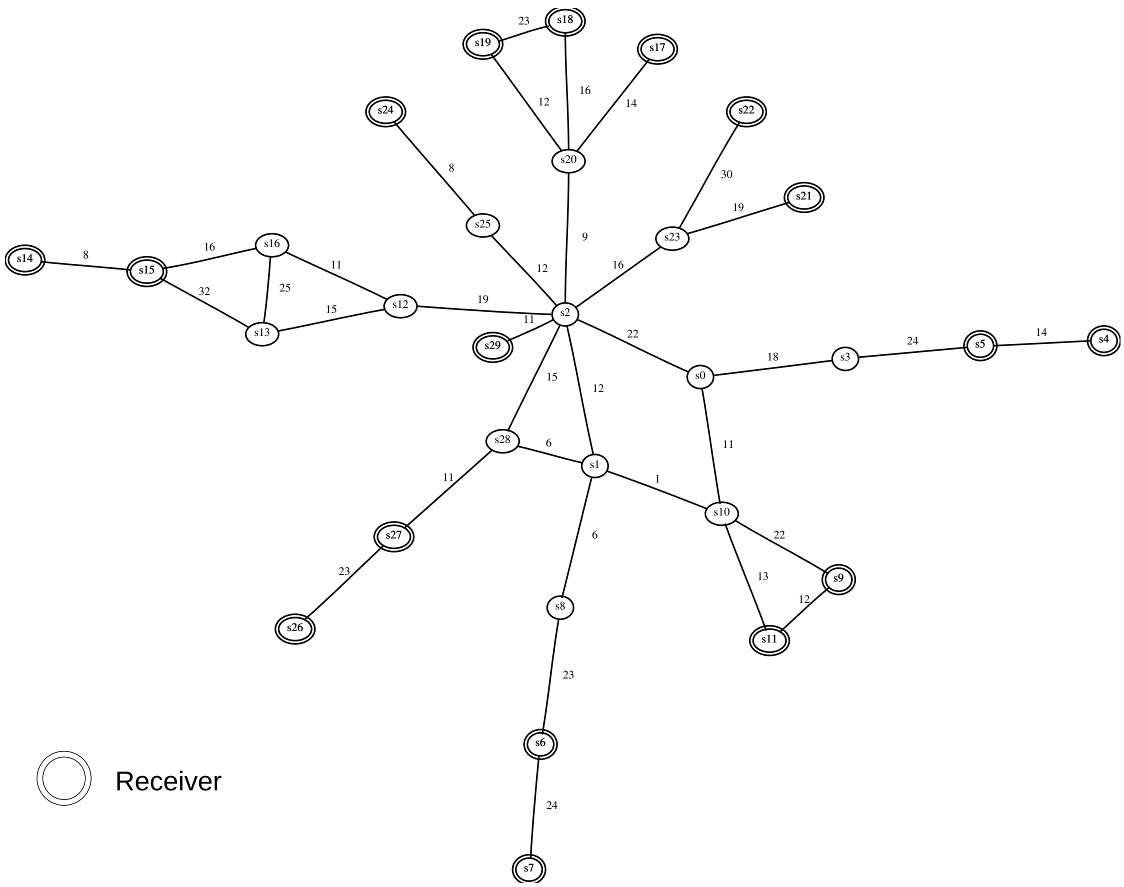

Figure 7-1: Test topology

Table 7.1: Protocol specific parameters

Parameter	Value	Protocols
Gossip period	0.25 sec	[r]pbcast
GC limit	10 rounds	[r]pbcast
Gossip selection	linear bias	[r]pbcast
Min heartbeat rate	0.25 sec	lbrm
Max heartbeat rate	32 sec	lbrm
Back off factor	2	lbrm
Retransmission rate	0.25 sec	lbrm

