

SCALABLE MESSAGE STABILITY DETECTION  
PROTOCOLS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Katherine Hua Guo

May 1998

© Katherine Hua Guo 1998

ALL RIGHTS RESERVED

# SCALABLE MESSAGE STABILITY DETECTION PROTOCOLS

Katherine Hua Guo, Ph.D.

Cornell University 1998

In group communication, in order to deliver multicast messages reliably in a group, it is common practice for each member to maintain copies of all messages it sends and receives in a buffer for potential local retransmission. The storage of these messages is costly and buffers may grow out of bound. A form of garbage collection is needed to address this issue. Garbage collection occurs once a process learns that a message in its buffer has been received by every process in the group. The message is declared *stable* and is released from the buffer. An important part of garbage collection is message stability detection.

This dissertation presents the result of an investigation into message stability detection protocols. A number of message stability detection protocols used in popular reliable multicast protocols are studied with a focus on their performance in large scale settings. This dissertation proposes a new gossip-style protocol with improved scalability and fault tolerance. This dissertation also shows that by adding a hierarchical structure to the set of basic protocols, their performance can be significantly improved when the number of participants is large.

# Biographical Sketch

Katherine Hua Guo was born in 1969 in Beijing, People's Republic of China. She entered the University of Science and Technology of China as a biology major. Shortly afterwards she transferred to the University of Texas at Austin in Austin, Texas. After three and half years of college education, she earned a B.S. in computer science and B.A. in mathematics with special honor and highest honor. Then she moved to Ithaca, New York for her graduate study at Cornell University where she earned an M.S. in computer science in 1995. Her Ph.D. followed in 1998. Now she joins Bell Laboratories in Holmdel, New Jersey.

To those on whose shoulders I stand, and to my family.

# Acknowledgements

First, I want to thank the Chair of my committee, Ken Birman, for his guidance and support during my entire graduate study. His strategic insights, his unique perspective on distributed systems and on computer science research has been invaluable in my training process.

I am very lucky to have the opportunity to work with Robbert van Renesse. Step by step, he has shown me how to discover problems and solve problems which result in valuable computer system research. Werner Vogels started working with me when I lost my directions. I am grateful for his unwavering encouragement and sense of humor. I also want to thank S. Keshav for many insightful discussions during this work.

I would like to thank my committee member Steve Vavasis for carefully reading this dissertation and giving me valuable comments. My thanks also go to Nick Trefethen for his encouragement and for the opportunity to explore numerical analysis.

I wrote three papers with Luís Rodrigues at University of Lisboa. I am grateful to him for sharing his insights, knowledge, experience and time with me, and for being true friend. I am grateful to Alexey Vaysburd, Olga Veksler and Yuri Boykov for being wonderful officemates and for many discussions that cleared up

my thoughts.

Many thanks go to all the Horus researchers who were always willing to help me. I also wish to express my gratitude to Andrew Feng, who gave me help and support in too many ways to enumerate.

Looking through my past, however, I must express my ever deep gratitude to David Kincaid at the University of Texas at Austin, whose advice and guidance have helped me choosing a career in computer science research and going through difficult times in life.

Finally I would like to thank my parents and my brother for their constant love and support.

# Table of Contents

<b>Biographical Sketch</b>	<b>iii</b>
<b>Dedication</b>	<b>iv</b>
<b>Acknowledgements</b>	<b>v</b>
<b>List of Tables</b>	<b>x</b>
<b>List of Figures</b>	<b>xv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Large Scale Multicast . . . . .	2
1.2 Two Categories of Reliable Multicast . . . . .	3
1.3 Separate Issues in Reliable Multicast . . . . .	4
1.4 Related Studies . . . . .	5
1.5 Dissertation Outline . . . . .	5
<b>2 Background</b>	<b>7</b>
2.1 Categories of Reliable Multicast Protocols . . . . .	7
2.1.1 Sender-initiated protocols . . . . .	7
2.1.2 Receiver-initiated protocols . . . . .	11
2.1.3 Combination of sender-initiated and receiver-initiated protocols . . . . .	17
2.1.4 Hierarchical protocols . . . . .	17
2.2 Buffer Management . . . . .	18
2.2.1 Garbage collection . . . . .	19
2.3 Other Applications . . . . .	21
2.4 Summary . . . . .	22
<b>3 Failure Detection</b>	<b>24</b>
3.1 Failure Detection Algorithms . . . . .	24
3.1.1 Basic algorithm . . . . .	25
3.1.2 Gossip-style algorithm . . . . .	26

3.2	Integration of Stability Detection and Failure Detection . . . . .	29
3.3	Summary . . . . .	30
<b>4</b>	<b>Stability Detection</b>	<b>31</b>
4.1	Assumptions . . . . .	31
4.2	The Basic Protocols . . . . .	36
4.2.1	<b>CoordP</b> . . . . .	36
4.2.2	<b>FullDist</b> . . . . .	40
4.2.3	<b>Train</b> . . . . .	42
4.2.4	<b>Gossip</b> . . . . .	46
4.2.5	Analysis of <b>Gossip</b> protocol . . . . .	52
4.3	The Structured Protocols . . . . .	63
4.3.1	<b>S_CoordP</b> . . . . .	65
4.3.2	<b>S_Train</b> . . . . .	68
4.3.3	<b>S_Gossip</b> . . . . .	70
4.4	Comparison of the Seven Protocols . . . . .	72
4.5	Summary . . . . .	75
<b>5</b>	<b>Simulation of the Stability Detection Protocols</b>	<b>76</b>
5.1	The Underlying Network . . . . .	76
5.2	Complexity Metrics . . . . .	80
5.3	The <b>Gossip</b> Protocol . . . . .	82
5.3.1	Simulation with a fixed group size . . . . .	83
5.3.2	Adaptive method: finding the window of optimal step intervals	92
5.3.3	Simulation with varying group sizes . . . . .	95
5.3.4	Summary . . . . .	96
5.4	Comparison of Various Protocols in Dense Groups with 50 Senders	99
5.4.1	Total number of messages on all hops in the system . . . . .	100
5.4.2	Average and maximum queue sizes over all the nodes in the system . . . . .	104
5.4.3	Time-per-round (TPR) . . . . .	106
5.5	Comparison of Various Protocols in Sparse Groups with 50 Senders	113
5.5.1	Total number of messages on all hops in the system . . . . .	114
5.5.2	Average and maximum queue sizes over all the nodes in the system . . . . .	116
5.5.3	Time-per-round (TPR) . . . . .	118
5.6	Comparison of Various Protocols in Dense Groups with One Sender	125
5.6.1	Total number of messages on all hops in the system . . . . .	125
5.6.2	Average and maximum queue sizes over all the nodes in the system . . . . .	126
5.6.3	Time-per-round (TPR) . . . . .	127
5.7	Comparison of Various Protocols in Sparse Groups with One Sender	130
5.8	Summary . . . . .	131

<b>6</b>	<b>Stability Triggering Mechanism</b>	<b>133</b>
6.1	Summary . . . . .	138
<b>7</b>	<b>Discussion and Conclusions</b>	<b>139</b>
<b>A</b>	<b>Simulation Results for Gossip in Dense Groups</b>	<b>142</b>
<b>B</b>	<b>Simulation Results for Gossip in Sparse Groups</b>	<b>146</b>
<b>C</b>	<b>Simulation Results for Dense Groups with One Sender</b>	<b>150</b>
<b>D</b>	<b>Simulation Results for Sparse Groups with One Sender</b>	<b>155</b>
	<b>Bibliography</b>	<b>162</b>

# List of Tables

4.1	Complexity of protocols (exact formula) . . . . .	73
4.2	Complexity of protocols . . . . .	74
5.1	The near-optimal step interval (in seconds) for <b>Gossip</b> for different group sizes and different numbers of senders . . . . .	98
5.2	Total number of messages on all hops for various protocols . . . . .	102

# List of Figures

2.1	An example run of the generic sender-based protocol when message $m$ arrives at every receiver successfully. . . . .	9
2.2	An example run of the generic sender-based protocol when message $m$ does not reach receiver $r_2$ in the original multicast. . . . .	10
2.3	An example run of the first variation of the generic receiver-based protocol when message $m_2$ does not reach receiver $r_2$ in the original multicast. . . . .	13
2.4	An example run of the second variation of the generic receiver-based protocol when message $m_2$ does not reach any receiver in the original multicast. . . . .	14
2.5	An example run of the third variation of the generic receiver-based protocol when message $m_2$ does not reach receiver $r_2$ in the original multicast. . . . .	15
3.1	The Gossip-style failure detection protocol. . . . .	27
3.2	An example run of the gossip-style failure detection protocol. . . . .	28
4.1	Steps of protocol <b>CoordP</b> . . . . .	37
4.2	Steps of protocol <b>FullDist</b> . . . . .	41
4.3	Steps of protocol <b>Train</b> . . . . .	44
4.4	An example run of the gossip-style stability detection protocol <b>Gossip</b> . . . . .	48
4.5	Part I of the stability detection protocol <b>Gossip</b> . (In this version, the stability array $S$ is piggybacked on all gossip messages.) . . . .	49
4.6	Part II of the stability detection protocol <b>Gossip</b> . (In this version, the stability array $S$ is piggybacked on all gossip messages.) . . . .	50
4.7	Number of micro-steps needed to achieve different probability of incomplete stability detections. $P$ stands for $P_{incomplete}$ in the figure. . . . .	62
4.8	Number of steps needed to achieve different probability of incomplete stability detections. $P$ stands for $P_{incomplete}$ in the figure. . . . .	63
4.9	Cost of quality in terms of number of micro-steps. . . . .	64
4.10	Cost of quality in terms of number of steps. . . . .	64
4.11	Steps of protocol <b>S_CoordP</b> . . . . .	67
4.12	Steps of protocol <b>S_Train</b> . . . . .	69

4.13	Structure of the <b>S_Gossip</b> protocol. . . . .	72
5.1	Typical network topologies: (A): A start topology, (B): A chain topology, and (C): A bounded-degree tree where interior nodes all have degree 4. . . . .	77
5.2	Simulation I (no message loss) with a dense group of size 200, and subset size 3. . . . .	85
5.3	Simulation I (no message loss) with a dense group of size 200 (part I). . . . .	86
5.4	Simulation I (no message loss) with a dense group of size 200 (part II). . . . .	88
5.5	Simulation II (queue size = 64) with a dense group of size 200 (part I). . . . .	90
5.6	Simulation II (queue size = 64) with a dense group of size 200 (part II). . . . .	90
5.7	TPR for simulations I and II with 20 sparse groups of size 200. . .	91
5.8	Part I of the adaptive algorithm: finding a near-minimum TPR. $f(x)$ is the average measured TPR value for a step interval value $x$ . . . . .	93
5.9	Part II of the adaptive algorithm: finding the optimal step interval window given a near-minimum TPR. $f(x)$ is the average measured TPR value for a step interval value $x$ . . . . .	94
5.10	Near-minimum TPR and the corresponding number of steps needed in a round for sparse groups in Simulation II using the global gossip scheme with 50 senders. A data point with subset size $x$ and step interval $y$ seconds is labeled as $(x, y)$ . . . . .	95
5.11	Near-minimum TPR and the corresponding number of steps needed in a round for sparse groups in Simulation II using the global gossip scheme with one sender. A data point with subset size $x$ and step interval $y$ seconds is labeled as $(x, y)$ . . . . .	97
5.12	Total number of messages on all hops for the four basic protocols in dense groups with 50 senders. . . . .	101
5.13	Total number of messages on all hops for the three structured protocols in dense groups with 50 senders. . . . .	103
5.14	Total number of messages on all hops for the basic and their corresponding structured protocols in dense groups with 50 senders (part I). . . . .	103
5.15	Total number of messages on all hops for the basic and their corresponding structured protocols in dense groups with 50 senders (part II). . . . .	104
5.16	Average and maximum queue sizes over all the nodes for the basic and structured protocols in dense groups with 50 senders. . . . .	105

5.17	Time-per-round (TPR) for the four basic protocols in dense groups with 50 senders. . . . .	107
5.18	Time-per-round (TPR) for <b>Gossip</b> and <b>FullDist</b> employing the scattering mechanism in dense groups with 50 senders. . . . .	108
5.19	Time-per-round (TPR) for the three structured protocols in dense groups with 50 senders. . . . .	109
5.20	Time-per-round (TPR) for the basic and their corresponding structured protocols in dense groups with 50 senders (part I). . . . .	110
5.21	Time-per-round (TPR) for the basic and their corresponding structured protocols in dense groups with 50 senders (part II). . . . .	110
5.22	Total number of messages on all hops for the basic protocols in sparse groups with 50 senders. . . . .	114
5.23	Total number of messages on all hops for the three structured protocols in sparse groups with 50 senders. . . . .	115
5.24	Total number of messages on all hops for the basic and their corresponding structured protocols in sparse groups with 50 senders (part I). . . . .	116
5.25	Total number of messages on all hops for the basic and their corresponding structured protocols in sparse groups with 50 senders (part II). . . . .	117
5.26	Average queue size over all the nodes for the four basic protocols in sparse groups with 50 senders. . . . .	118
5.27	Average queue size over all the nodes for the structured protocols in sparse groups with 50 senders. . . . .	119
5.28	Maximum queue size over all the nodes for the four basic protocols in sparse groups with 50 senders. . . . .	119
5.29	Maximum queue size over all the nodes for the structured protocols in sparse groups with 50 senders. . . . .	120
5.30	Time-per-round (TPR) for the four basic protocols in sparse groups with 50 senders. . . . .	120
5.31	Time-per-round (TPR) for the structured protocols in sparse groups with 50 senders. . . . .	121
5.32	Time-per-round (TPR) for the basic and their corresponding structured protocols in sparse groups with 50 senders (part I). . . . .	122
5.33	Time-per-round (TPR) for the basic and their corresponding structured protocols in sparse groups with 50 senders (part II). . . . .	123
5.34	Total number of messages on all hops for <b>Gossip</b> and <b>S_Gossip</b> with different number of senders in dense groups. . . . .	126
5.35	Average queue size over all the nodes with different numbers of senders for the basic and structured protocols in dense groups. . . . .	128
5.36	Maximum queue size over all the nodes with different numbers of senders for the basic and structured protocols in dense groups. . . . .	128

5.37	Time-per-round (TPR) with different numbers of senders for the four basic protocols in dense groups. . . . .	129
5.38	Time-per-round (TPR) with different numbers of senders for the three structured protocols in dense groups. . . . .	130
6.1	Optimal number of messages in the output buffer when the stability detection protocol should be triggered ( $\alpha = 1$ ) . . . . .	137
6.2	Optimal number of messages in the output buffer when the stability detection protocol should be triggered ( $\alpha = 0.2$ ). . . . .	138
A.1	Simulation I (no message loss) with a dense group of size $n = 200$ (part I). . . . .	143
A.2	Simulation I (no message loss) with a dense group of size $n = 200$ (part II). . . . .	143
A.3	Simulation II (queue size = 64 and 2 lost messages per step) with a dense group of size $n = 200$ (part I). . . . .	144
A.4	Simulation II (queue size = 64 and 2 lost messages per step) with a dense group of size $n = 200$ (part II). . . . .	144
A.5	Simulation III (2 lost messages per step) with a dense group of size $n = 200$ (part I). . . . .	145
A.6	Simulation III (2 lost messages per step) with a dense group of size $n = 200$ (part II). . . . .	145
B.1	Simulation I (no message loss) with 20 sparse groups of size $n = 200$ (part I). . . . .	147
B.2	Simulation I (no message loss) with 20 sparse groups of size $n = 200$ (part II). . . . .	147
B.3	Simulation II (queue size = 64 and 2 lost messages per step) with 20 sparse groups of size $n = 200$ (part I). . . . .	148
B.4	Simulation II (queue size = 64 and 2 lost messages per step) with 20 sparse groups of size $n = 200$ (part II). . . . .	148
B.5	Simulation III (2 lost messages per step) with 20 sparse groups of size $n = 200$ (part I). . . . .	149
B.6	Simulation III (2 lost messages per step) with 20 sparse groups of size $n = 200$ (part II). . . . .	149
C.1	Total number of messages on all hops for the four basic protocols in dense groups with one sender. . . . .	151
C.2	Total number of messages on all hops for the three structured protocols in dense groups with one sender. . . . .	151
C.3	Total number of messages on all hops for the basic and their corresponding structured protocols in dense groups with one sender (part I). . . . .	152

C.4	Total number of messages on all hops for the basic and their corresponding structured protocols in dense groups with one sender (part II). . . . .	152
C.5	Average and maximum queue sizes over all the nodes for the basic and structured protocols in dense groups with one sender. . . . .	153
C.6	Time-per-round (TPR) for the basic and structured protocols in dense groups with one sender. . . . .	153
C.7	Time-per-round (TPR) for the basic and their corresponding structured protocols in dense groups with one sender (part I). . . . .	154
C.8	Time-per-round (TPR) for the basic and their corresponding structured protocols in dense groups with one sender (part II). . . . .	154
D.1	Total number of messages on all hops for the four basic protocols in sparse groups with one sender. . . . .	156
D.2	Total number of messages on all hops for the three structured protocols in sparse groups with one sender. . . . .	156
D.3	Total number of messages on all hops for the basic and their corresponding structured protocols in sparse groups with one sender (part I). . . . .	157
D.4	Total number of messages on all hops for the basic and their corresponding structured protocols in sparse groups with one sender (part II). . . . .	157
D.5	Maximum queue size over all the nodes for the four basic protocols in sparse groups with one sender. . . . .	158
D.6	Maximum queue size over all the nodes for the structured protocols in sparse groups with one sender. . . . .	158
D.7	Average queue size over all the nodes for the four basic protocols in sparse groups with one sender. . . . .	159
D.8	Average queue size over all the nodes for the structured protocols in sparse groups with one sender. . . . .	159
D.9	Time-per-round (TPR) for the four basic protocols in sparse groups with one sender. . . . .	160
D.10	Time-per-round (TPR) for the structured protocols in sparse groups with one sender. . . . .	160
D.11	Time-per-round (TPR) for the basic and their corresponding structured protocols in sparse groups with one sender (part I). . . . .	161
D.12	Time-per-round (TPR) for the basic and their corresponding structured protocols in sparse groups with one sender (part II). . . . .	161

# Chapter 1

## Introduction

Multicast is an efficient communication paradigm for disseminating data in a group with a sender and a set of receivers. Typically a multicast group is identified by a single group address. The semantics of reliable multicast communication are normally defined such that all members of the group need to receive a copy of the multicast message. Informally, this means that all correct processes deliver the same set of messages, and that this set include all messages multicast by correct processes, and no spurious messages [Mul93].

The growth of the Internet has triggered the widespread use of real-time multicast, including applications that support voice (for example, vat [JM], NeVot [Sch92]) and video (nv [Fre]) which do not require reliable multicast. There are also applications that do require reliable multicast such as shared white-boards (for example, wb [FJL<sup>+</sup>96]).

Many other multicast applications also require reliable delivery of data to all the receivers. For example, reliable multicast is used in Distributed Interactive Simulations (DIS) for dynamic terrain updates [HSC95]. It is used for dissemina-

tion of stock quotes to a large number of clients, and for distribution of software products to groups of customers. It is also used by web servers to send updates of web pages to their proxies.

The increasing popularity of end-to-end multicast applications supporting either video-conferencing, Computer Supported Collaborated Work (CSCW) or the reliable data dissemination over the Internet is making the provision of reliable and unreliable multicast services an integral part of its architecture.

## 1.1 Large Scale Multicast

In the near future, global information exchange will become an essential part of everyday life. Driven by the availability of high speed networks and powerful processors, more and more applications will require reliable data transfer within large groups, whose members may be spread all over the world. Therefore, forthcoming communication systems must scale well with respect to both number of group members and geographical expansion.

Meanwhile, widespread availability of IP multicast [DC90] and the MBone [Kum95] have dramatically increased the geographic extent and the size of communication groups.

To support reliable multicast communication in such a scenario, efficient and scalable multicast control mechanisms have become more and more essential.

## 1.2 Two Categories of Reliable Multicast

Generally, reliable multicast techniques fall in two categories: sender-initiated and receiver-initiated, both of which employ a sequential numbering of data messages at the sender.

The sender-initiated approach is based on the use of positive acknowledgments (ACKs). It places the responsibility on the sender, which maintains state information of all the receivers that it is multicasting data to. The receivers acknowledge the receipt of data messages by sending unicast ACKs to the sender. The sender keeps track of from whom it has received ACKs for each multicast message. A timer is associated with each message at the sender, and whenever the timer expires before ACKs from all the receivers come in, the sender re-multicasts the message.

In contrast, the receiver-initiated approach does not use ACKs at all. Whenever receivers detect a missing message by observing gaps in the sequence number stream of data messages, they send negative acknowledgments (NAKs) which serve as repair requests. In general, none of the members keep the state information regarding the set of receivers. If the receivers send repair requests to the sender, then the sender is responsible for data retransmission; if the receivers multicast repair requests in the group, then any member who has the requested message may conduct the retransmission. The retransmission is multicast to the entire group.

### 1.3 Separate Issues in Reliable Multicast

Much work has been done on the performance issues of point-to-point and point-to-multi-point transport protocols. The flow control, transmission error recovery, and buffer space management issues seem to interact in a rather complicated way [WM87].

We follow the discipline first proposed by Clark et al. in [CLZ87], whereby any transport protocol that operates efficiently decouples flow control and error control. Mixing the two in a single mechanism can make flow control vulnerable to transmission errors and delays. Under this discipline, the protocol uses a window for error control only. In practice, some protocols follow this discipline, and some do not.

Protocols like TCP [Pos81] use the same window both for flow control and for error control. A TCP transmitter stops at the window boundary and waits for a new acknowledgment before it can continue. This wait for synchronization serves as TCP's flow control mechanism and can often cause performance degradation.

On the other hand, protocols like NETBLT [CLZ87], Blast File Transfer Protocol [The92], and StarBurst Multicast File Transfer Protocol [Sta97] separate error control from their rate-based flow control mechanism. NETBLT maintains a large window at the sender, whereas Blast FTP and StarBurst MFTP keep the entire file to be transferred in the error control window.