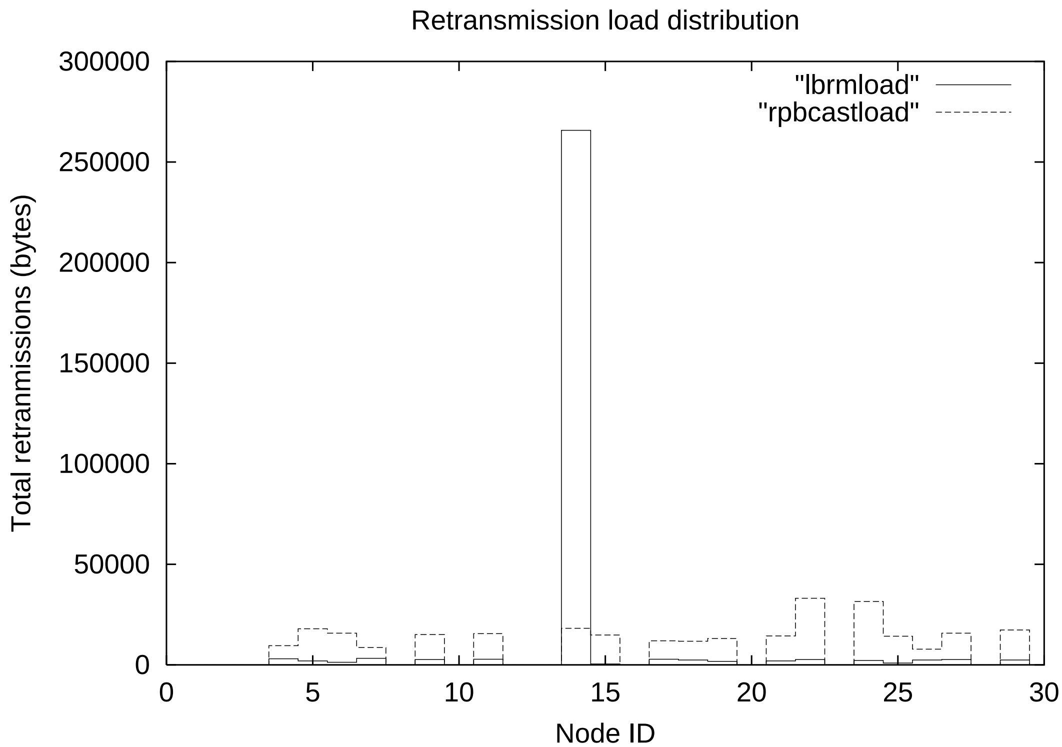

Figure 7-2: Load distribution

7.2 Retransmission load distribution and link utilization

The key advantage of gossip-based multicast over log-based multicast is balanced distribution of retransmission requests among all receivers. Figure 7-2 illustrates typical load distribution for *LBRM* and *rpbcast* when the network is not congested. This particular load distribution is for 18 senders with 1% packet loss and 360 packets per second overall.

As expected, the logger in *LBRM*, node 14, has significantly more retransmissions than any receivers in *rpbcast*. The retransmission traffic from non-logger nodes in *LBRM* are packets that did not reach the logger during the initial multicast, and are therefore retransmitted by the sender. *Rpbcast*'s balanced load distribution also results in better link utilization. Figure 7-3 shows the overall traffic per link. Notice the four peaks in the figure for *LBRM*. These four links are the bottleneck links from the logger to the hub in the center of the network. In practice, we expect large link bandwidth between a logger and the network backbone, thus higher link usage may not be as significant as logger service rate.

Balanced load distribution in *rpbcast* is not free. In order for every receiver to act as a retransmission source, each receiver has to buffer a packet for some time. Thus a receiver in *rpbcast* has higher memory requirements than *LBRM*. For our experiments, we buffer each packet for 10 gossip rounds. Thus each node buffers all packets that arrived in the past 2.5 seconds. In practice, we suggest a fixed buffer size and garbage collecting oldest packets when space is needed. This approach will affect convergence time. However, under