Out of the three important issues in the design of reliable multicast protocols: flow control, error control and buffer management for the error control window, this dissertation investigates the issue of buffer management, that is, how to free messages from the error control window.

A message can be released from the buffer at any member (a sender or a receiver) only if it is received by every member in the group at which point the message is called *stable*. Therefore, buffer management is essentially the same as detecting message stability and releasing stable messages from the buffers.

## 1.4 Related Studies

There have been comparative analysis of sender-initiated and receiver-initiated reliable multicast protocols by Pingali et al. in [PTK94] and by Levine in [Lev96]. These studies conduct throughput analysis of different protocols based on processing requirements at both the sending and receiving hosts rather than the communication bandwidth requirements.

This dissertation studies only one aspect of reliable multicast protocols — the buffer management mechanisms. The focus is on the scalability of these protocols.

We examine various buffer management mechanisms for reliable multicast protocols in a large scale environment where messages may be dropped and processes may crash. We take both processing and communication bandwidth requirements into account, and study delay performance using event simulation.

## 1.5 Dissertation Outline

The results presented in this dissertation are based on simulations. They apply to generic protocols, rather than to specific implementations. We believe that they provide valuable insight for the design of next generation scalable reliable multicast protocols. Chapter 2 starts with a study of error control mechanism in reliable

multicast protocols, then introduces the need to do stability detection in reliable multicast and other important protocols. In order to detect message stability, it is necessary to have group membership information. Chapter 3 presents the failure detection protocol and the integration of failure detection and stability detection. The detailed comparison of stability detection protocols is presented in Chapter 4 followed by the simulation results in Chapter 5. Chapter 6 investigates the mechanism to invoke detection of stability. Chapter 7 concludes the dissertation. The appendices at the end provide some additional simulation results.

# Chapter 2

## Background

### 2.1 Categories of Reliable Multicast Protocols

With respect to mechanisms for error correction, reliable multicast protocols can be broadly separated into two categories: sender-initiated and receiver-initiated. A sender-initiated protocol is defined as one in which the sender gets positive acknowledgments (ACKs) from all the receivers periodically and release messages from its buffer accordingly. A receiver-initiated protocol is defined as one in which the receivers send negative acknowledgments (NAKs) when they detect message losses, and they never send any ACKs to the sender.

#### 2.1.1 Sender-initiated protocols

Sender-initiated reliable multicast protocols are based on the use of ACKs. The responsibility of providing message reliability is placed on the sender, which maintains state information regarding all receivers to which it is multicasting data. The sender keeps the list of all the receivers, and for each packet, the receivers from

which it has received ACKs.

Reliability and the maintenance of this state information are ensured by having the receivers return ACKs for messages correctly received, and by using timers at the sender for the purpose of detecting message losses.

A “generic” protocol might operate as follows:

- Whenever the sender multicasts a message, it starts a timer associated with this message.
- Periodically, a receiver sends back a unicast ACK to the sender identifying the messages it has correctly received so far. The ACK might indicate either a specific message or a window of messages.
- Upon receipt of an ACK, the sender updates its ACK lists associated with these messages indicated in the ACK.
- Whenever the timer expires before ACKs from all the receivers arrive at the sender, the sender re-multicasts the message.
- Whenever the ACK list associated with a message contains all the receivers in the group, the sender can release this message from its buffer and cancel the corresponding timer.

Some example runs of the sender-initiated protocol are given in Figures 2.1 and 2.2. In the first case as illustrated in Figure 2.1, once the sender  $s$  multicasts the message  $m$  to receivers  $r_1$  and  $r_2$ , it starts a timer for  $m$ . After receiving the unicast ACK from both  $r_1$  and  $r_2$ , the sender cancels the timer.

In the second case as in Figure 2.2, message  $m$  is multicast in the group.  $m$  arrives at  $r_1$  successfully, but it is lost on its way to  $r_2$ . When the timer for  $m$

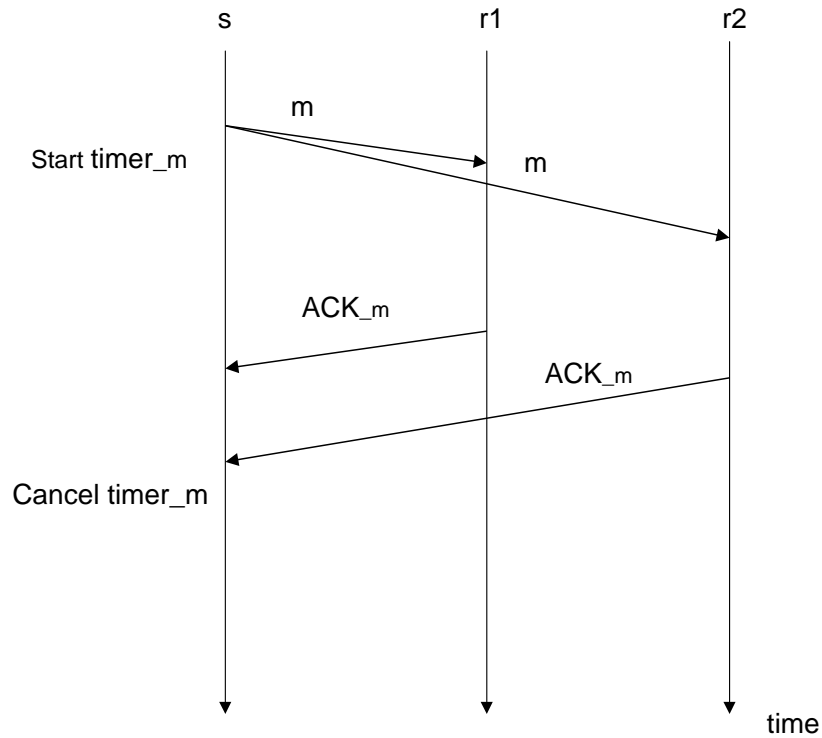


Figure 2.1: An example run of the generic sender-based protocol when message  $m$  arrives at every receiver successfully.

expires, the sender  $s$  has not received an ACK from  $r_2$ , therefore, it re-multicasts  $m$ .

The sender-initiated approach is used in early reliable multicast protocols such as the Xpress Transport Protocol (XTP) [SDW92,XTP95]<sup>1</sup>.

The main limitation of the sender-initiated protocol is that the sender needs to know the set of receivers, and needs to process ACKs from all the receivers. When the group size is large, the sender can be overwhelmed by the large amount of state

<sup>1</sup>XTP actually uses a combination of both sender-initiated and receiver-initiated approaches.

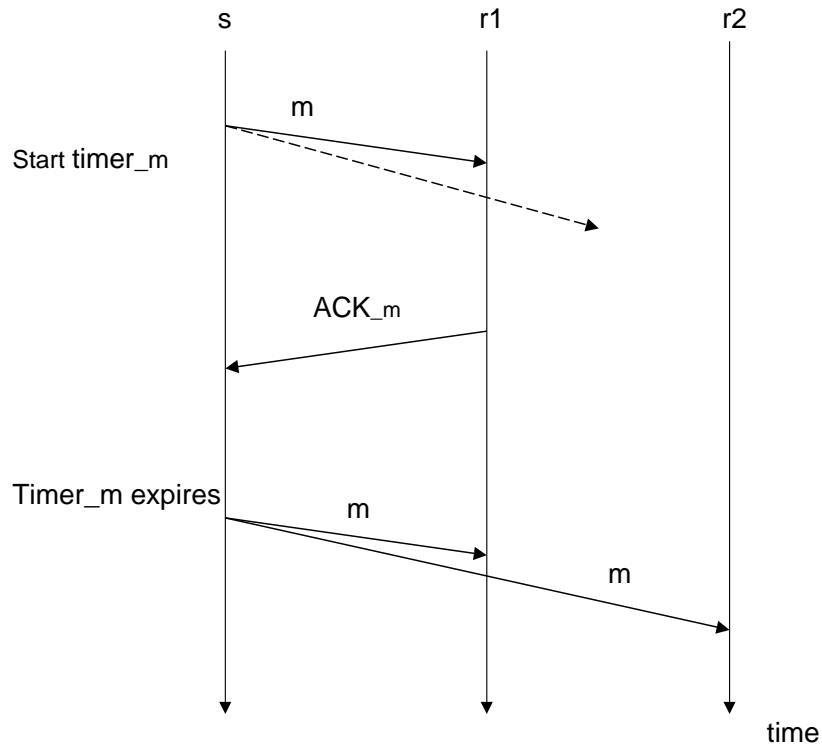


Figure 2.2: An example run of the generic sender-based protocol when message  $m$  does not reach receiver  $r_2$  in the original multicast.

information it must maintain, and experience an *ACK implosion* problem.

The ACK implosion problem is as follows. As the number of receivers becomes very large, the sender is overwhelmed with ACK messages from all the receivers. The network becomes congested from the arrival of a large number of messages in a short period of time, and the sender process experiences significant overhead due to the processing of the large number of ACK messages.

ACK implosion has the following impact:

- First, the tremendous number of ACK messages results in processing over-

head at the sender and results in delays in data communication.

- Second, a large number of ACK messages can cause an excess use of both buffer space and bandwidth, triggering additional message losses.

One approach to address the limitations of sender-initiated protocols is to use NAKs instead of ACKs for error detection and get rid of state information regarding all the receivers. We call this a receiver-initiated mechanism.

### **2.1.2 Receiver-initiated protocols**

Receiver-initiated protocols shift the burden of providing reliable data transfer to the receivers, and conduct error control based on negative acknowledgments (NAKs).

In this approach, the receivers are responsible for error detection and error recovery. They do not need to return status reports or acknowledgments to the sender. After a receiver detects lost messages by observing gaps in message sequence numbers, it informs other members via NAKs that solicit retransmissions. In order to guard against either the loss of the NAK or the subsequent message retransmission, the receiver starts a timer for each missing message. If the timer expires before the message is received, the receiver sends out another NAK message and restarts the timer. The timer is canceled upon successful receipt of the message.

To aid the loss detection for the last message in a burst, the sender must multicast the sequence number of that last message periodically.

A “generic” protocol might operate as follows:

- Whenever a receiver detects a lost message, it sends a NAK, and then starts a timer associated with this message.
- Whenever the timer expires before the requested message arrives, a receiver sends a NAK again and re-starts the timer.
- There are two approaches to send retransmission requests (NAKs): one is to unicast to the sender, the other is to multicast to the entire group. Combined with the way retransmission is sent out, we have the following three variations:
  - In the first variation, upon receipt of a unicast NAK, the sender re-multicasts the requested message.
  - In the second variation, upon receipt of a multicast NAK, the sender re-multicasts the requested message. Whenever a receiver receives a NAK requesting the same message it has requested in its own NAK, it resets the timer associated with the missing message. This mechanism suppresses redundant NAKs by backing off the timers at receivers.
  - The third variation is the same as the second one, except any member with a copy of the requested message can re-multicast in the group upon receipt of a NAK.

Examples of the three variations of the receiver-initiated protocol are presented in Figures 2.3 to 2.5. Figure 2.3 illustrates the first variation of the protocol. The sender  $s$  multicasts messages  $m_1$ ,  $m_2$ , and  $m_3$  to the two receivers. Out of these three multicast messages,  $m_1$  and  $m_3$  arrive at both receivers successfully,  $m_2$

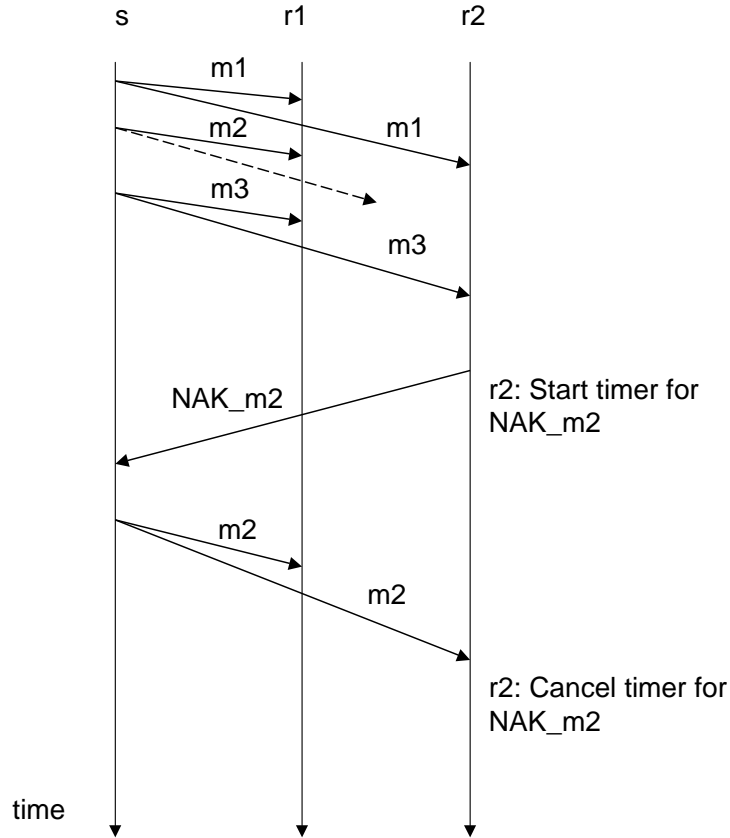


Figure 2.3: An example run of the first variation of the generic receiver-based protocol when message  $m_2$  does not reach receiver  $r_2$  in the original multicast.

arrives at  $r_1$  but not  $r_2$ . As soon as  $r_2$  detects the loss of  $m_2$  from the gap in the sequence number space, it sends a unicast NAK message to the sender  $s$  requesting the retransmission of  $m_2$ , and starts a timer for the NAK message of  $m_2$ . Once the sender receives the NAK from  $r_2$ , it re-multicasts  $m_2$  to both receivers.  $r_1$  ignores the message since it is a duplicate. After  $r_2$  receives  $m_2$  successfully, it cancels the timer for the NAK.

Figure 2.4 presents a sample run of the second variation. Message  $m_2$  fails to arrive at either  $r_1$  or  $r_2$ . Receiver  $r_1$  detects the loss first, then it multicasts a

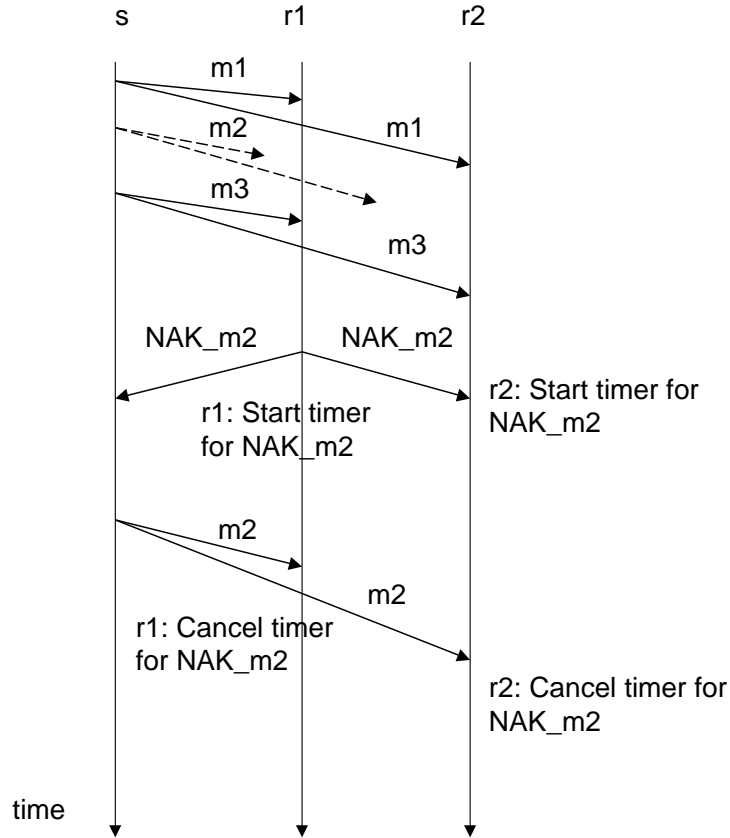


Figure 2.4: An example run of the second variation of the generic receiver-based protocol when message  $m_2$  does not reach any receiver in the original multicast.

NAK message for  $m_2$  and starts a timer for the NAK. After  $r_2$  receives the NAK, it starts a timer for the NAK for  $m_2$  as if it has just sent out the NAK itself. Once the sender  $s$  receives the NAK from  $r_1$ , it re-multicasts  $m_2$  in the group. After  $m_2$  arrives at  $r_1$  and  $r_2$ , both receivers cancel their corresponding timers for the NAK.

In Figure 2.5, the third variation is used. Message  $m_2$  arrives at  $r_1$  but not  $r_2$ . Once  $r_2$  detects the loss, it multicasts a NAK message requesting  $m_2$ . It also starts a timer for this NAK. As soon as  $r_1$  receives this NAK request, since it has already received  $m_2$ , it re-multicasts  $m_2$  in the group. Once  $m_2$  arrives at  $r_2$ , the timer

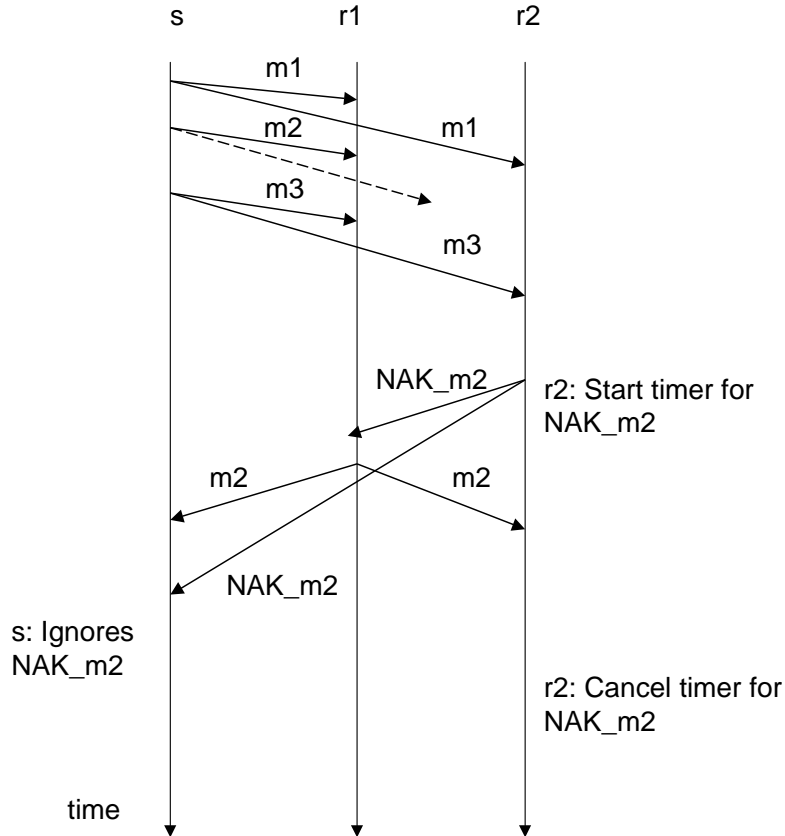


Figure 2.5: An example run of the third variation of the generic receiver-based protocol when message  $m_2$  does not reach receiver  $r_2$  in the original multicast.

for the NAK is canceled. The sender  $s$  receives the NAK for  $m_2$  after it receives  $m_2$ , it ignores the NAK accordingly since it knows some other member has already re-multicast  $m_2$ .

The first variation is used by Birman in the ISIS reliable multicast protocol [Bir93] in a local area network (LAN) environment.

The second variation is proposed by Ramakrishnan and Jain in the “Negative Acknowledgments with Periodic Polling” (NAPP) protocol [RJ87] for a LAN. Jacobson has used similar ideas to implement a reliable multicast protocol suitable

for the wide area network (WAN) [Jac93].

The third variation is used in the Scalable Reliable Multicast (SRM) protocol designed by Floyd et al. [FJL<sup>+</sup>96], where any member which has the requested message may conduct the retransmission. This technique effectively reduces the burden on the sender and shortens the retransmission delay.

Because the sender does not have group membership information, and the receivers do not send feedback to the sender upon successful receipt of messages, the sender has no mechanism to ascertain when it can safely release messages from its buffer. Furthermore, this approach is not suitable to provide a full-reliable communication service, because there is no mechanism for any member to detect when a receiver fails or leaves the group.

The benefit of this scheme is that since the state information is minimum, it scales well.

Clearly in the receiver-initiated approach, in order to handle possible retransmission requests, if the sender is responsible for retransmission, then all the multicast messages must be kept in the sender's buffer; whereas if all the members are responsible for retransmission, then all the messages must be maintained in the buffers of all the members.

A pure receiver-initiated approach lacks a buffer management scheme to detect when a message is received by all the members, and therefore can be safely released from each member's buffer. To combat the limitation of the pure receiver-initiated approach in practice, many protocols combine the use of both ACKs and NAKs.

### **2.1.3 Combination of sender-initiated and receiver-initiated protocols**

In the hybrid approach that combines the sender-initiated and receiver-initiated mechanisms, regardless of whether the sender or the receivers are in charge of detecting message losses and conducting retransmission, that is, despite of the use of ACKs or NAKs for retransmission requests, the sender is in charge of releasing a message from its buffer and from receivers' buffers after every receiver has positively acknowledged receipt of the message. ACKs are employed by the sender to ascertain that it is safe to release messages from memory.

The combination of ACKs and NAKs have been used extensively for reliable multicast protocols. For example, the Xpress Transport Protocol (XTP) [SDW92, XTP95], the "Negative Acknowledgments with Periodic Polling" (NAPP) protocol [RJ87], and the StarBurst Multicast File Transfer Protocol (StarBurst MFTP) [Sta97] group together large partitions of data messages that are periodically ACKed, while lost messages within the partition are NAKed.

### **2.1.4 Hierarchical protocols**

A hierarchical structure can reduce the ACK implosion problem in sender-initiated protocols. The Reliable Multicast Transport Protocol (RMTP) developed at Bell-Labs by Sabnani et al. [SLPB97] employs a hierarchy by dividing the receivers into some number of subsets and using Designated Receivers (DRs), one per subset, to collect ACKs from other members in that subset. Thus, the sender only keeps the list of DRs, and each DR keeps membership information of its subset. This hierarchy reduces both the amount of state information required at the sender, and

the number of ACKs collected by the sender.

The receiver-initiated protocol can also benefit from a hierarchy. For example, the Log-Based Receiver-reliable Multicast (LBRM) developed by Holbrook et al. [HSC95] uses a hierarchy of log servers to store all multicast messages indefinitely. Receivers request retransmissions of lost messages from a log server. The log servers significantly reduce the sender's burden of handling retransmissions.

The Local Group-based Multicast Protocol (LGMP) by Hofmann [Hof96a, Hof96b] is based on the concept of Local Groups where lost messages are first recovered inside Local Groups using NAKs. A message is requested from the sender only if not a single member of the Local Group holds a copy of the missing message.

## 2.2 Buffer Management

Transport level buffers have a certain bound in terms of size. For example, in Unix, the default buffer size for each TCP connection is 32K bytes at the sender and at the receiver. The purpose of transport level buffers is to store messages for reasonable amount of time in case retransmission is needed. After these buffers become full, new messages are either dropped or forced to replace existing messages in the buffer depending on the buffer management scheme.

In the following simple calculation, we assume the sender is responsible for retransmission, and it stores all the multicast messages it has sent out in its buffer. Assume the sender multicasts fixed-size messages at the rate  $r$  messages per second, and the message size is  $k$  bytes. Also assume the buffer size at the sender is  $B$  bytes. Then  $t = \frac{B}{r \times k}$  seconds after the sender starts multicasting, the buffer will

be full. After this point, new messages have to replace old messages in the buffer. If the time it takes to detect message loss and send back NAK messages is greater than  $t$ , then the original message is not available in the transport level buffer for retransmission any more.

If the application layer can store all relevant data, then lost messages that are not in the transport level buffers can be reconstructed by the application, and retransmitted; otherwise, they are lost and can not be retransmitted.

SRM uses the first approach, employing a concept called Application Level Framing (ALF) proposed by Clark and Tennenhouse in [CT90]. In the design principle ALF, the application breaks the data into suitable aggregates called Application Data Units, or ADUs. ADUs will take the place of the packet as the unit of manipulation. When mis-ordered or incomplete data units occur, the application rather than the transport protocol provides the data for retransmission by reconstructing the data.

### 2.2.1 Garbage collection

In scalable reliable multicast protocols [FJL<sup>+</sup>96,Hof96a,Hof96b,SLPB97], it is most efficient to use the *local repair* scheme, that is, for each group member to retransmit messages in response to requests by other members that have detected message losses. For applications that require all messages to be delivered to all correct processes in the group, it is also necessary to buffer all the received messages at every member to handle the case of sender crash and network partition.

On the other hand, the storage of these messages is costly and the buffer space at each member is limited, preventing the protocols from scaling to a large group

size. A form of *garbage collection* is needed to address this issue.

In order for members that join the group late to catch up with the rest of the group, a small number of members are designated as the Late-Join Handlers (LJHs). LJHs keep all messages they sent and received in their buffers. The decision of how long the LJHs should keep multicast messages is made by applications instead of the garbage collection mechanism.

Whenever a data message has been received by all the members, only the LJHs should store the message. Other members should discard it, since none of the members in the current multicast group needs retransmission. A message is called *stable* if it is received by all the members of the group. To do this garbage collection, a mechanism is needed to detect which messages are stable. Also a failure detection mechanism is needed to report the current group membership, otherwise a failed member could prevent garbage collection altogether.

Under the ideal situation where the sender does not crash and there is no network partition, sender-initiated protocols already have built-in buffer management and garbage collection mechanisms. After the sender gets ACKs from all the receivers, it can detect which messages are stable and can be released from each member's buffer. Receiver-initiated protocols do not have this stability detection mechanism since none of the members have group membership information.

If we want to guarantee reliable message delivery in face of sender crash and network partition, then we need a mechanism to do stability detection and buffer management for both sender-initiated and receiver-initiated protocols.

The problem of buffer management is largely ignored in existing scalable reliable multicast protocols in the literature. Based on the concept of Application Level

Framing (ALF) [CT90], the issue of how long a message should be buffered is handled by the application layer. For example, in SRM, all messages belong to the current white-board session are stored in the buffer of each group member. Depending on how long a session is, the number of messages in a session could be unbounded. To our knowledge, this is the first comprehensive study of transport level buffer management for reliable multicast protocols.

This dissertation evaluates the scalability and performance of existing stability detection protocols and further proposes new protocols.

## 2.3 Other Applications

The message stability detection mechanism can also support *atomic message ordering* which means that a message is not delivered to any group member until all the members have received it. For example, the research reported here was triggered by a problem that a Swiss bank faced when using the Isis group communication system [Bir93]. In their set-up, they had two server machines and about a hundred PC workstations organized in a group. The servers multicast updates to replicated data maintained at each of the workstations. The updates had to be delivered atomically, in spite of server failures. Therefore, the workstations had to buffer the data until it was known that the data was delivered everywhere. The rate of the updates was sometimes so high, that Isis' stability detection protocol was not able to keep up, and buffers grew too large. The effect was much exacerbated by the fact that multiple groups were used. Correct ordering between the groups required that switching from sending in one group to another was done only after the messages sent and delivered in the first group had become stable.

These machines were inside a single branch and on a single local area network. One can easily envision multiple branches being linked together, with many hundreds if not thousands of machines. Our interest is in finding a scalable stability detection protocol that fits future requirements.

The concept of message stability has been used in more traditional areas such as distributed database management and parallel computing. In distributed database systems, partial failures of transactions can lead to inconsistent results. Therefore, termination of a transaction that updates distributed data has to be coordinated among its participants. In the atomic commit protocols [BHG87], a process can not commit a transaction until everybody else has agreed to commit. This is similar to message stability detection protocols in which all processes must deliver a message if any does so.

In parallel computing, barrier synchronization [SKB89] requires that all processes execute the barrier construct before any process can proceed past it to the next statement. Every process has to know if all other processes have reached the barrier before it can proceed again. This is also an agreement problem similar to message stability and atomic commit.

## 2.4 Summary

This chapter provides the background to the study of message stability detection protocols to be presented in subsequent chapters. Two categories of reliable multicast protocols, the sender-initiated and the receiver-initiated protocols, are described along with the pros and cons of each category. Two common approaches to improve the scalability of reliable multicast protocols are studied: one is to com-

bine the two categories of protocols, the other is to employ a hierarchical structure. The question of efficient buffer management and the concept of garbage collection are raised in the context of reliable multicast.

This background makes it possible to now study the failure detection algorithms (Chapter 3), and look in detail at the different schemes to conduct message stability detection (Chapters 4 to 6).

# Chapter 3

## Failure Detection

To do effective garbage collection, one must detect when a message is stable and obtain a consistent view of the current group membership. Therefore, the stability detection protocol and failure detection protocol are two integral parts of garbage collection. This chapter describes the failure detection algorithms.

### 3.1 Failure Detection Algorithms

To conduct multicast in a distributed environment, one must face the problem of dynamic group membership changes. Initially, there are  $n$  members in the group numbered 1 through  $n$ . As time passes by, new members might join the group, existing members might crash or might leave the group voluntarily. The failure detection problem is to detect which members of the group are still operational and therefore constitute the current membership.

### 3.1.1 Basic algorithm

Traditionally failure detection algorithms are based on time-out mechanisms. In general, a failure is detected when the lack of response from a remote member process makes communication protocols unable to make progress. Under the assumption that every member process is constantly sending out messages, if a member has not been heard from after a certain time, it is assumed to have crashed or left the group. The failure detection algorithms normally reside in the transport layer that implements inter-process communication.

To ensure timely detection of failures when data traffic is low or unidirectional, some systems require each member to multicast “I-am-alive” session messages in the group periodically.

In general, failure detection algorithms can be divided into two categories under the same time-out principle according to Vogels in [Vog96].

The first scheme uses a heartbeat mechanism, where each process sends out “I-am-alive” session messages in the group of processes using multiple point-to-point messages or a single IP-multicast message. Each process records the reception times of messages and if a number of consecutive heartbeats from a certain member are missing, a suspicion is raised for this member. The lengths of inter-heartbeat gaps and time-out periods are configurable by the application. The application can also piggyback data messages on the heartbeat messages.

The second scheme uses a polling method where the failure detector sends request messages to processes in the group and collects acknowledgments from them. If no acknowledgments are received after a number of retries, the failure detector raises a suspicion. Poll periods, time-outs, and retransmission limits are

also configurable by the application.

When scaled up to more than several dozens of members, many failure detectors are either unreasonably slow, or make too many false detections as reported by van Renesse et al. in [vRMH98]. The reason is that in either the heartbeat method or the polling method, the large number of “I-am-alive” messages and polling requests add unnecessary loads to the system.

### 3.1.2 Gossip-style algorithm

We propose a new gossip-style algorithm to do failure detection. The protocol is based on the gossiping technique pioneered in the Clearinghouse project in the 1980’s [DGH<sup>+</sup>87,OD83]. Over a decade before that, Baker and Shostak [BS72] describe a gossip protocol using ladies and telephones before the widespread use of computers and networks.

The goal of gossip protocols is to distribute information in the group. The mechanism is for each member to forward new information to randomly chosen members periodically. The randomly chosen members during each gossip constitute a *gossip subset*. And the gossip period is called *step interval*.

The gossip-style failure detection algorithm works as follows. Every member maintains an  $n$ -element “Live” array  $L$  which is filled with 0’s initially. This protocol is divided into equally timed steps. During every gossip step, each member  $i$  increments all the other elements in its “Live” array  $L$  by 1 while keeping  $L[i] = 0$ , then it gossips  $L$  to a random subset. Upon receiving a data message from member  $j$ , a member sets  $L[j] = 0$ . Upon receiving another “Live” array  $L'$ , a member replaces its own “Live” array  $L$  with the element-wise minimum of its old  $L$  and

$L'$ . Small values in the live array indicate that the corresponding members are active, and large values signify that the corresponding members have not been heard from recently. The pseudo-code is presented in Figure 3.1.

*Each member  $i$  keeps a live array  $L_i$ .*

*Initially  $L_i = [0 \ 0 \ \dots \ 0]$  at every member  $i$ .*

*Periodically, member  $i$  does the following:*

*$L_i[j] = L_i[j] + 1$ ; for all  $j \neq i$*

*sends out a gossip message containing  $L = L_i$ ;*

*Every member reacts to received messages as follows:*

*Upon receiving a data message from  $j$ , member  $i$  does the following:*

*$L_i[j] = 0$ ;*

*Upon receiving  $L$ , member  $i$  does the following:*

*$L_i = \text{ArrayMin}(L_i, L)$ ;*

Figure 3.1: The Gossip-style failure detection protocol.

Figure 3.2 presents a sample execution of the gossip-style failure detection protocol. In this example, there are four members in the group —  $A$ ,  $B$ ,  $C$  and  $D$ . We examine the behavior of the protocol at member  $A$ . Initially, the “Live” array at  $A$  is  $[0 \ 0 \ 0 \ 0]$ . After one step interval,  $A$  increments every element of the “Live” array by 1 except for itself, and the “Live” array becomes  $[0 \ 1 \ 1 \ 1]$ . After this increment,  $A$  gossips its current “Live” array to  $B$ . When the next step interval passes, the “Live” array becomes  $[0 \ 2 \ 2 \ 2]$  and  $A$  gossips it to  $C$ . When  $A$  receives a data message from  $C$ , it sets the element for  $C$  to be 0. The “Live” array becomes  $[0 \ 2 \ 0 \ 2]$  at this point. After some time passes, the “Live” array is  $[0 \ 4 \ 3 \ 6]$ . At this moment,  $A$  receives  $D$ ’s “Live” array  $[2 \ 1 \ 2 \ 0]$ , and calculates the element-wise

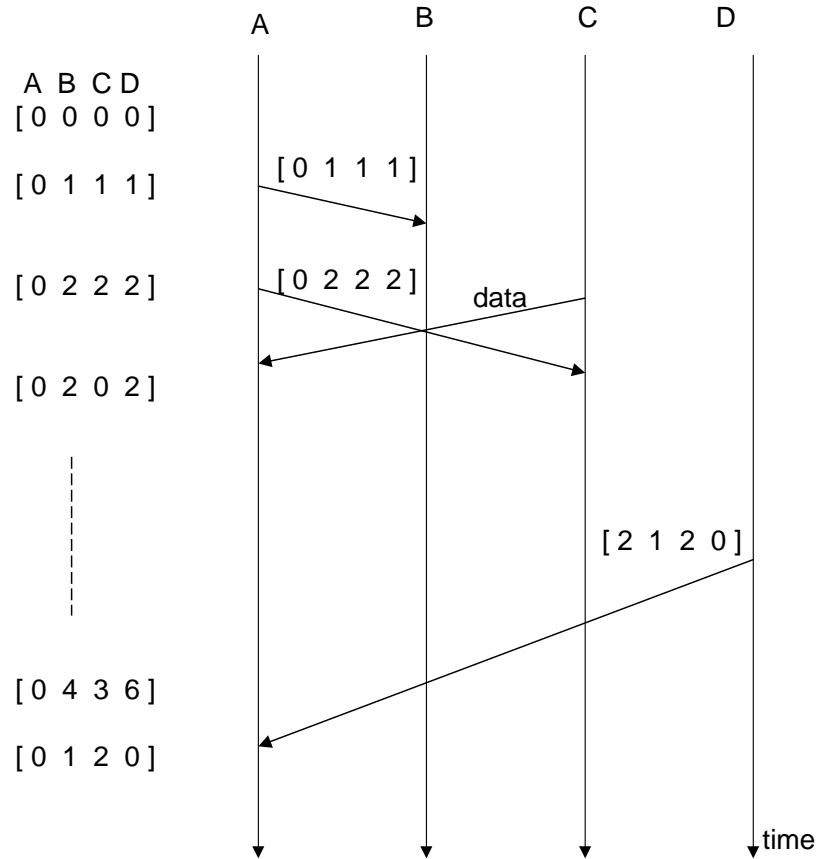


Figure 3.2: An example run of the gossip-style failure detection protocol.

minimum  $ArrayMin([0\ 4\ 3\ 6], [2\ 1\ 2\ 0]) = [0\ 1\ 2\ 0]$  which is the new “Live” array at  $A$ .

It takes time for the “Live” array from each member to propagate throughout the entire group. Therefore we set a threshold value  $K_{fail}$ , the maximum “Live” array value. Once  $K_{fail}$  is reached, the corresponding member is considered failed. The value  $K_{fail}$  depends on the gossiping rate (the length of step interval) and the subset size during each gossip step.  $K_{fail}$  is selected so that the probability that anybody makes an erroneous failure detection is less than some small threshold  $P_{mistake}$ .

In [vRMH98], van Renesse et al. conduct a statistical analysis for a better version of the protocol where during each gossip step, only one member is gossiping. This member is chosen at random and chooses one other member to gossip to at random. Under this condition, the number of steps needed increases logarithmically with the number of members  $n$ . The “Live” array size increases with the group size  $n$ , therefore, in order to keep the bandwidth requirement per member constant, the gossip step is set to be proportional to  $n$ . With this requirement,  $K_{fail}$  grows in the order of  $n \log n$ .

Assuming each element of the “Live” array occupies 1 byte, the size of a gossip message becomes  $n$  bytes where  $n$  is the group size. A hierarchical structure can be employed in the failure detection protocol to reduce gossip message sizes and improve scalability. Since the stability detection protocol is the focus of this dissertation, the failure detection protocol is not discussed further.

## 3.2 Integration of Stability Detection and Failure Detection

When there are membership changes, the failure detection protocol assists the stability detection protocol. If a faulty member is detected, this information is propagated throughout the group. As presented in Section 4.1, each member maintains an  $n$ -bit “Whom-I’ve-heard-from” bitmap array  $W$  for recording from which members it has received information needed for stability detection. Members always check the “Whom-I’ve-heard-from” array  $W$  against the current group membership before deciding if message stability is reached, thus preventing an indefinite

wait for faulty members' sequence number arrays.

Recall from Section 2.2, a small set of members are designated as Late-Join Handlers (LJHs) and the LJHs will provide new members with data necessary to catch up with existing members. When a new receiver joins the group, after receiving necessary information from the LJHs, it also joins the stability detection protocol. The “Whom-I’ve-heard-from” bitmap array  $W$  adds one more bit at the end representing the new receiver. Any member which hears indirectly or directly from this new receiver notices the change in  $W$  from gossip messages and adds one bit to its own  $W$ .

The invocation of the stability detection protocol depends on patterns of message sending and membership changing. Since there is a limit on buffer space at multicast group members, a round of the stability protocol should start whenever the buffers reach some threshold. An analytical model for determination of this threshold is discussed in Chapter 6.

### 3.3 Summary

The stability detection protocol and failure detection protocol are two integral parts of garbage collection. This chapter starts with a survey of failure detection protocols, followed by the proposal of a new gossip-style protocol to detect failures. Finally, this chapter describes how failure detection protocols assist stability detection protocols when there are membership changes. With this background, in the next few chapters, we can start looking at mechanisms to conduct stability detection which are the main focus of this dissertation.

# Chapter 4

## Stability Detection

The protocol that collects message stability and distributes this information to every group member is called a *message stability detection protocol*. Such protocols are implemented as an integral part of reliable multicast protocols in many distributed systems [ADKM92, Bir93, Car85, Cri91, CM95, Hay98, KTHB89, vRBM96]. We first study three representative protocols named **CoordP**, **FullDist**, and **Train**, then propose a new protocol called **Gossip**.

### 4.1 Assumptions

As mentioned in Chapter 1, this dissertation studies the message stability detection and failure detection framework intended for reliable multicast in a large scale environment where messages may be dropped and processes may crash.

In a dynamic distributed environment, certain communication problems may mimic process failures. For example, when processes  $p$  and  $q$  are both functional, but the communication link between them experiences transient failures, perhaps

process  $p$  will consider that process  $q$  has failed while process  $q$  believes the opposite is true. This situation is called a *network partition* or *partitioning failure*. In order for the system to make progress, one of these events must become official.

If a partitioning failure occurs in a system, it is impossible to guarantee that multiple components can deliver the same set of messages. To obtain strong system-wide guarantees, a protocol must wait for communication to be reestablished in at least one side of the partition. In the primary partition approach pioneered in the Isis system [Bir93], one and only one of these components is designated as the *primary component*. The primary component is permitted to make progress, and other components are forced to shut down. Processes within non-primary components reconnect to the primary component when communication is restored.

An alternative non-primary partition approach is used in the Transis [ADKM92, DMS94] and Totem [MAMSA94, MMSA<sup>+</sup>96] systems in which any component that can reach internal agreement on its membership is permitted to continue operation. However, only a single component of the system is the primary one. Applications might continue to be available in non-primary components. When the partition failure ends, non-primary components merge their states back into the primary one.

If the system follows the primary partition approach, then stability detection protocols only execute in the primary component when network partition happens. If the system allows non-primary component(s) to execute, the stability detection protocols are executed in these components also.

When the system is free of partition failures, whenever a message is determined stable by the stability detection protocols, it can be released (or garbage collected)

from every member's buffer. When network partition occurs, messages can not be released even if they are detected to be stable within the network component, because members in other component may need the retransmission during the merge process.

The merge of network partitions is handled by users of the stability detection protocols. Therefore we only describe the protocols under the situation where no network partition can happen.

We base our analysis on the following common assumptions about reliable multicast protocols. Notice we do not assume FIFO ordering.

- A multicast group of size  $n$  consists of a set of processes named from 1 to  $n$ .
- Each member of the group can be a sender multicasting data messages to the entire group. Without loss of generality, we assume  $m$  ( $m \leq n$ ) processes are senders and they are numbered 1 through  $m$ .
- Each member is always a receiver. This means the sender is also a receiver for the messages it sends out.
- The sender assigns each data message a sequence number that is unique for the particular sender. Therefore, each data message has a unique name; this name consists of the globally unique sender name and a locally unique sequence number.
- Each sequence number occupies 4 bytes. (The sequence number space is between 0 and  $2^{32}$ , which is large enough for most applications).
- A multicast message is always sent to the entire group, and therefore a sender also receives a multicast message from itself.

- Multicast in the stability detection mechanisms can use the underlying reliable multicast protocols, even though some stability detection protocols have their own built-in reliability mechanism.

For easy comparison, all protocols under consideration will be organized into *rounds*. A round begins when the protocol is initiated externally, and each round has a definite termination point. The reason is that one could imagine a stability protocol that runs autonomously, asynchronously, and continuously.

For a stability detection protocol to conduct useful work, the following condition must be satisfied. If a message is stable at time  $t$  when a protocol round begins, then the stability detection protocol must certify its stability by the time the round finishes. It is because during the execution of the protocol, the number of stable messages can only increase, but not decrease. With this constraint, we rule out any protocol that does not make progress.

Notice there is a time difference between the moment a message becomes stable and the moment it is *detected* to be stable. If a protocol is triggered when a message becomes stable at  $t_1$ , and it is detected to be stable at  $t_2$ , then the time difference  $t_2 - t_1$  indicates the performance of the stability detection protocol. The smaller the time difference, the better the protocol performs.

Without considering the underlying network topology, we can use two metrics to characterize the performance of stability detection protocols.

We use *number of steps per round* as a time complexity metric. Since not every step takes the same amount of time, the number of steps alone does not give accurate information about time complexity. We use it as a convenient aid in description of the protocols. We can also draw conclusions on protocol performance

based on the number of steps needed and the estimated time needed for each step.

To measure the distribution of processing load among group members, we use *number of messages processed per round*, which is the sum of the number of messages sent and received by each member during each protocol round.