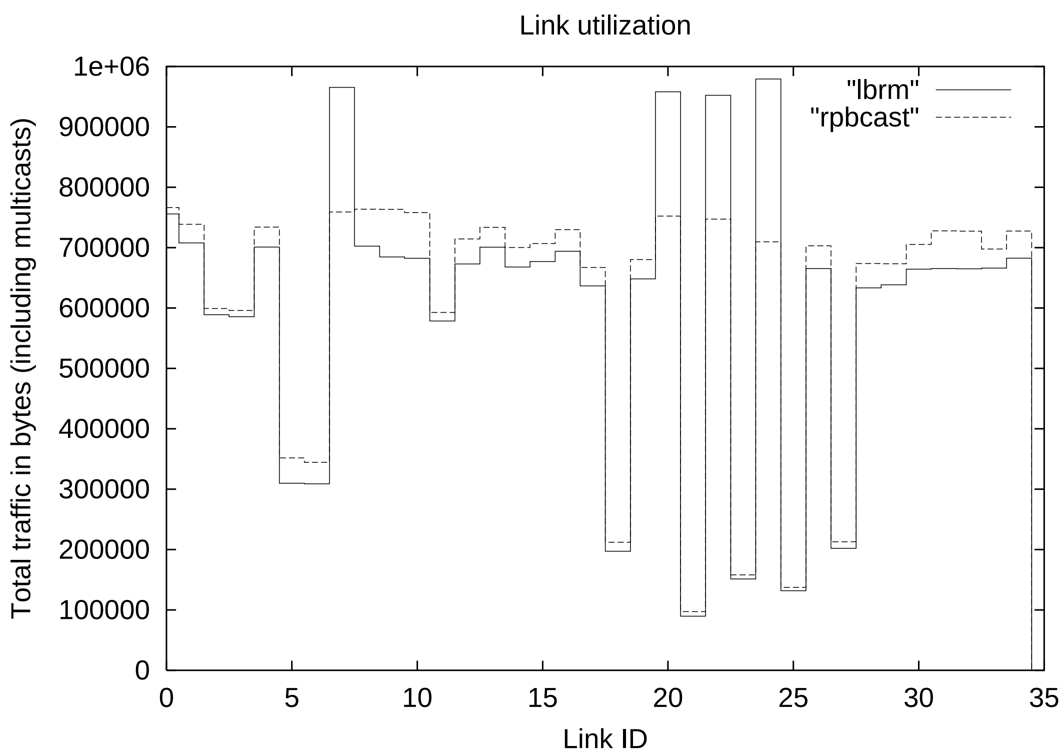

Figure 7-3: Link traffic

average network conditions, two or three rounds of gossiping is usually sufficient for delivering a packet to all receivers. In the worse case, receivers will resort back to contacting the logger. In particular *rpbcast* has the same behavior as *LBRM* when the network saturates due to an extended period of external noise.

7.3 Delivery Latency

The motivation for distributing retransmission requests among all receivers is to maintain low delivery latency under high send rates. We define delivery latency as the time between a sender multicasting a packet and all receivers receiving the packet. With high send rates, the number of missing packets will also increase, thus overloading dedicated loggers and introducing higher latency. Figure 7-4 illustrates this behavior.

In this experiment, we set packet size to 1 kilobyte, loss rate at 1%, and maximum retransmission limit at 1000 requests per second. The most interesting aspect of the latency figure is the cross-over point between *LBRM* and *rpbcast*. Before the cross-over, the logger in *LBRM* is not overloaded. Hence *LBRM* is able to provide timely retransmissions, whereas *rpbcast* is randomly selecting retransmission sources, some of which do not succeed. After the cross-over point, the logger at node 14 cannot keep up with retransmission requests, resulting in higher latency. This bottleneck does not exist in *rpbcast* until a much higher send rate, when gossips fail to service all retransmissions before garbage collection. If we stretch the plot farther out to the point where the network saturates, both *LBRM* and *rpbcast* will have large latencies. These latency

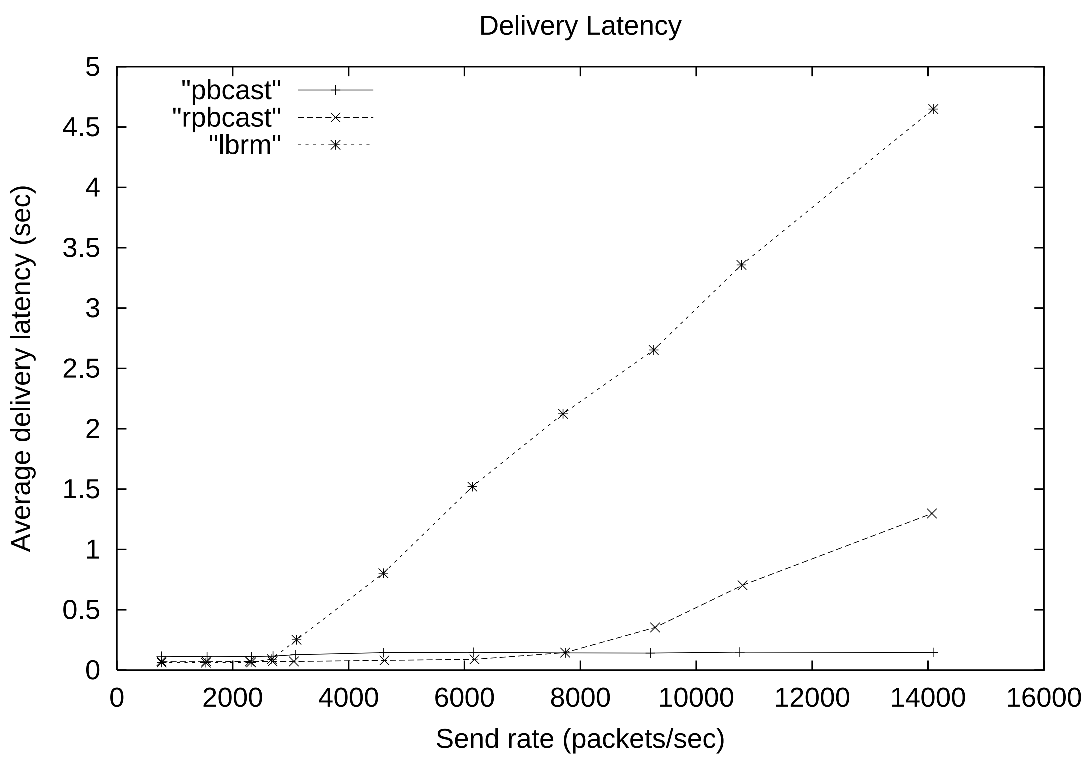

Figure 7-4: Latency vs send rate

results suggest that a further optimization would be to use *LBRM* for low send rate and switch to *rpbcast* after detecting logger congestion.

One may also notice that *rpbcast* has lower latency than *pbcast*. The second cross over point is between *rpbcast* and *pbcast*. This result supports our claim that pull-based recovery exhibits lower latency than push-based recovery. Another point to mention is that, although *pbcast* has constant latency, it does not deliver packets to all receivers at high send rates. Figure 7-5 shows the numbers of packets, sent during this 10 seconds trial, not received by all receivers.

7.4 Non-repair related overhead

Another important scalability factor is message overhead. Since each protocol must repair roughly equal number of packets, we separate retransmission packets from other protocol-specific overhead. In non-repair related overhead, we measure the amount of non-multicast and non-repair packets. This overhead for *pbcast* includes gossip messages and retransmission requests. In our implementation of *pbcast*, we use an interval representation for gossiping buffer content instead of listing each packet in the buffer. We chose the interval representation because high multicast rates will result in very large gossip messages if we simply list individual packets. In the worst case where we miss every other packet, an interval representation is twice as large as explicitly listing packets. For *LBRM*, the overhead messages consist of acknowledgments, periodic

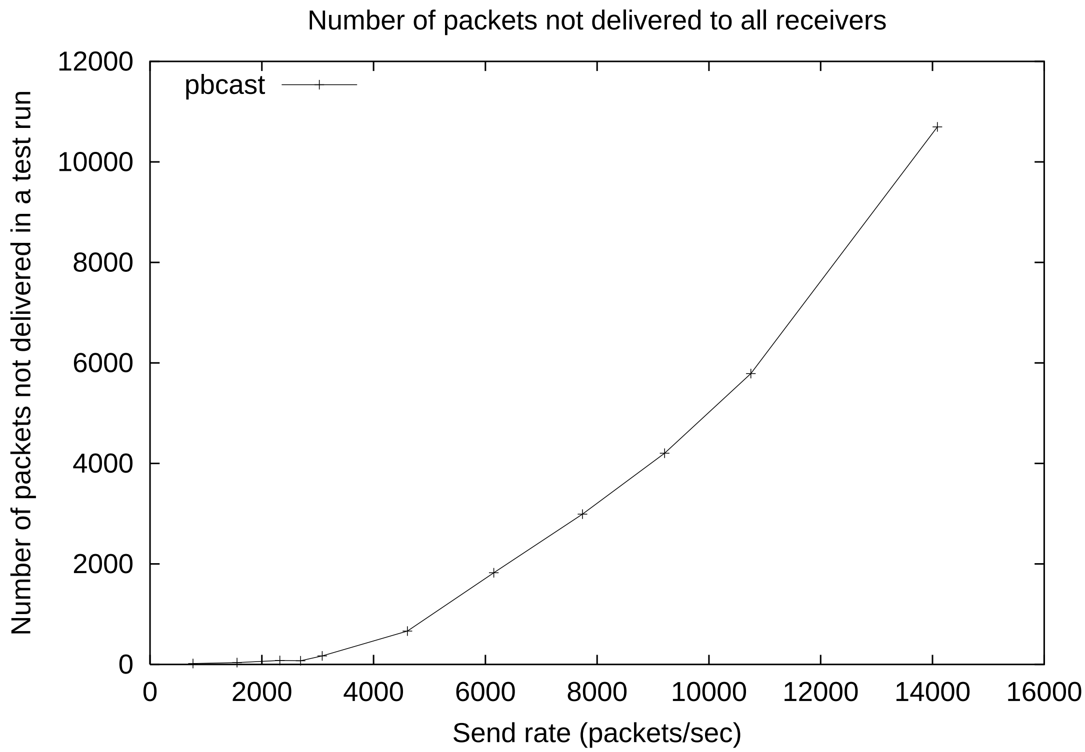

Figure 7-5: Unreliable packets in pbcast

heartbeats, and retransmission requests. *Rpbcast* overhead consists of gossips, garbage collected notifications, and acknowledgments.