In all the protocols we study, each member maintains an  $m$ -element sequence number array  $R$  where its  $j$ -th element  $R[j]$  is the maximum sequence number of all messages from sender  $j$  that have arrived at this member. Each member also maintains an  $n$ -element “Live” array  $L$  reflecting current group membership (as described in Section 3.1.2) and an  $n$ -bit “Whom-I’ve-heard-from” bitmap array  $W$  for recording from which members it has received sequence number arrays.

All the message stability detection protocols follow the same procedure in a round of execution:

- When the sequence number array  $R$  from each member is collected somehow, a stability array  $S$  is created where  $S[j]$  is the minimum of the  $j$ -th element of each member’s sequence number array.
- After the stability array  $S$  is built, it is distributed in the group using various mechanisms.
- After receiving  $S$ , each member can release data messages from sender  $j$  with sequence numbers less than  $S[j]$ .

## 4.2 The Basic Protocols

### 4.2.1 CoordP

**CoordP** is a centralized protocol run by one of the members of the group, designated as the *coordinator*. There are  $m$  senders in the group, each member  $i$  therefore maintains an  $m$ -element sequence number array  $R_i$  where its  $j$ -th element  $R_i[j]$  is the maximum sequence number in the sense that all messages with lower sequence numbers from sender  $j$  have arrived at member  $i$ . Three types of messages are used in this protocol: the START message has an empty body<sup>1</sup>, and the ACK and INFO messages are of size  $4m$  bytes. There are three steps in a round of execution as shown in Figure 4.1:

- **Step 1:** The coordinator multicasts a START message in the group;
- **Step 2:** After receiving the START message, each member  $i$  sends its sequence number array  $R_i$  to the coordinator as an ACK message by point-to-point links;
- **Step 3:** After collecting sequence number arrays from all the members, the coordinator calculates the stability array  $S = \text{ArrayMin}_{i=1\dots n}(R_i)$ , where  $S[j] = \min(R_1[j], R_2[j], \dots, R_n[j])$ .  $S$  is then multicast in the group as an INFO message. Based on the received  $S$ , any member in the group can label a message received from member  $i$  as stable if the message has a sequence number less than or equal to  $S[i]$ .

---

<sup>1</sup>In implementation of protocols, one field in message header normally designates the message type.

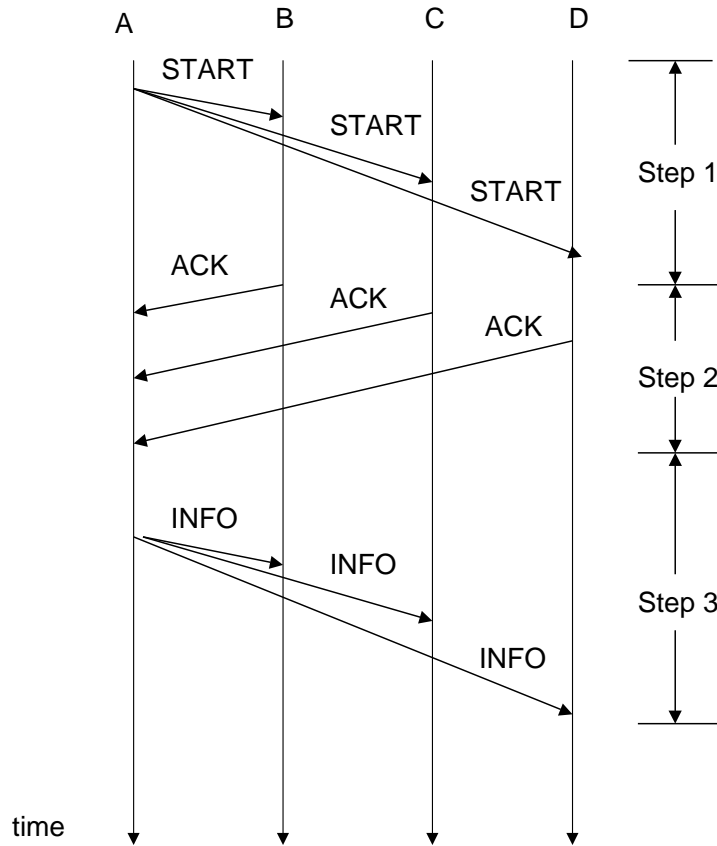


Figure 4.1: Steps of protocol **CoordP**.

In this protocol, there are 2 multicasts by the coordinator and  $n - 1$  point-to-point messages from the non-coordinators. The coordinator sends 1 **START** and 1 **INFO** multicast, it receives 1 **START**, 1 **INFO**, and  $n - 1$  **ACK**s. Therefore, the total number of messages processed by the coordinator is  $n + 3$ , of which 2 messages are sent and  $n + 1$  are received. The multicast is counted as a message sent and received by the coordinator. A non-coordinator sends 1 point-to-point **ACK** message, receives 1 **START** and 1 **INFO** messages. The total number of messages processed by a non-coordinator is 3, of which 1 is sent and 2 are received.

Lost **ACK**s from non-coordinators will prevent the coordinator from getting all

information in the group. This problem is solved by making non-coordinators send unicast ACK messages to the coordinator repetitively.

In an extension to the Tandem global update protocol [Car85] and the Amoeba total ordering protocol [KTHB89], a particular version of **CoordP** is employed as their stability detection algorithm.

The original Tandem global update protocol works as follows. One member is designated as the sequencer of the group. When a sender  $s$  sends a global update message  $u$  to the group, it first sends  $u$  to the sequencer, then the sender sends  $u$  to other group members one by one. At last it sends  $u$  to the sequencer again. Upon receipt of  $u$ , each member sends a positive acknowledgment back to the sender. Message losses are detected by time-outs, and result in message retransmissions.

The Tandem global update protocol allows at most one update to be multicast at a time in the group. Therefore the performance is very bad when the group size is large. In [CdBM94], Cristian et al. propose an extension to the global update protocol called the Positive Acknowledgment or PA protocol that allows concurrent multicasts.

In the PA protocol, a sender  $s$  starts the multicast of an update  $u$  by sending a message  $(u, l)$  to the sequencer where  $l$  is the local sequence number for  $u$ . If the previous message received from  $s$  by the sequencer had local sequence number  $l - 1$ , the sequencer assigns a global sequence number  $k$  to this update and sends a message  $(u, k)$  to every group member. Otherwise, the sequencer stores  $(u, l)$  in a local buffer until the previous message  $(u', l - 1)$  arrives from  $s$  after which it assigns  $u'$  and  $u$  global orders consistent with their origination order at  $s$ , and then multicasts  $u'$  and  $u$  in the group.

Upon receipt of  $(u, k)$ , each member sends a positive acknowledgment for  $k$  to the sequencer. Message losses are detected by time-outs, and result in message retransmissions. Updates are delivered at members in the order imposed by the global sequence numbers attached by the sequencer. Concurrency is allowed among multicast senders as well as multiple multicast requests at a single sender.

Message stability detection works as follows. Group members send to the sequencer the global sequence number  $l$  of the last update they have delivered. The sequencer records them in a sequence number array with one entry per member and multicasts the minimum sequence number  $l_{min}$  stored in the array. Thus, an update  $u$  with global sequence number  $k$  is stable if  $k \leq l_{min}$ .

The Amoeba multicast protocol is also sequencer-based. To initiate the multicast of update  $u$ , a sender sends  $u$  to the sequencer. The sequencer assigns a global sequence number  $k$  to  $u$  and sends messages  $(u, k)$  to all group members.

After having received a message  $(u, k)$ , a member sends a negative acknowledgment to the sequencer only if the next message it receives contains a sequence number greater than  $k+1$ . The sequencer retransmits messages only upon receiving such negative acknowledgments. Concurrency is allowed among different senders. But each sender handles only one multicast at a time. The message stability is established in the same manner as for the PA protocol.

In both systems, there is a sequencer (or a coordinator in our terminology) which assigns the global sequence number to each data message. Other members periodically send to the sequencer a 4-byte field — the last consecutive sequence number. After receiving these numbers, the sequencer calculates their minimum and announces the sequence number of the last stable message in the group.

### 4.2.2 FullDist

**FullDist** is a fully distributed protocol in the sense that every member periodically multicasts its information about message stability in the entire group. In **FullDist**, each member keeps a stability matrix  $E$  of size  $n \times m$  because there are  $m$  senders in the group. Matrix element  $E[i, j]$  stores the sequence number of the last message that is sent by sender  $j$  and has been received by member  $i$ . The  $i$ -th row of the matrix at member  $i$  stores the last sequence numbers of messages that have been received by member  $i$  from all the senders of the group. The minimum of the  $j$ -th column represents the last sequence number whose corresponding message is sent by the  $j$ -th sender and has been received by every member. Messages sent from sender  $j$  with this sequence number or lower are stable. This protocol only uses one type of  $4m$ -byte INFO messages. Periodically, each member multicasts its row of its stability matrix  $E$  in the group. For the purpose of measuring how long it takes for each member to detect message stability in Chapter 5, we introduce two steps in one round of execution of **FullDist** as illustrated in Figure 4.2:

- **Step 1:** The first member multicasts the first row of its matrix  $E$  via an INFO message. No significance is attached to the choice of the first member.
- **Step 2:** After receiving the INFO message, the  $i$ -th member multicasts the  $i$ -th row of its stability matrix  $E$  via an INFO message. Every member replaces the  $i$ -th row of its matrix with the received row information from member  $i$ .

As every member maintains its own stability matrix and determines whether or not any data messages in the system are stable, **FullDist** is decentralized with-

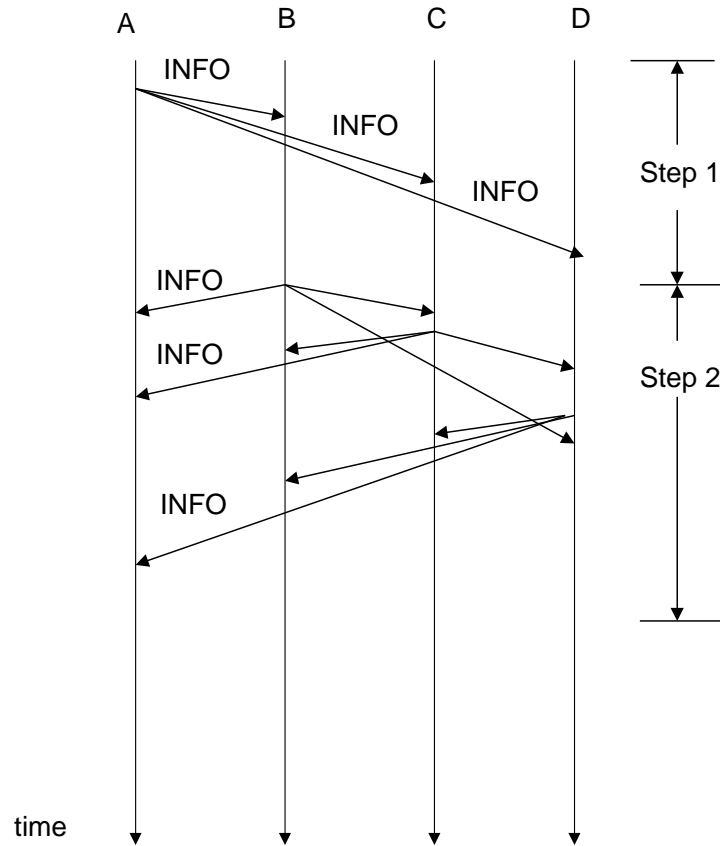


Figure 4.2: Steps of protocol **FullDist**.

out any coordinator. This protocol includes  $n$  multicast **INFO** messages. Every member sends 1 **INFO** multicast and receives  $n$  **INFO** messages. Hence, the total number of messages processed by each member is  $n + 1$ .

As with **CoordP**, this protocol is executed periodically, and as a result, an **INFO** message will compensate for the one lost in the previous step.

**FullDist** is used as the stability detection protocol in Horus [vRBM96], Ensemble [Hay98] and SRM [FJL<sup>+</sup>96].

Horus is a group communication system which offers great flexibility in the properties provided by protocols. It supports dynamic group membership, mes-

sage ordering, synchronization and failure handling. In the Horus architecture, protocols are constructed dynamically by stacking micro-protocols, which support a common interface. Each micro-protocol offers a small integral set of communication properties, and Horus implements them as different layers. One such micro-protocol is the message stability detection protocol. Horus and its next generation system Ensemble [Hay98] implement a set of stability protocols in separate layers so users can pick the appropriate one for their application. **FullDist** is one of the stability protocols offered by Horus and Ensemble.

In SRM, each member periodically multicasts session messages to report the sequence number state for active senders. This is essentially the **FullDist** protocol. The average bandwidth consumed by session messages is limited to a small fraction (for example, 5%) of the aggregate data bandwidth. SRM members dynamically adjust the generation rate of session messages in proportion to the multicast group size.

### 4.2.3 Train

**Train** is a decentralized linear protocol in the sense that a fixed size “train” is passed around group members to spread the message stability information. In the **Train** protocol, each member  $i$  keeps a sequence number array  $R_i$  with  $m$  elements where  $m$  is the number of senders. There is a cyclic order among group members  $1, 2, \dots, n$ . A “train” with a fixed size of  $4m$  bytes passes through all the members in this cyclic order. Member 1 starts the protocol by putting its sequence number array on the train. When the “train” arrives at any other member, this member gets the array from the “train”, calculates the minimum of this array and its own

stability array using *ArrayMin*, then puts the result back in the “train”. After one circulation, the first member gets back in the “train” the minimum of stability arrays of all the members, since the “train” has visited every member once. The “train” containing this minimum array then passes around the group in the circle again. In this second step, every member takes this minimum array and marks stable messages accordingly.

The two types of messages used by the **Train** protocol are the  $4m$ -byte ACK and INFO messages. The first member starts the protocol by assigning  $M_1 = R_1$ , then sending a point-to-point ACK message containing  $M_1$  to the second member. Upon receiving this ACK, the second member calculates  $M_2 = \text{ArrayMin}(M_1, R_2)$ , and sends  $M_2$  to the third member via an ACK. In general, after receiving an ACK message containing  $M_{i-1}$ , member  $i$  calculates  $M_i = \text{ArrayMin}(M_{i-1}, R_i)$ , and sends  $M_i$  on an ACK message to member  $i + 1$ . After an ACK message from member  $n$  arrives at the first member, that is, after the ACK finishes circulating the group once, the first member starts passing  $M_n$  in the same circle in an INFO message. At this point, the stability array  $S$  is constructed since  $S = M_n = \text{ArrayMin}_{i=1\dots n}(R_i)$ , that is,  $S$  contains the sequence numbers for the last stable messages sent from each member. The **Train** protocol is shown in Figure 4.3.

In this protocol, the ACK message needs to circulate the group once, and so does the INFO message. Hence,  $2n$  steps are required. Each member sends out and also receives 1 ACK and 1 INFO message. The total number of messages processed by each member is 4, out of which 2 are sent and 2 are received.

Once this protocol starts, each member repeatedly sends the same message to

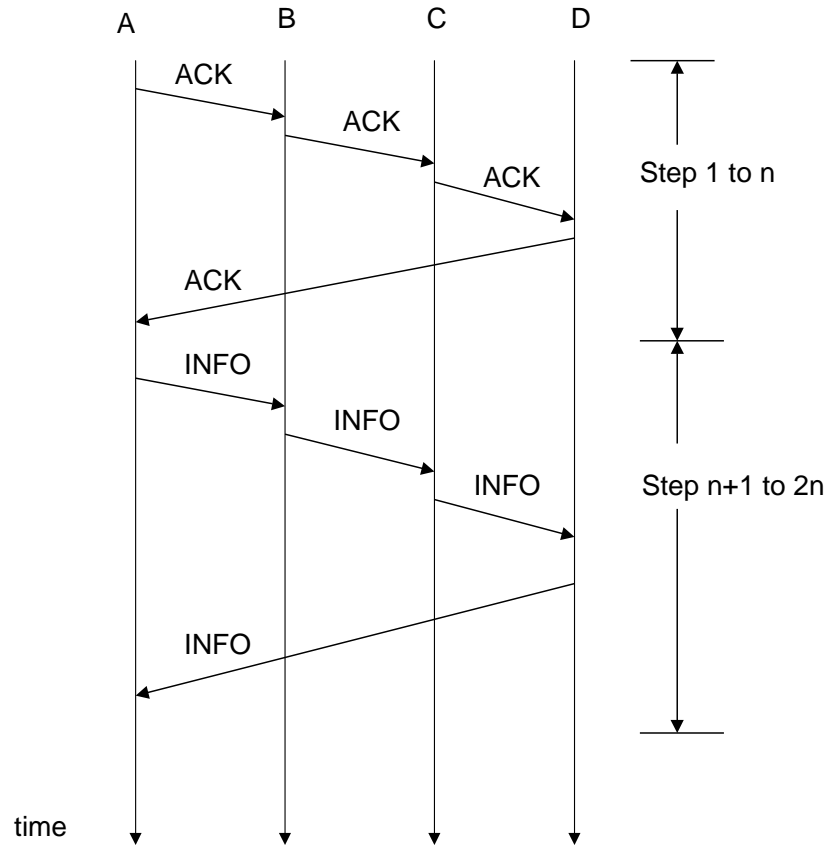


Figure 4.3: Steps of protocol **Train**.

the next member in line. This mechanism effectively combats message losses.

In the Train [Cri91] and Totem single-ring [MMSA<sup>+</sup>96] protocols, the **Train** protocol is used to offer message stability detection.

In the Train protocol, there is a cyclic order among group members. A train containing a sequence of messages circulates from one member to another in this order. In order to multicast a message, a sender waits for the train to arrive. When the train arrives, the sender first delivers all messages carried by the train, and then appends all new messages that it wants to multicast at the end of the train. The sender removes its own multicast messages when it sees the train again.

Group members other than the sender delivers all messages on the train when the train passes by.

If there are no multicasts in progress, the empty train remains idle at some designated group member, the *trainmaster*, and in such a case a sender must request the train in order to multicast a message. A lost train is detected by the trainmaster based on a time-out mechanism. The trainmaster is also responsible for regenerating the train.

A message is detected to be stable when the train completes one more round after delivering the message to all group members. The stability detection protocol used here is a special version of the generic **Train** protocol. When the train circulates in the first round starting at the sender, it delivers the message to all members. After that the sender removes the message from the train. Therefore when the train circulates the second round, it implicitly tells all members this message is stable since the message is not on the train any more.

The Totem single-ring protocol [MSMA91,AMMS<sup>+</sup>95,MMSA<sup>+</sup>96] provides reliable totally ordered delivery of messages using a logical token-passing ring. The token circulates around the ring as point-to-point messages. Only the member holding the token can multicast a message. Unlike the Train protocol, the token does not contain the multicast message. Instead of following the logical ring, messages are multicast in the group by the sender.

In the single-ring protocol, a sequence number field in the token, called *global\_seq* provides global message sequence numbers for all multicast messages, and thus a total order on messages. When a sender multicasts a new message, it increments the *global\_seq* field of the token and gives the message that sequence number. Messages

are delivered with increasing global sequence numbers. Members recognize missing messages by detecting gaps in sequence numbers, and request retransmissions by inserting the sequence numbers of the missing messages into the retransmission request list of the token.

Again, a special version of the **Train** protocol is used here to detect message stability. Each member stores in *seq* its maximum sequence number such that all messages with lower sequence numbers have arrived successfully. As the token passes around the ring, it records the minimum *seq* of the members it has visited so far in its all-received-upto field, or *aru*. After a full token rotation, the *aru* field records a sequence number so that all members on the ring have received all messages with lower sequence numbers. Each member can then reclaim the buffer space used by messages with sequence numbers up to *aru*, because they will never need to be retransmitted.

#### 4.2.4 Gossip

The three basic stability detection protocols we have discussed so far have their limitations on scalability. When the group size is large, an implosion problem will occur at the coordinator in **CoordP**, and at every member in **FullDist** because the number of messages they need to process increases linearly with the group size. For **Train**, the message train needs to traverse the entire group before the stability information is collected, as a result, the time for a round of protocol execution to finish increases linearly with the group size.

To avoid the implosion problem in **CoordP** and **FullDist**, and the linear traversal of all the group members in **Train**, we propose a new protocol called **Gossip**.

The protocol is divided into equally timed steps. During each step, every member constructs a gossip subset consisting of  $p$  distinct members with ranks randomly chosen from 1 to  $n$ . In the first step, every member sends its sequence number array  $R$  to its gossip subset. After receiving a gossip message, a member computes the “Min-so-far” array  $M$  which is the element-wise minimum of sequence number arrays of itself and of other members that it has heard from. It also computes the “Whom-I’ve-heard-from” array as the element-wise maximum of “Whom-I’ve-heard-from” arrays of itself and of other members that it has heard from. In the subsequent steps, every member gossips its “Min-so-far” array  $M$  and its “Whom-I’ve-heard-from” array  $W$  to a different random subset. Instead of sending their information to one coordinator, each member uses gossip messages to disseminate their information in the group step by step. After certain number of steps, one member receives information about all current members, and the “Min-so-far” array  $M$  at this member becomes the stability array  $S$ . This is detected when the “Whom-I’ve-heard-from” bitmap array  $W$  contains 1’s for all current group members.