Three variables contribute to the variations in protocol overhead: number of senders, packet loss rate, and multicast rate. We present two sets of overhead measurements by varying the number of senders and the packet loss rate. Since a higher multicast rate simply results in more dropped packets, we omit that measurement here. Figure 7-6 shows changes in protocol overhead as we vary number of senders from a single sender to all 18 senders. In these test runs, the loss rate is 1%, and multicast traffic rate is fixed at approximately 360 total packets per second.

Note that because of the positive gossips, overhead for *pbcast* is approximately linear with the number of senders. On the other hand, *LBRM* and our *rpbcast* are insensitive to the number of senders. The constant difference in overhead between *rpbcast* and *LBRM* is due to additional information in *rpbcast*'s gossip messages, such as hashing signatures. If an application also requires membership information, then overhead for *LBRM* will increase due to an additional membership protocol while *rpbcast* overhead already includes membership overhead. Observe that when every node is sending packets (18 senders), no idle heartbeats are generated. Consequently, *rpbcast* overhead decreases by half because hash signatures for heartbeats are no longer needed.

A similar experiment was conducted with loss rates ranging from 0.1% to 5%. Figure 7-7 shows the increase in protocol overhead as loss rate increases. The growth in overhead in all three protocols is dominated

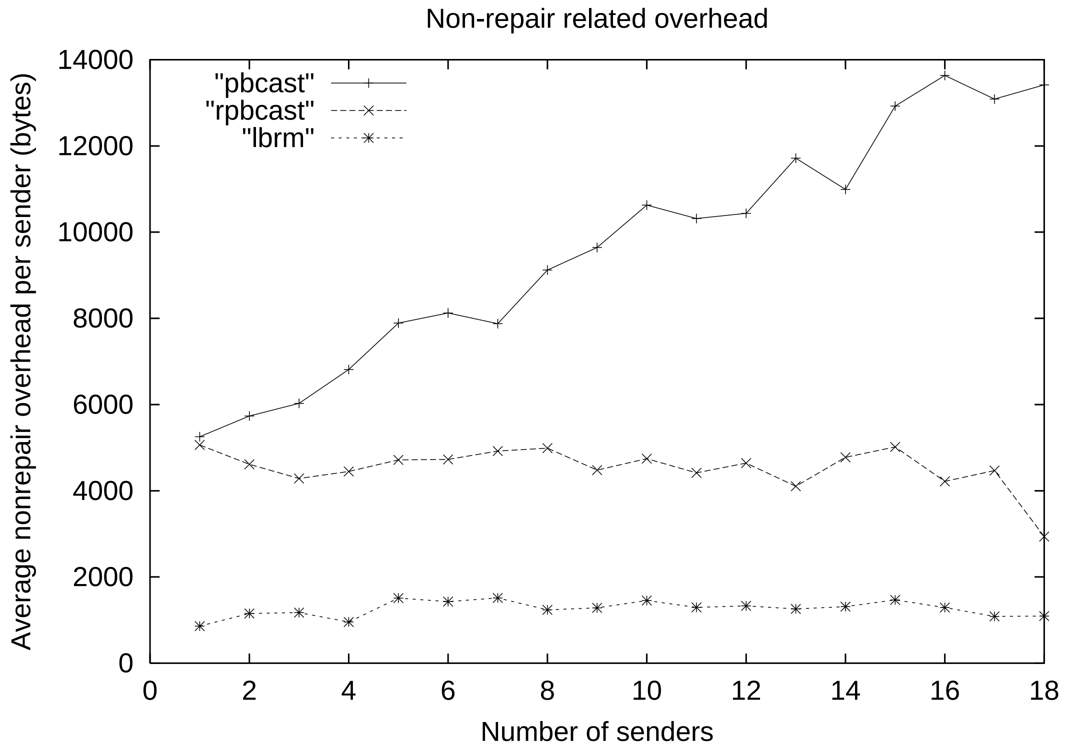

Figure 7-6: Varying number of senders

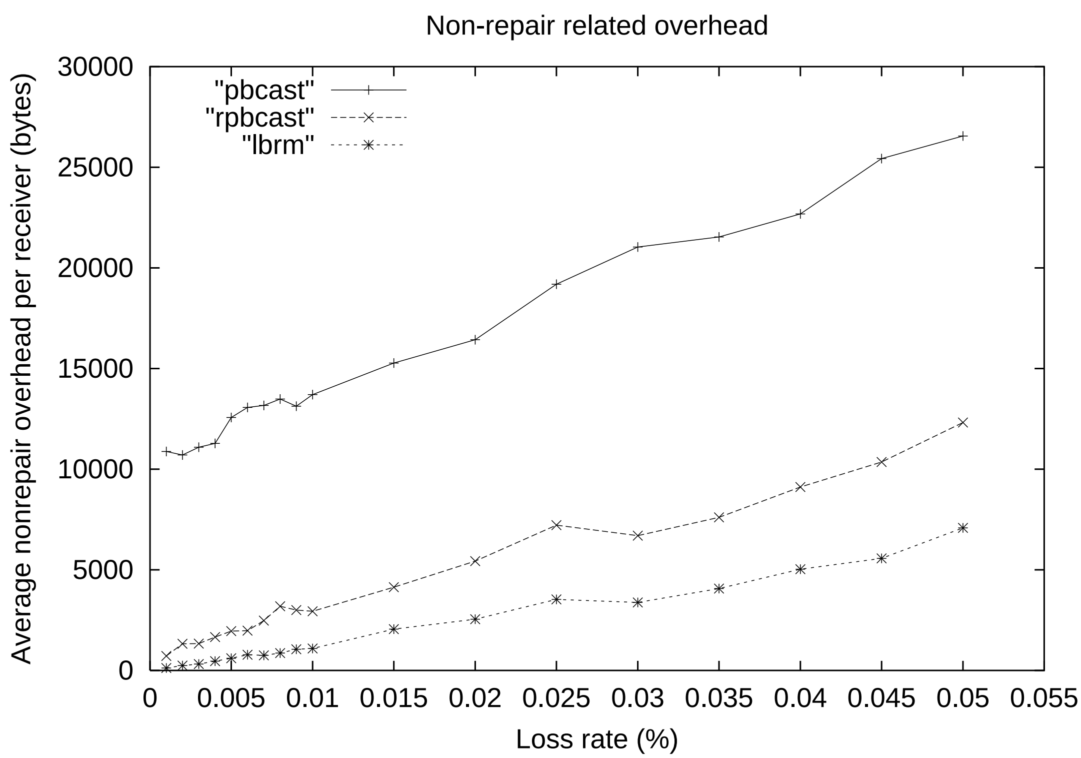

Figure 7-7: Varying loss rate

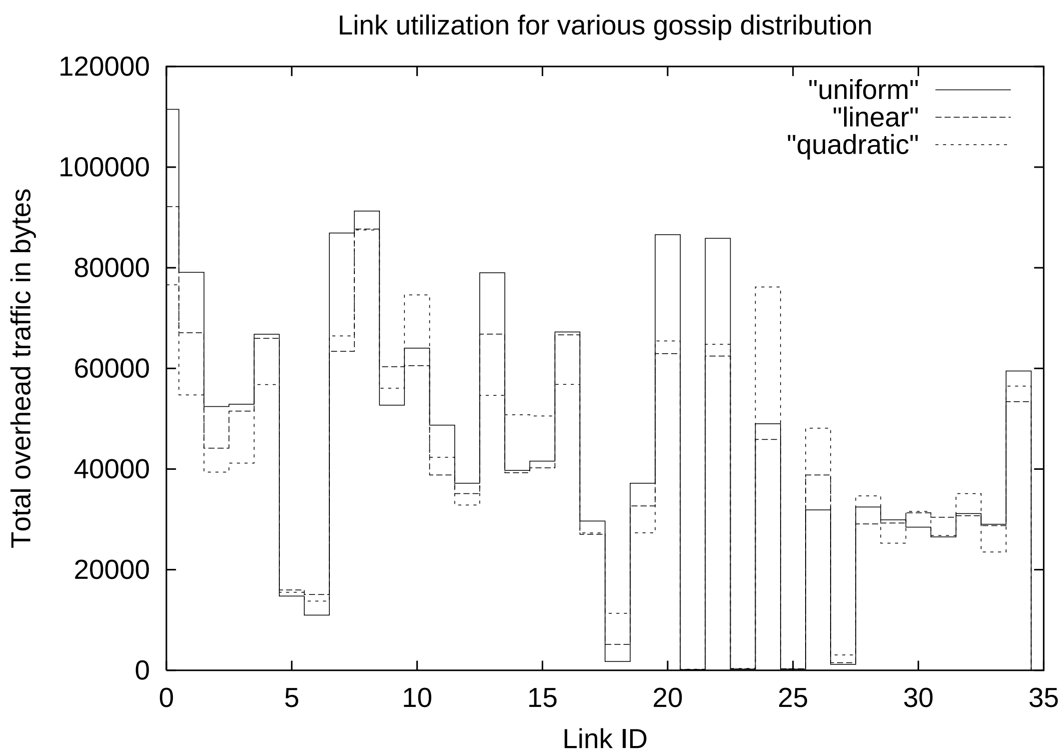

Figure 7-8: Link utilization

Table 7.2: Delivery latency

Distribution	Average latency	Std. Dev.
uniform	0.069361	0.162196
linear	0.082338	0.187496
quadratic	0.135950	0.439507

by more retransmission requests. *Pbcast* has more overhead than the other two because of the positive gossips. Again the difference between *LBRM* and *rpbcast* is the hash signature information in gossip messages.

7.5 Effects of different gossip selection distribution

In these test runs, we use linear biasing based on estimated round trip time in *rpbcast* for selecting gossip target. We also explored using uniform selection and quadratic biasing where the selection probability decreases quadratically with respect to the increase in round trip time. Figure 7-8 and table 7.2 summaries link utilization and latency for each of the three distributions. The experiments have 1% loss rate and approximately 360 packets per second.

Uniform selection has the lowest delivery latency. However, uniform gossips result in higher network traffic than biased distributions. Since the difference in network traffic is insignificant between linear and quadratic biasing, we chose linear biasing in the test runs because of the better latency.

Chapter 8

Conclusion

In this thesis, we have described our hybrid protocol *rpbcast* for high send rates and many senders. We preserved performance advantages of gossip-based multicast while adding packet reliability guarantees using loggers. The main contributions of our work are

- Integration of gossip based recovery mechanism with logger based recovery mechanism.
- Use negative gossips and gossiper-pull recovery mechanism to reduce overhead and improve delivery latency.
- Use hashing to reduce overhead generated by gossiping heartbeats and membership.
- A weak membership that exploits the flexibility in the garbage collection criteria to avoid expensive join/leave operations.

Our simulation performance results demonstrate that our hybrid protocol *rpbcast* improves upon previous work in situations with high send rates and many senders. Under high send rates, our average delivery latency result (Figure 7-4) shows that loggers in *rpbcast* overload at a much higher send rate than *LBRM*. Average delivery latency is also better when compared to *pbcast*. But more importantly, *rpbcast* guarantees reliable delivery under high send rates while *pbcast* does not. For situations with many senders, *rpbcast* imposes much less overhead traffic than *pbcast* due to its use of negative gossip and hashing techniques. From Figures 7-6 and 7-7, we see that the overhead for *LBRM* is less than *rpbcast*. However, *LBRM* does not have any membership or garbage collection mechanisms. Thus in a real world application setting, the combined overhead for *LBRM* is probably equivalent to *rpbcast* overhead.