At this point, the member that detects message stability starts disseminating  $S$  in the group by putting it on the future gossip messages. Upon receiving  $S$ , a member discards stable messages accordingly. To save on bandwidth requirement of future gossip messages and to disseminate  $S$  faster, one could multicast  $S$  in the entire group. Instead of implementing reliable multicast again, an existing reliable multicast protocol can be used. However, this method has a drawback. Some reliable multicast protocols do not guarantee a multicast message to be received by all members in the group. If these protocols are used to distribute  $S$ , there is

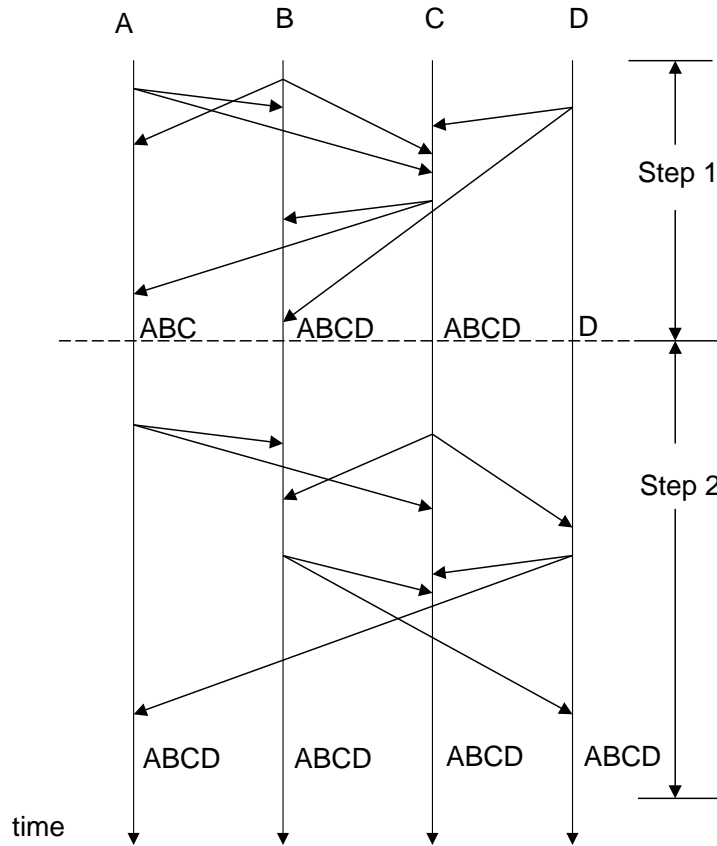


Figure 4.4: An example run of the gossip-style stability detection protocol **Gossip**.

no guarantee that  $S$  will arrive at every member. A hybrid scheme can fix this problem.  $S$  is multicast to the entire group, and periodically  $S$  is piggybacked on future gossip messages to reach members which are left out in the original multicast of  $S$ .

An example is given in Figure 4.4 to illustrate the protocol. The group size is 4, and the subset size for gossip is 2. One round of the protocol is finished after 2 steps. At the end of the first step, members  $B$  and  $C$  construct the stability array, whereas  $A$  and  $D$  only have partial information. During the second step,  $A$  receives  $D$ 's sequence number array from  $D$  directly, and then constructs the

*Notation:*

*ArrayMin* is element-wise minimum of the input arrays.

*ArrayMax* is element-wise maximum of the input arrays.

For two arrays  $A$  and  $B$ ,  $A < B$  means  $A[i] < B[i]$  for all  $i$ .

There are  $n$  members in the group numbered from 1 to  $n$ .

Each member  $i$  keeps four arrays and one number.

$R_i$ : Sequence number array.

$M_i$ : Minimum-so-far array.

$W_i$ : Whom-I've-heard-from array.

$S_i$ : Stability array at the end of the previous round.

$r_i$ : Round number for the current round.

Initially, every member  $i$  has  $M_i = R_i$ ,  $S_i = [0 \dots 0]$ ,  $W_i[i] = 1$ ,  $W_i[j] = 0$  for  $j \neq i$  and  $r_i = 0$ .

Periodically each member sends out a gossip message containing three arrays

and one number —  $(M, W, S, r)$  where  $M = M_i$ ,  $W = W_i$ ,  $S = S_i$ , and  $r = r_i$ .

Figure 4.5: Part I of the stability detection protocol **Gossip**. (In this version, the stability array  $S$  is piggybacked on all gossip messages.)

stability array. Meanwhile,  $D$  obtains the stability array directly from  $B$  and/or  $C$ . At the end of the second step, all 4 members have the stability array, therefore they can discard messages accordingly. In the optimized scheme,  $B$  and  $C$  multicast the stability array in the group at the end of the first step, and only one step is needed to detect stability.

The end of a round of the protocol is reached at each member when the member receives  $S$ . Each member keeps a round number to distinguish gossips from different rounds. The starting points of gossip for group members are scattered randomly during the interval of one step rather than concentrated at the beginning of each time step. This effectively reduces message bursts. The pseudo-code is presented in Figures 4.5 and 4.6.

Every member reacts to received messages as follows:

Upon receipt of a data message, member  $i$  updates  $R_i$ .

Upon receipt of a gossip message  $(M, W, S, r)$ , member  $i$  takes the following actions:

```

if (r == ri) /* receive a message in the current round */
  if (Wi > W) /* this message is redundant */
    do nothing;
  else if (Wi < W) /* the received message is more up-to-date */
    Mi = M; Wi = W;
  else /* normal process */
    Mi = ArrayMin(Mi, M);
    Wi = ArrayMax(Wi, W);
  end if
  Si = ArrayMax(Si, S); /* each round can have many (up to n) concurrent Si's,
  the maximum is the most up-to-date one. */
  if (Wi contains all 1's) /* start next round */
    Si = Mi; ri = ri + 1; Mi = Ri;
    Wi[i] = 1; Wi[j] = 0 for j ≠ i;
  end if
else if (r == ri + 1) /* receive a message from the next round */
  Mi = M; Wi = W;
  Si = ArrayMax(S, Si);
  ri = r;
  if (Wi contains all 1's) /* start another round */
    Si = ArrayMax(Si, Mi);
    ri = ri + 1;
    Mi = Ri;
    Wi[i] = 1; Wi[j] = 0 for j ≠ i;
  end if
else if (r > ri + 1) /* should never receive a message like this */
  error;
else if (r < ri) /* receive a message from previous rounds, ignore, since it is out of date */
  do nothing;
end if

```

Figure 4.6: Part II of the stability detection protocol **Gossip**. (In this version, the stability array  $S$  is piggybacked on all gossip messages.)

It is possible that different stability arrays are constructed at different members at the same time and piggybacked on future gossip messages. Actually during one round of the **Gossip** protocol, there could be  $n$  concurrent stability arrays! This race condition is not harmful because all the stability arrays are valid and only the largest array is the most up-to-date. Therefore as shown in Figure 4.6, whenever a gossip message is received by any member  $i$ , the element-wise maximum of the stability array at member  $i$  and the one on the gossip message is calculated to replace the current stability array at member  $i$ .

The **Gossip** protocol tolerates message losses without requiring a reliable multicast protocol underneath. By periodically sending gossip messages to random sets of group members, this scheme overcomes routing errors, transient link failures and omission failures, because messages are randomly sent to other members, and a message may take a different route in different steps.

The **Gossip** protocol tolerates group membership changes caused by member crashes, leaves, or joins in two ways. First, it incorporates a gossip-style failure detection protocol. Second, it propagates membership changes throughout the entire group using gossip messages, and the normal operation of each group member is not affected when membership change happens.

**Gossip** is implemented in the Ensemble system [Hay98] to aid various scalable group communication.

Under the receiver reliable multicast model where group members do not know the individual addresses of other members, one has to rely on other schemes to construct subsets for gossip messages. One such scheme is to use the Time-To-Live (TTL) [Com95] field in IP packets to limit the scope of a gossip. The TTL field

is initialized to a non-zero number when an IP packet is sent out by the source. Every time the packet is forwarded by a router, the TTL field decrements by one. A router or destination that decrements a packet's TTL field to zero discards the packet. Thus, a source could control how many hops away a packet can travel on the network. Also in the case of a routing loop, the TTL field would eventually kill off the packet.

#### 4.2.5 Analysis of Gossip protocol

The **Gossip** protocol requires two input parameters — the step interval for each gossip step and the subset size for each gossip. In order to find suitable values for them, it is necessary to understand how these parameters affect the probability of an *incomplete* stability detection. A stability detection is *incomplete* if not every member has received sequence number information from all other members.

For simplicity, we analyze the protocol for a system in which the membership of the group is static, that is, no new members join and no existing members fail during the execution of the stability detection protocol. We also assume that group membership is known to all members through the failure detection protocol.

The goal of a gossip-style protocol is to disseminate information in the group step by step, where during each step, each member sends information it has collected to a random subset of the group. The probabilities involved in the **Gossip** protocol can be calculated using epidemic theory (for example, [Bai75,BHO<sup>+</sup>98,DGH<sup>+</sup>87,GT92]). A typical way is to use stochastic analysis given that the execution is broken up into synchronous steps, during which every process gossips once. For a particular sender  $i$ , a member that has received the sequence number array

from sender  $i$  is called *infected*. It is indicated by  $W[i] = 1$  in the “Whom-I’ve-heard-from” array  $W$ .

During each step, the probability that a certain number of members are infected given the number of already infected members is calculated. This is done for each number of infected members. Therefore at the end, the number of steps needed to achieve a large probability that every member is infected can be found.

This analysis becomes very complex if all members are gossiping in each step or if more than one piece of information are propagated in the group using the gossip technique. Usually approximations and/or upper bounds are used instead (for example, see [BHO<sup>+</sup>98]).

We conduct the following analysis based on the same simplification proposed by van Renesse et al. in [vRMH98] and we calculate an upper bound on the number of steps needed to achieve certain probability of an incomplete stability detection as in [vRMH98].

In practice, each member gossips at regular step intervals, but the intervals are not synchronized. In fact, we intentionally scatter the gossips from different members in each step to reduce message burstiness. Message propagation time is typically much shorter than the length of step intervals. We therefore define a different concept of step, called *micro-step*, in which only one (not necessarily infected) member is gossiping. The member is chosen at random and chooses a number of other members to gossip to at random. This set of members constitute a gossip subset of size  $p$ . Thus, in a micro-step, the number of infected members can grow by at most  $p$ . First, to simplify the analysis, we set  $p = 1$ . Later we conduct the same analysis for  $p = 3$ .

### Analysis when the size of gossip subset is one

As a first step, assume only one member's sequence number array needs to be disseminated in the group using the gossip protocol. That is, only one piece of information is propagated.

Let  $n$  be the total number of members,  $k_i$  be the number of infected members in micro-step  $i$ , and  $P_{arrival}$  be the probability that a gossip successfully arrives at a chosen member before the start of the next micro-step. Initially only one member is infected. The number of infected members cannot shrink because we assume group membership remains constant during the execution of the stability detection protocol.

We model the gossip process as a discrete time Markov chain. The state of the system denotes the number of infected members. State transitions occur at the end of each gossip micro-step. We conduct a transient analysis of this Markov chain in which  $P(k_{i+1} = k)$  is used to denote the probability that the total number of infected members becomes  $k$  after micro-step  $i + 1$ .

If  $k$  out of  $n$  members are infected already, then the probability that a gossip is from an infected member is  $\frac{k}{n}$ , the probability that this gossip is sent to an uninfected member is  $\frac{n-k}{n-1}$ , thus the probability that the number of infected members increments by one in a micro-step is

$$P_{inc}(k) = \frac{k}{n} \times \frac{n-k}{n-1} \times P_{arrival} \quad (4.1)$$

The probability that the number of infected members in micro-step  $i + 1$  is  $k$  ( $0 < k \leq n$ ) consists of two parts:

1. The number of infected members in micro-step  $i$  is  $k$  already and does not

increase during micro-step  $i + 1$ .

2. The number of infected members in micro-step  $i$  is  $k - 1$  and increases by one during micro-step  $i + 1$ .

This probability therefore becomes

$$P(k_{i+1} = k) = (1 - P_{inc}(k)) \times P(k_i = k) + P_{inc}(k - 1) \times P(k_i = k - 1) \quad (4.2)$$

Since initially one member is infected, the initial conditions for this Markov chain are the following:

- $P(k_i = 0) = 0$
- $P(k_0 = 1) = 1$
- $P(k_0 = k) = 0$  for  $k \neq 1$

$P(k_r = n)$  is the probability that all the  $n$  members get infected after  $r$  micro-steps. Then, the probability that any member does not get infected by the sequence number information from member  $a$  after  $r$  micro-steps is  $P_{incomplete}(a, r) = 1 - P(k_r = n)$ . To find  $P_{incomplete}(r)$ , which is the probability that any member is not infected by sequence number information from any other member, we need to range over all members.

For any two members  $a$  and  $b$ ,  $P_{incomplete}(a, r)$  and  $P_{incomplete}(b, r)$  are not independent. We bound  $P_{incomplete}(r)$  using the Inclusion-Exclusion Principle [Koz91]:  $Pr(\cup_i A_i) \leq \sum_i Pr(A_i)$ . Using this principle, we can bound  $P_{incomplete}(r)$  as follows:

$$P_{incomplete}(r) \leq n \times P_{incomplete}(a, r) = n \times (1 - P(k_r = n)) \quad (4.3)$$

From this we can calculate now many micro-steps are needed to achieve a certain quality of stability detection. The calculation is done iteratively starting at the first micro-step.

### **Analysis when the size of gossip subset is three**

When the size of gossip subset is greater than one, the analysis is more complicated. Here we present a case for  $p = 3$ , and analysis for other values of  $p$  can be conducted in a similar fashion.

Again, we assume only the sequence number information from one member needs to propagate through the group and only one member is infected initially. Each gossip message is sent from one source to three *different* destinations.

If  $k$  out of  $n$  members are infected already, then the probability that a gossip is from an infected member is  $\frac{k}{n}$ .

The probability that the number of infected members is incremented by one in a micro-step consists of three parts because there are three scenarios where this can happen.

1. One copy of the gossip message is sent to an uninfected member, two copies are sent to two different already infected members and all copies arrive successfully at their destinations. The process can be divided into two stages. In the first stage, three out of three messages arrive successfully. In the second stage, one message is sent to an uninfected member, two are sent to already

infected members. The probability for this event to happen is

$$\begin{aligned}
 P_1 &= \frac{k}{n} \times \binom{3}{3} \times P_{arrival}^3 \times \frac{\binom{n-k}{1} \binom{k-1}{2}}{\binom{n-1}{3}} \\
 &= 3 \times \frac{k}{n} \times \frac{n-k}{n-1} \times \frac{k-1}{n-2} \times \frac{k-2}{n-3} \times P_{arrival}^3 \quad (4.4)
 \end{aligned}$$

2. One copy of the gossip message arrives at an uninfected member, another copy arrives at an infected member and yet another copy is lost. This process can be separated into two stages. In stage one, two out of three messages arrive at their destinations successfully. In stage two, one of these two messages selects an uninfected member as its destination and the other selects an infected member. The probability for this event is

$$\begin{aligned}
 P_2 &= \frac{k}{n} \times \binom{3}{2} \times P_{arrival}^2 \times (1 - P_{arrival}) \times \frac{\binom{n-k}{1} \binom{k-1}{1}}{\binom{n-1}{2}} \\
 &= 6 \times \frac{k}{n} \times \frac{n-k}{n-1} \times \frac{k-1}{n-2} \times P_{arrival}^2 \times (1 - P_{arrival}) \quad (4.5)
 \end{aligned}$$

3. One copy of the gossip message arrives at an uninfected member, the other two copies are lost. Again, this process can be divided into two stages. In the first stage, one out of three messages arrive at its destination successfully. In the second stage, this message picks a destination from the set of uninfected

members. The probability for this to happen is

$$\begin{aligned}
 P_3 &= \frac{k}{n} \times \binom{3}{1} \times P_{arrival} \times (1 - P_{arrival})^2 \times \frac{\binom{n-k}{1}}{\binom{n-1}{1}} \\
 &= 3 \times \frac{k}{n} \times \frac{n-k}{n-1} \times P_{arrival} \times (1 - P_{arrival})^2 \tag{4.6}
 \end{aligned}$$

Adding up these three parts, we have the probability that the number of infected members is incremented by one in a micro-step is

$$P_{inc}(k, 1) = P_1 + P_2 + P_3 \tag{4.7}$$

There are two cases where the number of infected members can increase by two.

1. Two copies of the gossip message are sent to different uninfected members.

The other copy is sent to an infected member. None of these copies are lost.

This process can be separated into two stages. In stage one, three out of three messages are picked to arrive successfully. In stage two, two of these messages choose their destinations from the set of uninfected members and one choose from the set of infected members. This event's probability is

$$\begin{aligned}
 P_4 &= \frac{k}{n} \times \binom{3}{3} \times P_{arrival}^3 \times \frac{\binom{n-k}{2} \binom{k-1}{1}}{\binom{n-1}{3}} \\
 &= 3 \times \frac{k}{n} \times \frac{n-k}{n-1} \times \frac{n-k-1}{n-2} \times \frac{k-1}{n-3} \times P_{arrival}^3 \tag{4.8}
 \end{aligned}$$

2. Two copies of the gossip message arrive at different uninfected members successfully, and the other copy is lost. Again, this process can be divided into two stages. In the first stage, two out of three messages arrive successfully. In the second stage, both messages select their destinations from the set of uninfected members. The probability is

$$\begin{aligned}
 P_5 &= \frac{k}{n} \times \binom{3}{2} \times P_{arrival}^2 \times (1 - P_{arrival}) \times \frac{\binom{n-k}{2}}{\binom{n-1}{2}} \\
 &= 3 \times \frac{k}{n} \times \frac{n-k}{n-1} \times \frac{n-k-1}{n-2} \times P_{arrival}^2 \times (1 - P_{arrival}) \quad (4.9)
 \end{aligned}$$

Therefore, the probability that the number of infected members increases by two is

$$P_{inc}(k, 2) = P_4 + P_5 \quad (4.10)$$

There is only one scenario where the number of infected members increases by three, which is when all three copies of the gossip message arrive at three different uninfected members successfully. The probability for this is

$$\begin{aligned}
 P_{inc}(k, 3) &= \frac{k}{n} \times \binom{3}{3} \times P_{arrival}^3 \times \frac{\binom{n-k}{3}}{\binom{n-1}{3}} \\
 &= \frac{k}{n} \times \frac{n-k}{n-1} \times \frac{n-k-1}{n-2} \times \frac{n-k-2}{n-3} \times P_{arrival}^3 \quad (4.11)
 \end{aligned}$$

The probability that the number of infected members in micro-step  $i + 1$  is  $k$  ( $0 < k \leq n$ ) consists of four parts: the first part comes from the case where

the number of infected members in micro-step  $i$  is  $k$  and does not increase, the second part comes from the case where the number of infected members is  $k - 1$  and increases by one, and so on. This probability is

$$\begin{aligned}
P(k_{i+1} = k) &= (1 - P_{inc}(k, 1) - P_{inc}(k, 2) - P_{inc}(k, 3)) \times P(k_i = k) \\
&\quad + P_{inc}(k - 1, 1) \times P(k_i = k - 1) \\
&\quad + P_{inc}(k - 2, 2) \times P(k_i = k - 2) \\
&\quad + P_{inc}(k - 3, 3) \times P(k_i = k - 3)
\end{aligned} \tag{4.12}$$

(when  $k \geq 3$ ).

There are two special cases for this probability when  $k = 1$  and  $k = 2$ . Since initially one member is infected, the probability that one member is infected in micro-step  $i + 1$  only comes from the case where no member gets infected in that micro-step. Thus,

$$P(k_{i+1} = 1) = (1 - P_{inc}(1, 1) - P_{inc}(1, 2) - P_{inc}(1, 3)) \times P(k_i = 1) \tag{4.13}$$

Similarly, the probability that two members are infected in micro-step  $i + 1$  comes from either of the two cases in which no member gets infected or exactly one member gets infected. The probability is

$$\begin{aligned}
P(k_{i+1} = 2) &= (1 - P_{inc}(2, 1) - P_{inc}(2, 2) - P_{inc}(2, 3)) \times P(k_i = 2) \\
&\quad + P_{inc}(1, 1) \times P(k_i = 1)
\end{aligned} \tag{4.14}$$

Since initially one member is infected, the initial conditions for this Markov chain are the following:

- $P(k_i = 0) = 0$

- $P(k_0 = 1) = 1$
- $P(k_0 = k) = 0$  for  $k \neq 1$

$P(k_r = n)$  is the probability that all the  $n$  members get infected after  $r$  micro-steps. As with the subset size 1 case, the probability that any member does not get infected by the sequence number information from member  $a$  after  $r$  micro-steps is  $P_{incomplete}(a, r) = 1 - P(k_r = n)$ .  $P_{incomplete}(r)$ , the probability that any member is not infected by sequence number information from any other member is bounded by

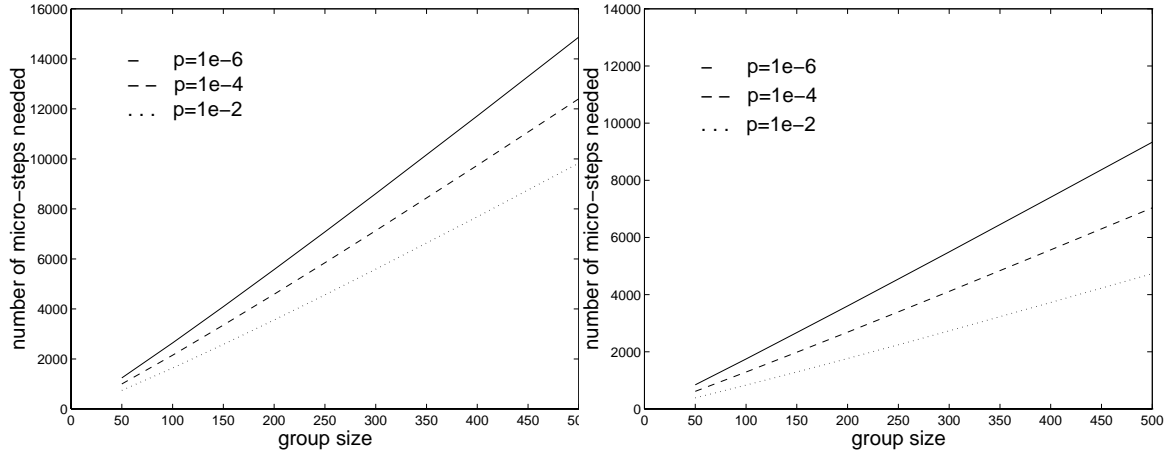
$$P_{incomplete}(r) \leq n \times P_{incomplete}(a, r) = n \times (1 - P(k_r = n)) \quad (4.15)$$

### Comparing results from the analysis

We are interested in the time that it takes to reach a certain low probability of incomplete stability detection being made. By iterating  $P_{incomplete}$ , we can find how many micro-steps this takes. The probability that a gossip message arrives at its destination in a micro-step is set to  $P_{arrival} = 1$ . This is reasonable, since all that is required is that gossip arrives at the member before the member itself gossips next.