We recognize that our protocol is not an ideal solution for low send rates or few senders because of higher ratio between gossip overhead and actual data traffic. Since delivery latency between *LBRM* and *rpbcast* is indistinguishable at low send rates, we feel that *LBRM* should out-perform our *rpbcast* in those cases. We emphasize applications of our protocol in large scale information distribution services, such as publish/subscribe systems. For future work, we intend to explore the integration of variable gossip rates to reduce overhead, sampling negative gossips for multicasting retransmissions, and dynamically switching between *LBRM* and *rpbcast* based on logger congestion.

Throughout this thesis, we have used I/O automata as the tool for describing our protocol and proving its correctness. The formal modeling of the environment and the protocol gave us several insights into the working of the protocol. For example, by breaking up the formal description of the logger based recovery phase and the gossip based recovery phase into two modules, we were able to clearly isolate the dependency between the two. In this case, we learned that our protocol will function correctly as long as the missing message IDs are moved from the gossip phase to the logger phase. Moreover, these two modules can be replaced by protocols with similar functionalities if the dependency is preserved in the process. We feel this is a great insight. Using the formal modeling also explicitly requires us to state our assumptions. For this particular thesis, stating our assumptions allowed us to state precisely when the stability-oriented garbage collection in Section 5.2 will work properly and what the membership protocol must provide. At an earlier stage of this thesis, there was a bug in the garbage collection because the membership protocol did not provide sufficient guarantees. However, breaking down the protocol into smaller components using I/O automata was difficult, especially when there are many shared variables between modules. Although we presented garbage collection and membership as two separate modules, the modules are tightly coupled. In those cases, overall functionality is not easy to grasp. In conclusion, we had a pleasant experience using I/O automata. Though the formal description and proof arguments took considerable time, we do have more confidence in the functionality of the protocol.

Bibliography

[1] D. A. Agarwal. *Totem: A reliable ordered delivery protocol for interconnected local-area networks.* PhD thesis, University of California, Santa Barbara, Department of Electrical and Computer Engineering, August 1994.

[2] M. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. Chandra. Matching events in a content-based subscription system. In *Principle of Distributed Computing*, 1999.

[3] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[4] T. Anker, D. Breitgand, D. Dolev, and Z. Levy. CONGRESS: Connection-oriented group-address resolution service. In *Proceedings of SPIE on Broadband Networking Technologies*, November 2-3 1997.

[5] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Int'l Conference on Distributed Computing Systems*, 1999.

[6] D. Belsnes. Single-message communication. *IEEE Transactions on Communications*, COM-24(2):190–194, February 1976.

[7] K. Birman, R. Friedman, M. Hayden, and I. Rhee. Middleware support for distributed multimedia and collaborative computing. In *Multimedia Computing and Networking (MMCN98)*, 1998.

[8] K. Birman, A. Schiper, and P. Stephenson. Lightweight casual and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[9] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

[10] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design of a scalable event notification service: interface and architecture. Technical Report CU-CS-863-98, Department of Computer Science, University of Colorado, August 1998.

[11] A. J. Demers, D. H. Greene, C. Hauser, W. Irish, and J. Larson. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, Vancouver, British Columbia, Canada, August 1987.

[12] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, Brisbane, Australia, February 1988.

[13] S. Floyd, V. Jacobson, C.G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, pages 784–803, December 1997.

[14] R. Golding and K. Taylor. Group membership in the epidemic style. Technical Report UCSC-CRL-92-13, UC Santa Cruz, Dept. of Computer Science, 1992.

[15] Object Management Group. Corba services: Common object service specification. Technical report, Object Management Group, July 1998.

[16] V. Hadzilacos and S. Toueg. a modular approach to the specification and implementation of fault-tolerant broadcasts. Technical report, Department of Computer Science, Cornell University, 1994. TR94-1425.

[17] M. Harchol-Balter, T. Leighton, and D. Lewin. Resource discovery in distributed networks. In *18th Annual ACM-SIGACT/SIGOPS Symposium on Principles of Distributed Computing*, Atlanta, May 1999.

[18] H. Holbrook, S. Singhal, and D. Cheriton. Log-based receiver-reliable multicast for distributed interactive simulation. In *Proceedings of ACM SIGCOMM '95*, 1995.

[19] T. Inc. Rendezvous information bus. Technical report, TIBCO Inc., 1996. http://www.rv.tibco.com/whitepaper.html.

[20] B. Jonsson. Compositional specification and verification of distributed systems. *ACM Transactions on Programming Languages and Systems*, 16(2):259–303, March 1994.

[21] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[22] J.C. Lin and S. Paul. A reliable multicast transport protocol. In *Proc. of IEEE INFOCOM'96*, pages 1414–1424, March 1996.

[23] B. Liskov and R. Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 29–39, Calgary, Alberta, Canada, August 1986.

[24] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., 1996.

[25] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Master's thesis, Massachusetts Institute of Technology, April 1987. Abbreviated version in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137-151, Vancouver, British Columbia, Canada, August, 1987.

[26] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, Chateau de Bonas, Gers, France, October 1988.

[27] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, pages 56–65, June 1994.

[28] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs, i. *Acta Informatica*, 6(4):319–340, 1976.

[29] O. Ozkasap, R. van Renesse, K. Birman, and Z. Xiao. Efficient buffering in reliable multicast protocols. In *First International Workshop on Networked Group Communication*, Pisa, November 1999.

[30] O. Ozkasap, Z. Xiao, and K. P. Birman. Scalability of two reliable multicast protocols. Technical report, Cornell University, Dept. of Computer Science, 1999.

[31] D. Park. Concurrency and automata on infinite sequences. In *Lecture Notes in Computer Science*, pages 167–183, Springer-Verlag, New York, 1981.

[32] S. Pingali, D. Towsley, and J. Kurose. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. *IEEE JSAC*, 15, April 1991.

[33] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end argument in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[34] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quencing. In *Proceedings of AUUG97*, July 1998.

[35] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proc. of Middleware '98*, pages 55–70, September 1998.