We have plotted the number of micro-steps required for different values of  $n$ ,  $P_{incomplete}$ , and subset size in Figure 4.7. The near-linearity in  $n$  can be confirmed visually. The growth of these curves is actually on the order of  $n \log(n)$ , which will become clear in the next paragraph.

In real execution of the protocol, every member gossips once during each step interval. We are interested in the number of steps needed for different incomplete probability. Dividing the number of micro-steps by the group size, we get the



(A): subset size = 1

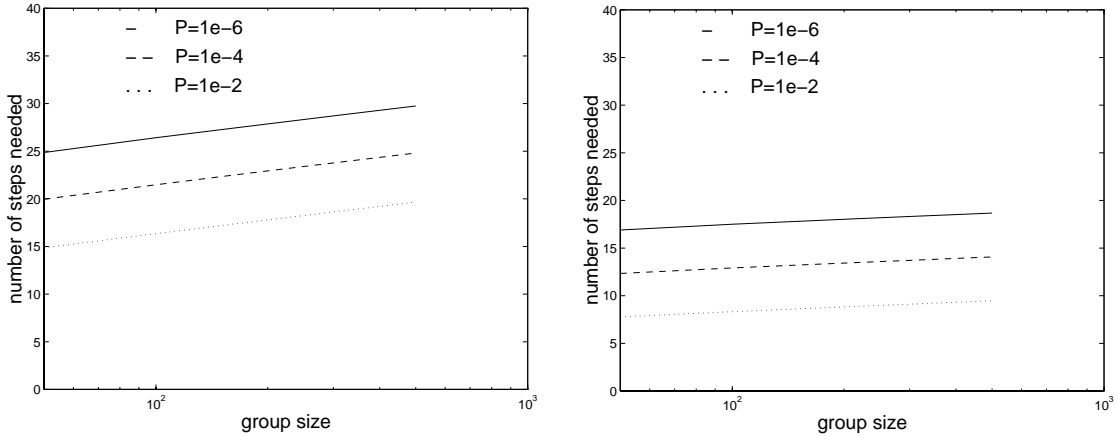
(B): subset size = 3

Figure 4.7: Number of micro-steps needed to achieve different probability of incomplete stability detections.  $P$  stands for  $P_{incomplete}$  in the figure.

number of steps needed. From the results plotted in Figure 4.8 in log scale, we can see clearly a logarithmical increase with group size. This confirms that the number of micro-steps needed in Figure 4.7 is on the order of  $n \log(n)$ .

From Figures 4.7 and 4.8, we see that the increase of subset size decreases the number of micro-steps and steps needed to detect stability. When the subset size changes from 1 to 3, this decrease is only about 55%. But the number of messages sent out in the system per micro-step increases three times! This analysis suggests that it is more beneficial to use a small subset size. It is better to gossip to one member 55% faster to get the same time and quality as gossiping to three members during each gossip.

There is a trade-off between minimizing the probability of making an incomplete stability detection, and the number of steps needed for stability detection. In



(A): subset size = 1

(B): subset size = 3

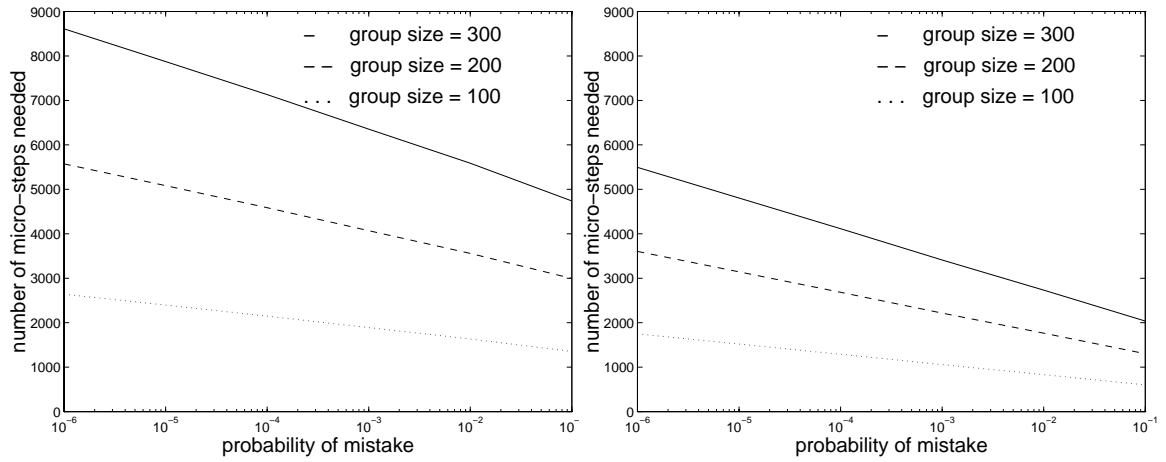
Figure 4.8: Number of steps needed to achieve different probability of incomplete stability detections.  $P$  stands for  $P_{incomplete}$  in the figure.

Figures 4.9 and 4.10, we show that this trade-off is satisfactory, in that only a few extra gossip steps are necessary for better quality.

Notice in the optimized protocol execution, once any member collects message stability information, it multicasts to the entire group. The analysis conducted above gives the upper bound for the actual number of steps needed.

### 4.3 The Structured Protocols

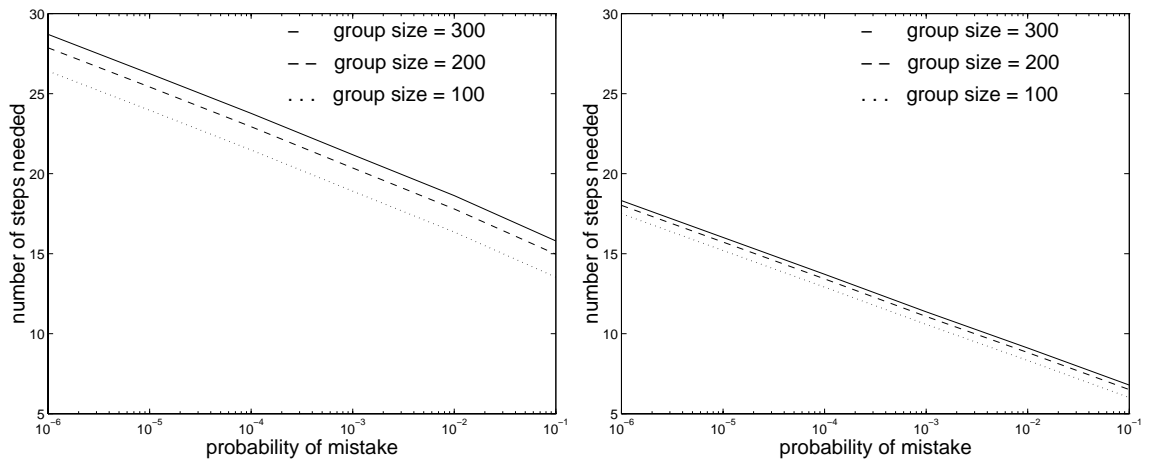
It is a common practice to use hierarchical structures to improve scalability of communication protocols. But the improved scalability comes with complexity of managing the structure. On the other hand, since many reliable multicast protocols employ built-in local groups [HSC95,SLPB97], their group division can be used by



(A): subset size = 1

(B): subset size = 3

Figure 4.9: Cost of quality in terms of number of micro-steps.



(A): subset size = 1

(B): subset size = 3

Figure 4.10: Cost of quality in terms of number of steps.

the structured protocols we are proposing. This section focuses on the scalability aspect of the structured protocols.

Hierarchical structures have been used in various reliable multicast protocols in distributed systems [BFvR97,GVvR96,Mit97,MMSA<sup>+</sup>96] and barrier synchronization algorithms [MCS91] in parallel systems. To improve scalability significantly, we derive three structured stability detection protocols by adding a spanning tree structure to the basic protocols. They are called **S\_CoordP**, **S\_Train**, and **S\_Gossip** since they are derived from **CoordP**, **Train**, and **Gossip** respectively. As we will see later, **S\_FullDist** is equivalent to **S\_CoordP**.

To describe the structured protocols, we use the same assumptions made in Section 4.1. We also assume the group with  $n$  members is organized into a complete tree with height  $p$  and fixed branching factor  $b$  at all levels. The root of the tree is the first member. The size of the group can be expressed by tree parameters  $b$  and  $p$  as  $n = 1 + b + b^2 + \dots + b^p = \frac{b^{p+1}-1}{b-1}$ . The sibling members and their parent constitute a sub-group of size  $b + 1$ . In the following description, a node is identified by a pair  $(d, e)$ , where  $d$  is its depth and  $e$  is its location on level  $d$  of the tree. On level  $d$ , the left most node is labeled as 1 and the right most node is labeled as  $b^d$ . Hence  $d$  ranges from 0 to  $p$  and  $e$  ranges from 1 to  $b^d$ . Each node  $(d, e)$  also has a global number  $q = 1 + b + b^2 + \dots + b^{d-1} + e$ .

### 4.3.1 S\_CoordP

We derive **S\_CoordP** from **CoordP** by employing a hierarchical structure. We assign the root to be the coordinator. Each member  $(d, e)$  maintains a sequence number array  $R_{(d,e)}$  with  $m$  elements. Three types of messages are used: an empty

START message, and  $4m$ -byte ACK and INFO messages. The protocol has  $p + 2$  steps as illustrated in Figure 4.11:

- **Step 1:** The root starts the protocol by multicasting a START message.
- **Step 2:** After receiving START, each leaf member sets  $M_{(p,e)} = R_{(p,e)}$  and sends a point-to-point ACK message containing  $M_{(p,e)}$  to its parent with node number  $(p - 1, l(e))$  where  $l(e) = \text{ceiling}(\frac{e}{b})$ .
- **Step 3 to p+1:** (The tree height is  $p$ , hence  $p - 1$  steps are needed for the ACKs from the next to bottom level to reach the root). Upon receiving  $M_{(d+1,e)}$  from all its children, an internal member  $(d, c)$  (that is, a member that is neither the root nor a leaf) sets  $M_{(d,c)} = \text{ArrayMin}(R_{(d,c)}, M_{(d+1,e)})$  over  $e = (c - 1)b + 1, (c - 1)b + 2, \dots, cb$ . It then sends  $M_{(d,c)}$  to its parent.
- **Step p+2:** With all the  $M_{(1,e)}$ 's collected from its children, the root sets the stability array  $S = M_{(0,1)} = \text{ArrayMin}(R_{(0,1)}, M_{(1,e)})$  over  $e = 1, 2, \dots, b$ . Then it multicasts an INFO message containing the stability array  $S$ . After receiving  $S$ , a member can label any message from member  $i$  stable if it has a sequence number less than or equal to  $S[i]$ .

In this protocol, there is 1 multicast of START, 1 multicast of INFO from the coordinator located at the root and  $n - 1$  point-to-point ACK messages from other members. The root sends 1 START and 1 INFO multicast, while it receives 1 START, 1 INFO and  $b$  ACKs. Hence, the number of messages processed by the root is  $b + 4$ , out of which 2 are sent and  $b + 2$  are received. An internal member sends 1 point-to-point ACK message, and receives 1 START, 1 INFO and  $b$  ACK messages. Therefore the number of messages processed is  $b + 3$ , out of which 1 is

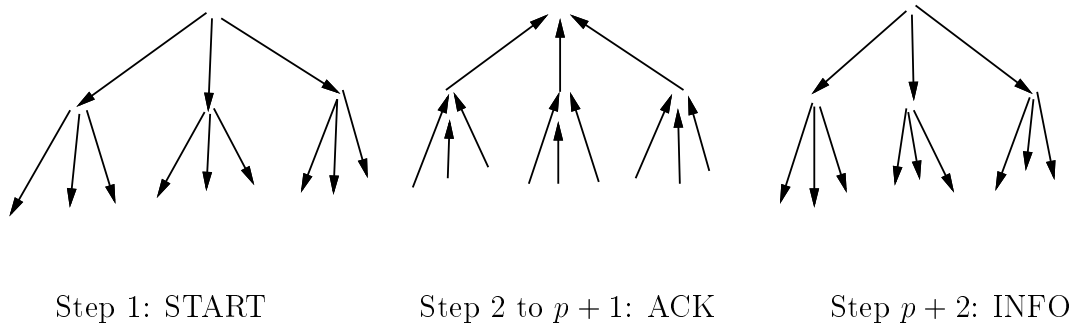


Figure 4.11: Steps of protocol **S\_CoordP**.

sent and  $b + 2$  are received. A leaf member sends 1 point-to-point ACK message, receives 1 START and 1 INFO message. The number of messages processed is 3, out of which 1 is outgoing and 2 are incoming.

In the tree structure, each interior node belongs to a lower sub-group with all its children as well as an upper sub-group with its parent and all of its siblings. Adding a tree structure to **FullDist** results in a protocol where each interior node multicasts its sequence number array in its lower sub-group and after collecting information from its children, it then multicasts the combined information in its upper sub-group. But this multicast is redundant — a member only needs to report its sequence number array to its parent in order for the information to propagate to the top of the tree. After changing the redundant multicast into a point-to-point message to its parent, the protocol becomes **S\_CoordP**.

As in **CoordP**, ACK messages are sent out repeatedly in order to combat message losses. Also, the hierarchy is designed such that each member knows about all of its ancestors, so that when its parent crashes, it connects to its grandparent.

### 4.3.2 S\_Train

Adding a tree structure to **Train** results in **S\_Train** [AMMSB95]. We assign the root to be the coordinator. There is a cyclic order  $1, 2, \dots, b$  among sibling members of each sub-group. Each member  $(d, e)$  maintains a sequence number array  $R_{(d,e)}$  with  $m$  elements. This protocol uses four types of messages: the empty START message, the 4m-byte ACK1, ACK2 and INFO messages. As in Figure 4.12,  $pb + 2$  steps are needed.

- **Step 1:** The root starts the protocol by multicasting an empty START message.
- **Step 2:** After receiving START, the first member  $(p, e)$  of each leaf group sets  $M_{(p,e)} = R_{(p,e)}$  where  $e = 1, b + 1, 2b + 1, \dots, (p - 1)b + 1$ , and sends a point-to-point ACK1 message containing  $M_{(p,e)}$  to the second member of the same leaf sub-group.
- **Step 3 to b+2:** Upon receiving ACK1 from its sibling  $(p, e + i - 1)$ , member  $(p, e + i)$  with  $(0 < i < b)$  sets  $M_{(p,e+i)} = \text{Min}(R_{(p,e+i)}, M_{(p,e+i-1)})$ , then member  $(p, e + i)$  sends  $M_{(p,e+i)}$  on an ACK1 message to member  $(p, e + i + 1)$ . Member  $(p, e + b - 1)$  sends  $M_{(p,e+b-1)}$  on an ACK2 message to its parent member  $(p - 1, l(e + b - 1))$ .
- **Step b+3 to bp+1:** (Each level requires  $b$  steps to pass all its information to its parent node, hence a total of  $bp$  steps are required for all the ACKs to reach the root). After collecting the START and ACK2 from its  $b$ -th child, the first member  $(d, e)$  in an internal sub-group, where  $e = 1, b + 1, 2b + 1, \dots, (d - 1)b + 1$ , sets  $M_{(d,e)} = \text{ArrayMin}(R_{(d,e)}, M_{(d+1,eb)})$ , and then

sends  $M_{(d,e)}$  to the second member  $(d, e + 1)$  of the same sub-group via an ACK1 message. After receiving ACK2 from its  $b$ -th child, and ACK1 containing  $M_{(d,e+i-1)}$  from its sibling  $(d, e + i - 1)$ , a member  $(d, e + i)$  with  $(0 < i < b)$  sets  $M_{(d,e+i)} = \text{ArrayMin}(M_{(d,e+i)}, M_{(d,e+i-1)}, M_{(d+1,(e+i)b)})$ , and then sends  $M_{(d,e+i)}$  via an ACK1 message to its sibling  $(d, e + i + 1)$ , whereas member  $(d, e + b - 1)$  sends  $M_{(d,e+b-1)}$  via an ACK2 message to its parent  $(d - 1, l(e + b - 1))$ .

- **Step  $bp+2$ :** After the root receives an ACK2 message from its  $b$ -th child  $(1, b)$ , it constructs the stability array  $S = M_{(0,1)} = \text{Min}(R_{(0,1)}, M_{(1,b)})$ , then multicasts an INFO message containing the stability array  $S$  in the entire group. After receiving  $S$ , a member can label any message from member  $i$  stable if it has a sequence number less than or equal to  $S[i]$ .

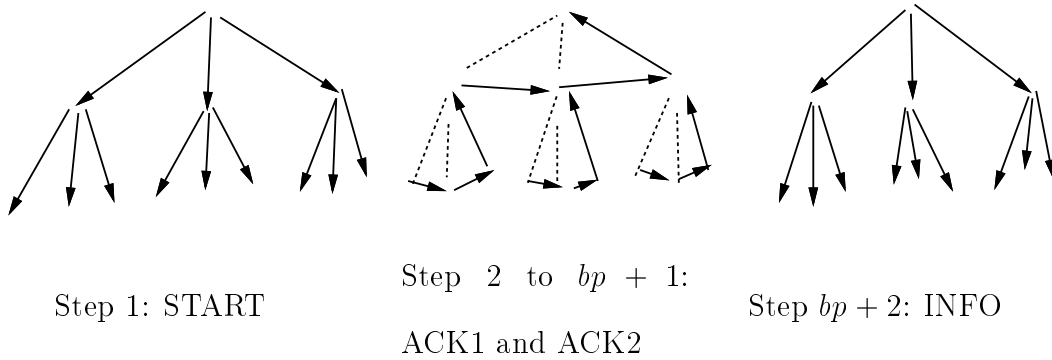


Figure 4.12: Steps of protocol **S\_Train**.

In this protocol, there is 1 multicast of START, 1 multicast of INFO from the root node and  $n - 1$  point-to-point ACK1 and ACK2 messages from other members. The root sends 1 START and 1 INFO multicast, while it receives 1 START, 1 INFO and 1 ACK2 message. Hence, the number of messages processed

by the root is 5, out of which 2 are sent and 3 are received. An internal node sends 1 point-to-point ACK1 or ACK2 message, and receives 1 START, 1 INFO and 1 ACK2 from one of its children. Each internal node except the first sibling in a sub-group also receives 1 ACK1 message from the previous sibling. Thus, an internal node processes 4 or 5 messages depending on its location. A leaf node sends 1 point-to-point ACK1 or ACK2 message, receives 1 START and 1 INFO message. Each leaf node that is not the first sibling in a sub-group also receives 1 ACK1 message from the previous sibling. Therefore, the number of messages processed by a leaf node is 3 or 4, depending on its location.

As with **Train**, each member repeatedly sends the same message to the next member in line to prevent message losses. In **S\_Train**, every member needs to know about all of its ancestors. In case its parent crashes, the sub-group reports to their grandparent. In addition, every member also needs to know about all of its siblings. In case any of the siblings crashes, the train has to follow a different route.

### 4.3.3 S\_Gossip

With the global stability detection framework **Gossip** described in Section 4.2.4, a member gossips to a random set of members during each step. Furthermore, the location of group members on the network does not have any effect on the construction of the random set of members used in each gossip step. Each group member on a subnet propagates its sequence number information by sending gossip messages to some random members either on its local subnet or on some remote subnet.

A more efficient way to detect message stability would be to collect local information on each subnet first, then allow some designated members on each subnet to exchange local information to form the global information.

We propose **S\_Gossip**, a protocol that combines the gossip technique with the concept of local groups. In this scheme, the multicast group is divided into a number of *local groups* according to their location on the network. Each local group has  $G$  Stability Controllers (SCs) where  $G$  is a user-defined parameter which determines how robust the protocol is in case of SC failures. SCs from different local groups constitutes the global *SC group*. In Figure 4.13, there are two subnets connected by a WAN link. Members on different subnets form their own local groups. There are two SCs in each local group. The four SCs then form the global SC group.

The protocol proceeds in two phases. In the first phase, each member gossips to other members in its local group trying to obtain stability information within the local group. After the local stability arrays are constructed, the SCs start the second phase by gossiping among all the SCs. After one SC receives the stability information from SCs representing the other local group(s), the global stability array is constructed and multicast to the entire group.

The global scheme would require group membership information at every member. This is not likely to be feasible in a WAN. The local gossip scheme solves this problem by only requiring each local group member to maintain the addresses of other members on the same subnet.

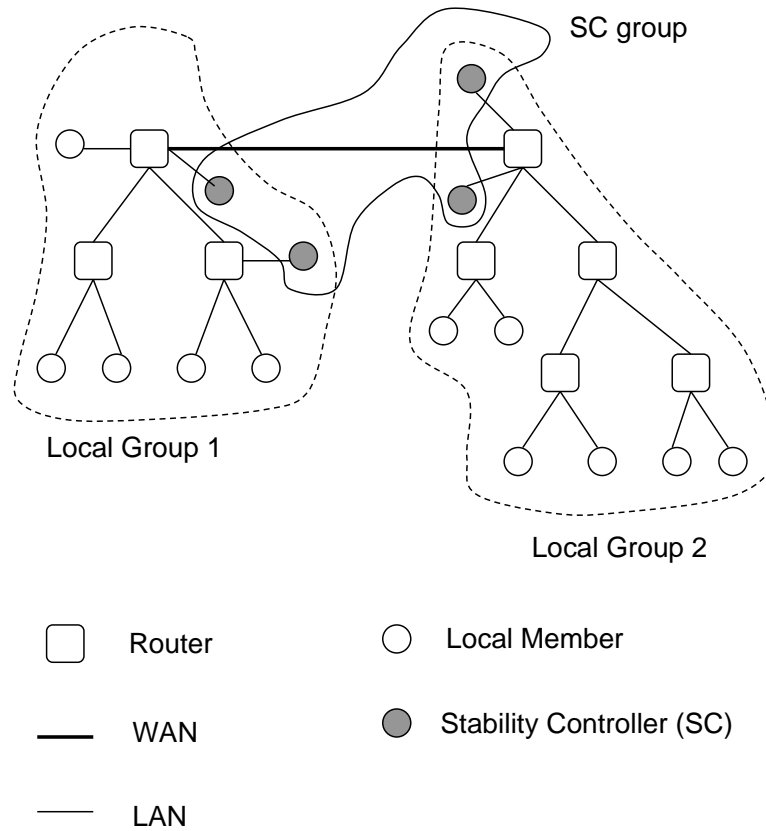


Figure 4.13: Structure of the **S\_Gossip** protocol.

## 4.4 Comparison of the Seven Protocols

This section compares the complexity of the stability detection protocols. The measure for time complexity is the number of steps per protocol round, the measure for processing load is the number of messages processed at each member per round.

From the description of the **Gossip** protocol, one can not determine the exact number of steps needed to achieve message stability. However, each protocol step is separated by a constant time interval and during each step, each member sends out a constant number of messages, therefore on average, each member receives a constant number of messages during unit time. As a result, there is no implosion in

Table 4.1: Complexity of protocols (exact formula)

	<b>CoordP</b>	<b>FullDist</b>	<b>Train</b>	<b>Gossip</b>
# steps	3	2	$2n$	N/A
# messages processed	$n + 3$ (coord) 3 (other)	$n + 1$	4	N/A
	<b>S_CoordP</b>	<b>S_CoordP</b>	<b>S_Train</b>	<b>S_Gossip</b>
# steps	$p + 2$		$bp + 2$	N/A
# messages processed	$b + 4$ (root) $b + 3$ (internal) 3 (leaf)		5 (root) 4 or 5 (internal) 3 or 4 (leaf)	N/A

the gossip-style protocols. From the analysis in Section 4.2.5, we know the upper bound of the number of steps increases logarithmically with group size  $n$ .

For the **S\_Gossip** protocol, the group is divided into a number of local groups, and **Gossip** is executed within each local group. Therefore, the upper bound for the number of steps is still  $O(\log(n))$ . The number of messages processed by each member is still constant during unit time.

For the rest of the protocols, exact formula do exist for both time complexity and processing load complexity. They are summarized in Tables 4.1 and 4.2. The processing load is reduced to  $O(1)$  at every node in **S\_CoordP**. This indicates that message implosion is completely eliminated, therefore better scalability is expected from the structured protocol.

Table 4.2: Complexity of protocols

	<b>CoordP</b>	<b>FullDist</b>	<b>Train</b>	<b>Gossip</b>
# steps	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$
# msgs processed	$O(n)$ (coord) $O(1)$ (other)	$O(n)$	$O(1)$	$O(1)$
	<b>S_CoordP</b>	<b>S_CoordP</b>	<b>S_Train</b>	<b>S_Gossip</b>
# steps	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
# msgs processed	$O(1)$	$O(1)$	$O(1)$	$O(1)$

**S\_CoordP** increases the number of steps to  $O(\log n)$  from  $O(1)$  of the basic protocols **CoordP** and **FullDist**. One might observe that it takes longer for the structured protocol **S\_CoordP** to complete a round than its corresponding basic protocols. However, if the tree in the structured protocols is built according to the physical layout of nodes on the network, a message from a leaf to the root needs to go through  $p$  hops in any case, and the latency is not increased, but rather decreased because of elimination of the implosion problem.

**S\_Train** reduces the number of steps to  $O(\log n)$  from  $O(n)$  of the basic protocol **Train**, while keeping the processing load unchanged at each member. In **S\_Train**, the tree structure is used to help passing the sequence number arrays up to the root concurrently, therefore effectively reducing the latency of the protocol.

## 4.5 Summary

This chapter starts with the assumptions under which the stability detection protocols are presented. Particularly, we assume network partition does not happen. This chapter then describes in detail the four basic protocols and their corresponding structured versions. For each protocol, the algorithm is presented followed by the mechanisms to combat message losses and process failures. Because of the statistical nature of the **Gossip** protocol, a stochastic analysis is conducted to derive the upper bound of the number of gossip steps needed to detect message stability. Finally, the seven protocols are compared in terms of the number of steps per protocol round and the number of messages processed by each member per round.

# Chapter 5

## Simulation of the Stability

## Detection Protocols

Sections 4.2 and 4.3 calculate the number of steps and the number of messages processed by each member for the basic and structured protocols. Comparisons are made for these protocols without considering the importance of the network environment. This section measures other important performance metrics in a number of network topologies.

### 5.1 The Underlying Network

For a given underlying network topology, set of group members, set of senders, and patterns for message sending and message loss, it is possible to analyze the behavior of the various stability detection protocols. However, our interest lies in the performance of these algorithms across a wide range of network topologies and scenarios. For this, we conducted simulations using the ns [MF95] simulator.

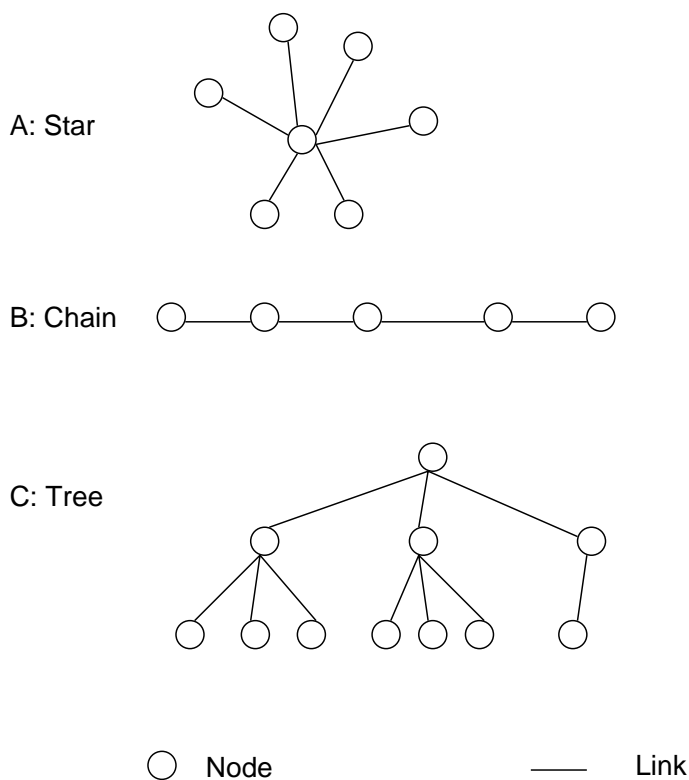


Figure 5.1: Typical network topologies: (A): A star topology, (B): A chain topology, and (C): A bounded-degree tree where interior nodes all have degree 4.

Network topologies can be divided into a few simple, yet representative categories, namely chain, star and tree topologies as shown in Figure 5.1. A tree topology is the most general one since it combines aspects of both chains and stars. Therefore, we conduct our simulation under the tree topology in order to provide a foundation for understanding the behavior of various stability detection protocols in more complex environments.

For a large randomly labeled tree, the probability that a particular node has a degree of at most four approaches 0.98 [Pal85], therefore, the underlying network

used in the simulation is a balanced bounded-degree tree where interior nodes all have degree four. The network topologies are based on some generic network simulation schemes used in [FJL<sup>+</sup>96].

To gain insight on scalability of different stability detection protocols, the simulation is conducted in a set of wide-area networks. Each network in the simulations consists of nodes and links.

We need to assign realistic parameters to links and nodes. A transmission link can be characterized by two parameters: *bandwidth* and *latency*. The *bandwidth* of a link measures its information carrying capacity [Bla90]. Using a plumbing analogy, it is the width of the “information pipe”. A link carries information in the form of analog symbols, where a symbol may correspond to one or more bits. Thus the bit rate of a line, measured in bits per second, is the product of its capacity measured in symbols per second and the mean number of bits each symbol represents.

A voice channel typically requires 64K bps(bits per second) bandwidth, we want to allocate small percentage of the available bandwidth to stability detection protocols, therefore we allocate 30K bps.

Link *latency* is determined by the time taken for a signal to propagate over the medium. Since the speed of light in fiber is approximately  $0.7c$ , where  $c$  (the speed of light in vacuum) is  $3 \times 10^5$  kilometers/second. The speed of light propagation delay on a fiber-optic line is  $0.7 \times 3 \times 10^5$  kilometers/second. This works out to 4.76 microseconds/kilometer.

Assume there is a direct fiber-optic line between two endpoints 4100 kilometers apart, that is, we ignore the effect of queueing and switching delays, then the one-way latency between these two endpoints is  $4100 \times 4.76 = 19516$  microseconds =

19.5 milliseconds.

The one-way propagation delay between New York and San Francisco, a distance of about 4100 kilometers, is about 20 milliseconds. The delay between New York and London, a distance about 5700 kilometers, is about 27 milliseconds. To simulate a wide-area network, it is reasonable to set the latency for each link to be 5 milliseconds.

We assume each link is bi-directional and each direction has a bandwidth of 30K bps allocated for the session messages in various protocols to conduct message stability detection. Message propagation delay on each link is  $w = 5$  milliseconds, which is typical for wide area links. A rate-controlled network is assumed in which the data source is shaping its traffic by delaying packet sends to meet the 30K bps allocated rate requirement.

Under this assumption, the expected time a  $u$ -byte message spends on the wire to travel one link is  $t_0 = w + u/v$ , where  $v$  is the bandwidth. The router processing time for a message is 1 millisecond despite of message size<sup>1</sup>. The time needed for a host to send a message follows the formula  $t_s(u) = 100 + 2(94 + 35u/4000 + 50u/1000) + 50 = 338 + 47u/400$  (microseconds) [AKS96,KP93]. The time needed for a host to receive a message is normally about 10% higher than the sending time since interrupts need to be handled [AKS96]. It is set to  $t_r(u) = 1.1 \times t_s(u)$ . The queuing delays incurred at the hosts and routers are also simulated by ns. The message header size is set to  $h = 32$  bytes which is enough for most transport protocols [AKS96,vR96].

In the ns simulator, multicast messages follow the multicast routing trees that

---

<sup>1</sup>This is valid because the routing function at the router only examines the header of the message before sending it to the appropriate output port.

are provided by underlying multicast routing protocols. In the Internet, these routing trees are constructed using protocols such as Core Based Tree (CBT) [BFC93], Distance Vector Multicast Routing Protocol(DVMRP) [WPD88] and Protocol Independent Multicast (PIM) [DEF<sup>+</sup>94].

For simplicity, the stability detection protocols are tested in the situation where group membership remains unchanged.

## 5.2 Complexity Metrics

Recall from Chapter 4, two primitive metrics (number of steps per round and number of messages processed at each member per round) are used to aid the analysis of complexities of the various protocols. Given the underlying network topology, we can study more accurate complexity metrics.

The most important goal for a message stability detection protocol is to minimize the time to stabilize a message, reducing the buffer space required for data messages at each member to a minimum. The effectiveness of the protocol in achieving this goal is analyzed in terms of time and space.

To measure the time required for various stability detection protocols, we use *Time-Per-Round* (TPR) which is defined as the duration of time between the start of the stability detection protocol and the moment the first member constructs the stability array  $S$ . After the first member constructs  $S$ , it has the option to multicast the array in the group immediately, therefore every member will receive the stability information within one multicast.

A more important time metric is *Time-To-Stable* (TTS), which is defined as the time between the moment a data message is multicast and the moment it is

detected to be stable by at least one member. TTS depends on three things: TPR, the frequency of rounds of the stability detection algorithm, and the underlying reliable multicast protocol. Analysis of TTS requires implementation or simulation of the reliable multicast protocols. If the reliable multicast protocol can deliver a message to all the receivers within  $D$  seconds, the stability detection algorithm is triggered every  $F$  seconds, and it can detect the message's stability within TPR seconds, then the maximum TTS becomes  $D + F + \text{TPR}$  seconds. This means that at most  $D + F + \text{TPR}$  seconds after a message is multicast from a sender, it can be deleted from the network. Since TPR is the factor that is determined by the stability detection protocol, this section only studies TPR, and an analysis on the triggering mechanism for stability detection protocols is presented in Chapter 6.

To measure the space requirement, the queue size at each network node is recorded whenever the node sends or receives a message. The *maximum* and *average of the recorded queue sizes* over all nodes in the network are calculated. They indicate the load of processing message sends and receives, and also indicate how congested the links are.

Normally in applications where the multicast group is large, a small percentage of the members are active sources for data messages. Without loss of generality, the number of senders is set to  $m = 1$  and  $m = 50$  where group sizes range from 50 to 500. We choose  $m = 1$  because this is the absolutely smallest number of senders. We also test the protocols when  $m = 50$ , because 50 is large in a group with 50 members, but relatively small in a group with 500 members.

Recall from Chapter 4, in a group of size  $n$  with  $m$  senders, each gossip message used to detect stability in **Gossip** and **S\_Gossip** contains a 32-byte header, an  $m$ -

element sequence number array  $M$  where each sequence number occupies 4 bytes, an  $n$ -element bitmap array  $W$  where each element is one bit long and an integer round number which has 4 bytes. The overall gossip message size for a group of size  $n$  is  $32 + 4 \times m + n/8 + 4 = 36 + 4 \times m + n/8$  (with some padding to make it aligned in the packet). In the simulation,  $n$  ranges from 50 to 500; and the packet size ranges from 47 to 103 bytes when there is only 1 sender, and from 243 to 299 bytes when there are 50 senders. The message size in other protocols is only  $32 + 4 \times m$  bytes, smaller than the size of gossip message. Since a typical WAN can handle packets shorter than 500 bytes without fragmentation [Pos81], packet fragmentation is not considered in the simulations.

### 5.3 The Gossip Protocol

For a given underlying network topology, set of group members, set of senders, and patterns for message sending and message loss, it is possible to analyze the behavior of the **Gossip** protocol with a fixed step interval, and a fixed subset size for each gossip. However, we are interested in finding the optimal step interval and subset size for a range of network topologies and scenarios.

To investigate the behavior of the protocol in detail, two more metrics are measured. The first is the *average number of steps needed for a round*. The second metric is the *average number of messages sent out by each member during unit time*. This is an indicator of the load the **Gossip** protocol adds to the network. All the metrics are used to investigate the behavior of the protocols under a number of scenarios.

### 5.3.1 Simulation with a fixed group size

For a given group size and a given number of senders, there are two control parameters in the **Gossip** protocol: the step interval and the subset size for each gossip. The goal is to analyze the behavior of the **Gossip** protocol under different values of these two parameters. To see the effect of step interval on the protocol, the step interval is ranged from 1 second to 20 seconds with incremental steps of 1 second. To see the effect of the subset size on the protocol, the subset size is varied from 1 to 5. In a WAN, IP-multicast is not efficient in sending messages to small groups that are constantly changing [DL93], thus  $k$  unicasts are used to send gossip messages to each subset of size  $k$ .

For the fixed group size  $n = 200$  with  $m = 50$  senders, tests are conducted in two categories: the dense test and the sparse test. In the dense test, a balanced bounded-degree tree of size 200 is built, where every node in the tree is a member of the multicast group. In the sparse test, a balanced bounded-degree tree of size 1000 is built. 200 nodes are randomly chosen to be in the multicast group, and the remaining 800 are routers.

In both tests, the following two simulations are conducted:

- I. Every node in the tree has infinite buffer space, thus the stability detection protocol has no message loss.
- II. Every node in the tree only has enough buffer space to store 64 gossip messages for each connected link<sup>2</sup>. Gossip messages arriving at a node with a full buffer are dropped. Additionally, two random gossip messages in each step interval

---

<sup>2</sup>In Unix, the default buffer space for each TCP connection is 32K bytes. Therefore, each buffer can store  $32K/500 = 64$  500-byte messages. To be conservative, the buffer size is limited to 64 gossip messages.

are dropped at the senders.

An intermediate simulation is also conducted where every node in the tree has infinite buffer space, and two random gossip messages in each step interval are dropped at the senders. The results are similar to the first simulation, therefore only presented in Appendices A and B.

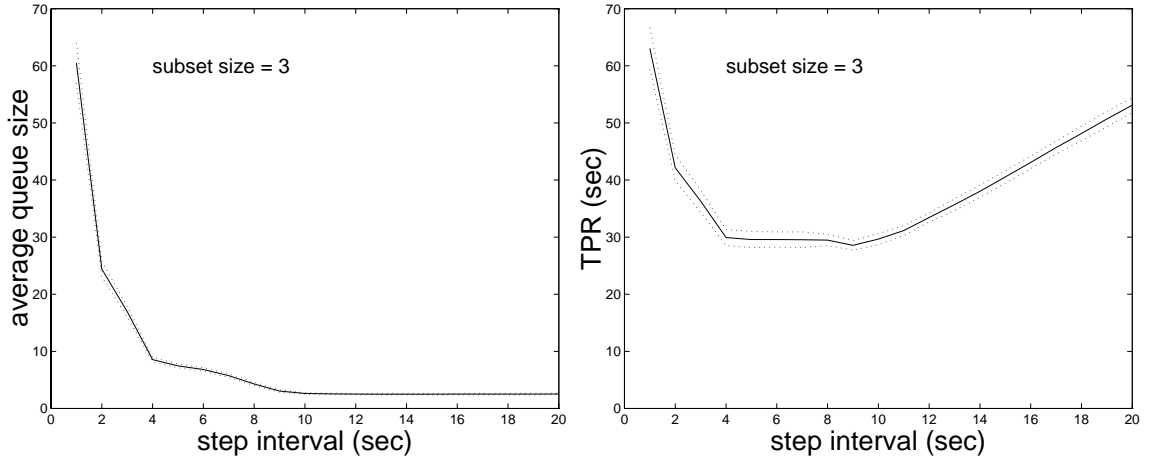
A suite of 20-run statistical tests are conducted where each test has a given subset size and step interval. For dense groups, a random number generator is used for constructing subsets, and each run uses a different seed for the generator. For sparse groups, each run corresponds to a different randomly constructed 200-member group in the 1000-node tree.

The standard deviation of the sampling results follow similar trends in all the simulations. Figure 5.2 presents the sample mean and sample standard deviation for two metrics in simulation I with a dense group of size  $n = 200$  and subset size 3. The two metrics are the average queue size and TPR. The solid line in each figure is the mean and the two dotted lines plot one standard deviation above and below the mean. The largest ratio of standard deviation over mean among 20 tests, each with 20 samples is 7% for the average queue size and 5% for TPR. Under the general assumption that the metrics follow normal distribution, approximately 67% of the sample points fall between the two dotted lines. Since the standard deviation is small, only the mean is reported in the rest of this chapter.

Out of the 12,000 individual runs of the simulations<sup>3</sup>, 17 of them can not detect message stability within 10 minutes. This means that close to 0.15% of the sample

---

<sup>3</sup>The dense and sparse tests each contain 3 simulations where each simulation covers 5 subset sizes and 20 step intervals. 20 sample runs are executed for a pair of subset size and step interval value. The total number of runs is  $2 \times 3 \times 5 \times 20 \times 20 = 12,000$ .



(A): average queue size

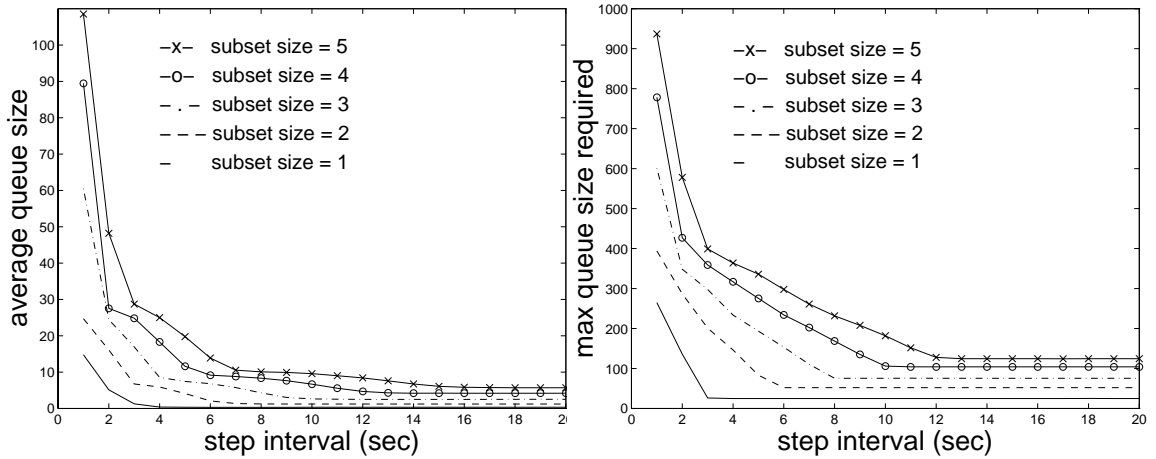
(B): time-per-round (TPR)

Figure 5.2: Simulation I (no message loss) with a dense group of size 200, and subset size 3.

points are bad. These bad samples are excluded from the statistical analysis in the rest of this section. The probability for a round not to finish after a relatively long time exists, but is very slim. In practice, a second round of the stability detection protocol can start with a different seed for generating subsets. The probability that both rounds take an unreasonably long time is even slimmer — 0.0225% ( $0.0015 \times 0.0015 = 0.000225$ ).

### Simulation I of dense groups

Since gossip messages are sent out by unicast, the number of messages sent out by each member during each step is the same as the subset size. For a given data point  $(x, y)$  in the simulation with subset size  $x$  and step interval  $y$ , the number of messages each member sends out per unit time is  $x/y$ . Therefore the traffic load



(A): average queue size

(B): maximum queue size

Figure 5.3: Simulation I (no message loss) with a dense group of size 200 (part I).

generated by the stability detection protocol is proportional to  $x/y$ .

The average queue size reflects message burstiness. When large numbers of messages are sent out during a short amount of time, buffers at hosts and routers near message sources will receive a large number of messages to be processed and forwarded, resulting in large queue sizes. Figure 5.3(A) plots the average queue size recorded over all the nodes in the network.

For any given curve, the decreasing part comes from the fact that as the step interval increases, the burstiness of messages decreases until it reaches a minimum. After a certain point, the increase of step interval will not reduce message burstiness any further. Subset size also influences message burstiness. For any fixed step interval, as the subset size increases, the burstiness from sending to one subset increases, which results in the increase of the average queue size.

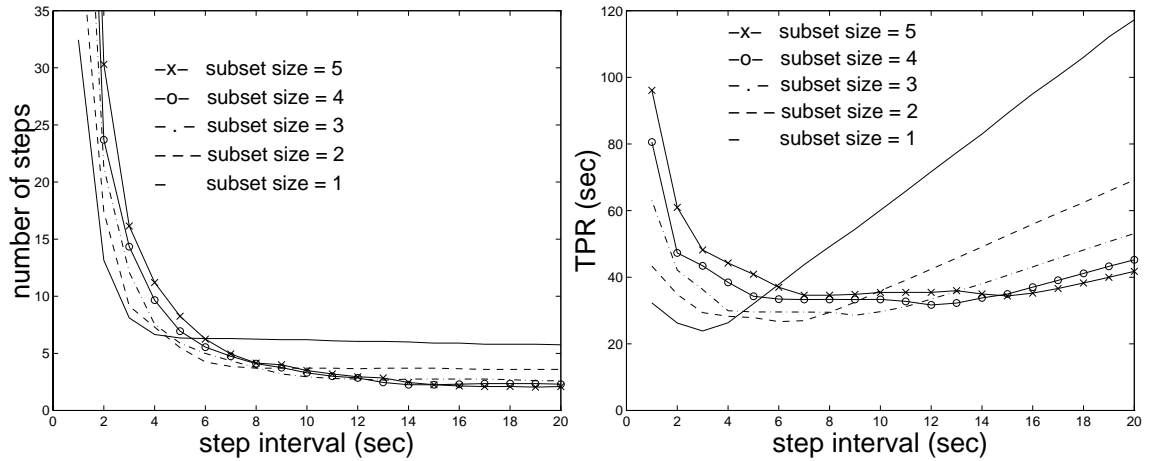
The maximum queue size shows the same trend as the average queue size, and

is presented in Figure 5.3(B).

The goal of the **Gossip** protocol is to eliminate the implosion problem experienced by **CoordP** and **FullDist** when the group size  $n$  is large. As we can see from Figure 5.3, if the step interval is too small, the implosion problem actually happens in **Gossip**! We can see this effect from the dramatic increase of queue sizes when the step interval is reduced pass certain limit. From Figure 5.3 alone we can conclude that for each subset size, there is a smallest acceptable step interval such that any step interval smaller than this limit would cause significant increase of message burstiness which in turn will damage the performance of the **Gossip** protocol.

TPR is the product of the step interval and the number of steps needed in a round. The step interval is a parameter that can be controlled, whereas the number of steps is an indicator of the behavior of the protocol. All the lines in Figure 5.4(A) show the same trend: as the step interval increases, the number of steps decreases to a minimum and remains there with any further increase in the step interval. The step interval at which the minimum number of steps is reached is called the *critical point*.

For a given subset size, before reaching the critical point, a decrease in the step interval results in an increase in the number of steps. As the step interval decreases, each member is scheduled to gossip its sequence number array in a shorter time period. This shorter time period prevents each member from receiving all the available new information. Therefore, more steps are needed to detect message stability. Moreover, the less new information in each step, the more redundant or repeating information flows in the network, increasing network traffic load and



(A): number of steps in a round

(B): time-per-round (TPR)

Figure 5.4: Simulation I (no message loss) with a dense group of size 200 (part II).

postponing the arrival of new information. These factors contribute to the increase in the number of steps needed for detecting message stability.

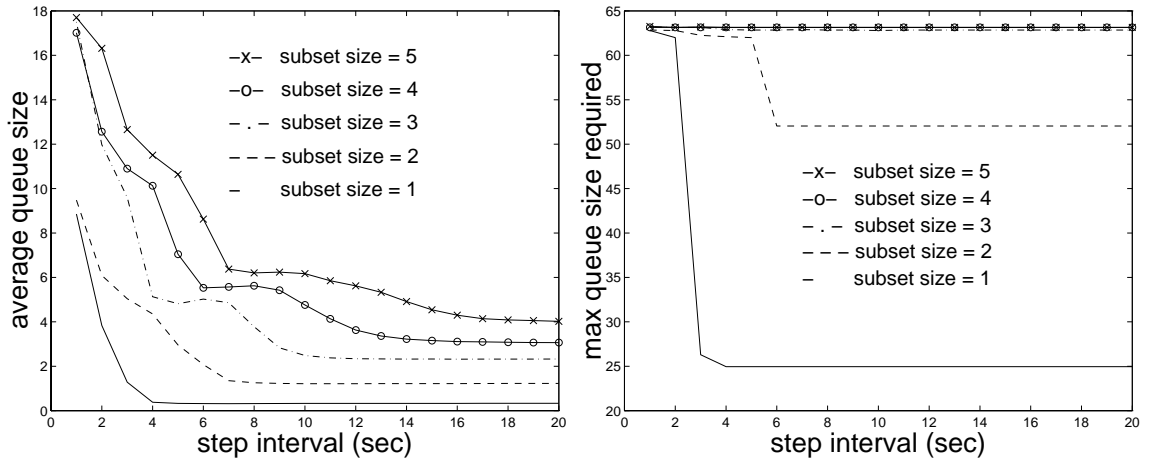
For a fixed step interval less than 4 seconds, that is, before the critical point is reached for any curve, the number of steps increases with the subset size because traffic load increases from more messages sent out by each member during each gossip. After the critical point is reached for all the curves, the step interval is greater than 15 seconds, and gossip messages are scattered far enough to reduce both traffic load and redundant gossips to a minimum. In this situation, the larger the subset size, the more information is exchanged in each step, and a smaller number of steps are needed to reach message stability. When the step interval varies between 4 and 15 seconds, the number of steps goes through a transition from increasing to decreasing with the subset size.

The behavior of the TPR curves plotted in Figure 5.4(B) can be derived from

Figure 5.4(A), because TPR is the product of the number of steps and the step interval. Before reaching the critical point, the steeply declining trend of the number of steps dominates the behavior of TPR, even though the increasing step interval damps the TPR's decline. After passing the critical point, TPR becomes a linear function of the step interval with the coefficient being the value of the number of steps, which is almost a constant. As Figure 5.4(A) shows, the smaller the subset size, the larger the number of steps needed for detecting stability when the step interval passes the critical point. This feature transferred into Figure 5.4(B) says that the smaller the subset size, the steeper the slope of the TPR function is. When the step interval falls in the range between 4 and 15 seconds, the opposite movements of the two components of the TPR function cause TPR to fluctuate in a narrow range from 28 to 40 seconds, except for the case of subset size 1. For each subset size, a window of optimal step intervals exists in which TPR is near-minimum.

### **Simulation II of dense groups**

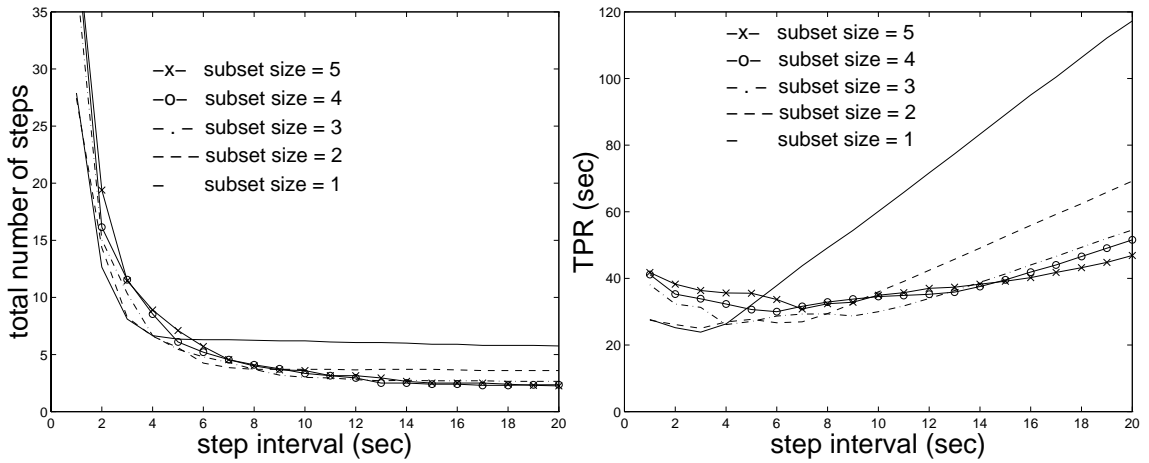
Figures 5.5 and 5.6 present the four metrics in simulation II for a dense group of size 200. When the step interval is smaller than the critical point, a large number of messages are dropped at intermediate and destination nodes because of the 64-message buffer limit. Whereas in simulation I, a lot of bandwidth is wasted carrying gossip messages that have no effect in the determination of stability. This explains the better TPR observed in simulation II than simulation I.



(A): average queue size

(B): maximum queue size

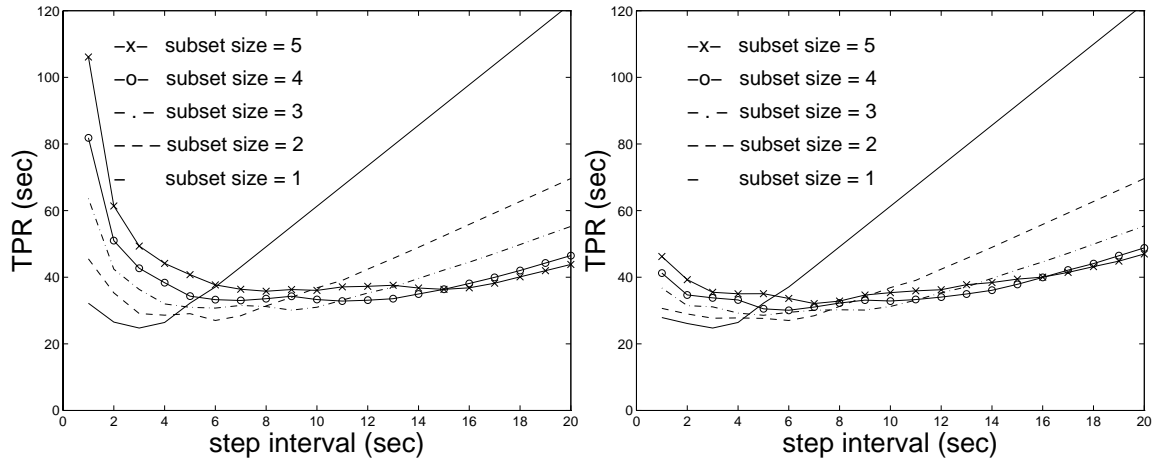
Figure 5.5: Simulation II (queue size = 64) with a dense group of size 200 (part I).



(A): number of steps in a round

(B): time-per-round (TPR)

Figure 5.6: Simulation II (queue size = 64) with a dense group of size 200 (part II).



(A): Simulations I

(B): Simulation II

Figure 5.7: TPR for simulations I and II with 20 sparse groups of size 200.

### Simulations of sparse groups

The same simulation for a dense group and a sparse group of the same size shows no major difference in all the performance metrics. This indicates that location of group members and network topology do not have a noticeable effect on the protocol. This is the result of the combination of the following factors: processing speed, bandwidth and propagation delay. Under the given network condition, bandwidth limitation is the dominating factor in TPR when the step interval is smaller than the critical point, and the length of the step interval dominates TPR when it is larger than the critical point. TPRs for the sparse group simulations are presented in Figure 5.7.

### 5.3.2 Adaptive method: finding the window of optimal step intervals

As observed from Figures 5.4(B), 5.6(B), and 5.7, every TPR curve has a flat portion in which one can select any step interval to achieve a near-minimum TPR. The process of finding the flat portion has two parts. The first part finds a step interval that achieves a near-minimum TPR, and the second part finds a window around that step interval. If TPR can be expressed as an analytical function, then Newton's method [DS93] can find its minimum. Otherwise, better approaches exist in experimental optimization. One such approach is the *golden-ratio method* [han79]. Figures 5.8 and 5.9 present the two-part algorithm for finding the window of optimal step intervals.

Different groups and network topology require different input values for the algorithm. For example, the following input values are given for a 200-member dense group in simulation I:  $g = 2$  seconds,  $a_0 = 0$ ,  $b_0 = 20$  seconds,  $e = 10\%$ , and  $p = 1$  second. For the TPR curve for subset size of 3, part one finishes after 5 iterations, ending up with 9 seconds as the step interval that achieves a minimum TPR of 28.6 seconds. Part two finds the lower bound  $s_1 = 4$  seconds and the upper bound  $s_2 = 10$  seconds after total of 5 iterations of the while loops. Any step interval in the range from 4 to 10 seconds can be used to achieve a TPR between 28.6 to 31.2 seconds. The window sizes are different for different subset sizes. For subset size of 1, the window is only from 2 to 4 seconds. As the group size and network condition change over time, this two-part algorithm is executed periodically to find the current window of optimal step intervals.

As indicated by Figures 5.4(B), 5.6(B), and 5.7, as the subset size increases,

the minimum TPR increases slightly, but the window size of optimal step intervals increases significantly. There is a trade-off between the stability of the protocol and the minimum TPR. If the window size is too small, for example, when subset size is 1, then a slight perturbation of the network condition will result in dramatic increase of TPR if the step interval is unchanged. A large window size is preferred because slight changes in network condition will result in overlapping between the new and current windows, therefore only slight changes in TPR. Also notice the optimal window size is about the same for subset size of 4 and 5. As a result, the recommended subset size is 2, 3 or 4.

*Input:*  $a_0$ : smallest step interval in the search.  
 $b_0$ : largest step interval in the search.  
 $g$ : distance between the two ends when the search stops.  
*Output:*  $s$ : the step interval that achieves a near-minimum TPR.

```

a := a0; b := b0; stop := false;
while ¬stop do
  d := b - a;
  if (d > g) then
    m1 := a + 0.382 * d;
    m2 := a + 0.618 * d;
    if (f(m1) ≥ f(m2)) then a := m1; else b := m2;
  else
    stop := true;
    s := (a + b)/2;

```

Figure 5.8: Part I of the adaptive algorithm: finding a near-minimum TPR.  $f(x)$  is the average measured TPR value for a step interval value  $x$ .

*Input:*  $s$ : the step interval that achieves a near-minimum TPR from part I.  
 $f(s)$ : the near-minimum TPR from part I.  
 $e$ : the fluctuation index. Step intervals that achieve TPR in the range  
from  $(1 - e) * f(s)$  to  $(1 + e) * f(s)$  should be in the window.  
 $p$ : initial search step size.

*Output:*  $s_1$ : lower bound of the window.  
 $s_2$ : upper bound of the window.

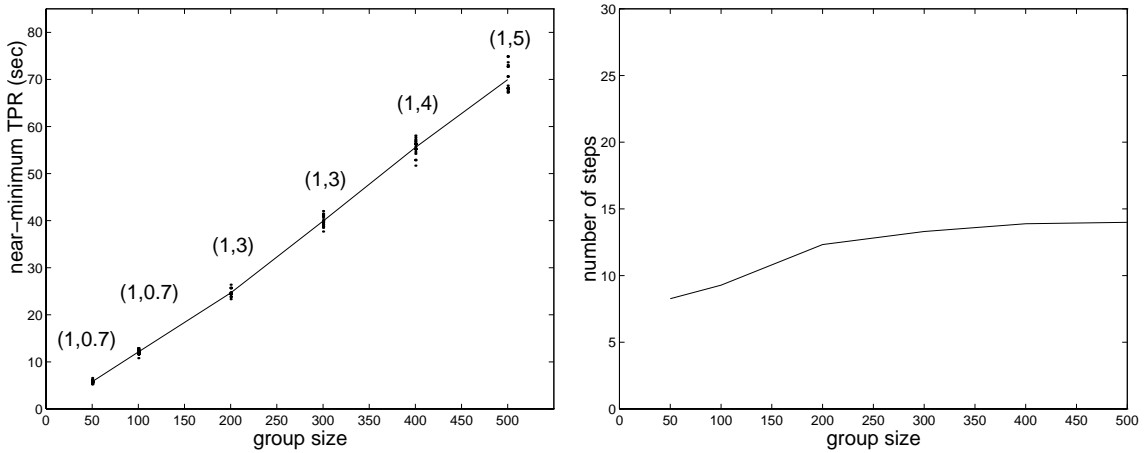
```

 $s_1 := s; k := 1;$ 
while  $((s_1 - k * p > 0) \wedge (|f(s_1 - k * p) - f(s)| < e))$  do
     $s_1 := s_1 - k * p; k := 2 * k;$ 
 $k := k/2;$ 
while  $((k \geq 1) \wedge (|f(s_1 - k * p) - f(s)| < e))$  do
     $s_1 := s_1 - k * p; k := k/2;$ 

 $s_2 := s; k := 1;$ 
while  $(|f(s_2 + k * p) - f(s)| < e)$  do
     $s_2 := s_2 + k * p; k := 2 * k;$ 
 $k := k/2;$ 
while  $((k \geq 1) \wedge (|f(s_2 + k * p) - f(s)| < e))$  do
     $s_2 := s_2 + k * p; k := k/2;$ 

```

Figure 5.9: Part II of the adaptive algorithm: finding the optimal step interval window given a near-minimum TPR.  $f(x)$  is the average measured TPR value for a step interval value  $x$ .



(A): near-minimum TPR

(B): average number of steps needed

Figure 5.10: Near-minimum TPR and the corresponding number of steps needed in a round for sparse groups in Simulation II using the global gossip scheme with 50 senders. A data point with subset size  $x$  and step interval  $y$  seconds is labeled as  $(x, y)$ .

### 5.3.3 Simulation with varying group sizes

Simulations I and II are conducted for dense and sparse groups with various group sizes, and the same pattern is observed as in the tests for 200 members. Simulation II for sparse groups is done for different group sizes in a balanced bounded-degree tree of size 1000, and its near-minimum TPR is presented in Figure 5.10(A). The average number of steps needed in a round to achieve the corresponding near-minimum TPR is presented in Figure 5.10(B).

The group sizes are 50, 100, 200, 300, 400 and 500. For each group size, 20 simulations are conducted with subset size of 1 and an optimal step interval. For

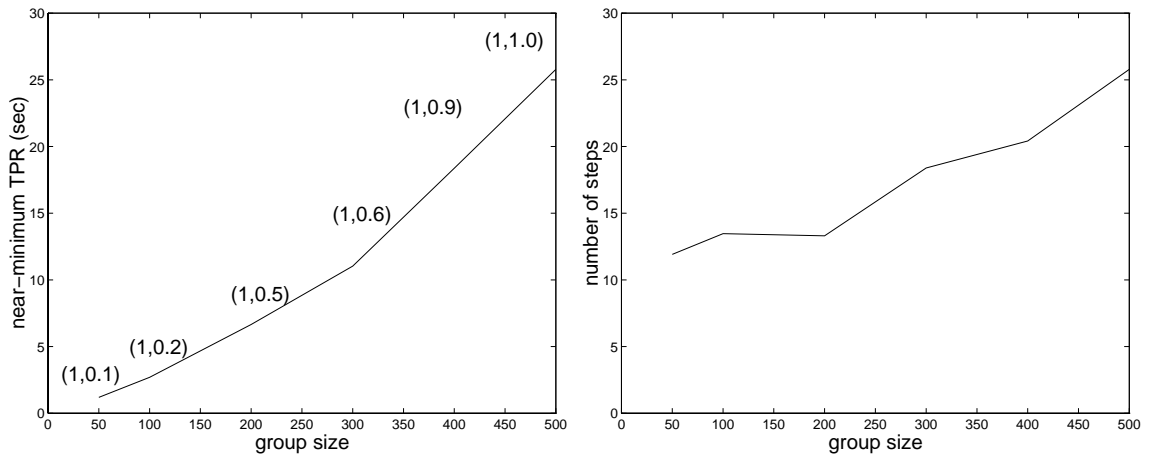
each simulation, a new group is randomly constructed in the 1000-node tree. Each simulation is represented by a dot in Figure 5.10(A). The solid line represents the mean of the TPRs. We can observe that the minimum TPR increases approximately linearly with group size  $n$ . When  $n = 500$ , it reaches 71 seconds. The resulting number of steps needed for each round of the protocol is plotted in Figure 5.10(B). We can see an approximately  $\log(n)$  increase of the number of steps with the group size  $n$ .

The simulations are also conducted with the number of senders  $m = 1$ , subset size of 1 and the optimal step interval corresponding to each group size. The TPR and the corresponding number of steps are presented in Figure 5.11. The optimal step intervals are listed in Table 5.1. We see that the TPR is a lot smaller with one sender than with 50 senders. This is because the message size is reduced by  $4 \times 49 = 196$  bytes from the 50-sender case. As a result, messages are processed faster at its sender and receiver, and spend less time on network links.

The simulations are run with  $30K$  bps network bandwidth allocated to the protocol. The same simulations are also conducted with a  $300K$  bps bandwidth and a 10-time decrease in TPR is observed. More bandwidth combined with an optimal step interval will result in a faster stability detection time.

### 5.3.4 Summary

This section studies the behavior of the **Gossip** protocol under a wide range of network topologies and scenarios. For a fixed group size, there is an optimal combination of the size of step interval and the subset size for each gossip. An adaptive mechanism is presented to find a window of optimal step intervals for a



(A): near-minimum TPR

(B): average number of steps needed

Figure 5.11: Near-minimum TPR and the corresponding number of steps needed in a round for sparse groups in Simulation II using the global gossip scheme with one sender. A data point with subset size  $x$  and step interval  $y$  seconds is labeled as  $(x, y)$ .

Table 5.1: The near-optimal step interval (in seconds) for **Gossip** for different group sizes and different numbers of senders

	1 sender	50 senders
$n = 50$	0.1	0.7
$n = 100$	0.2	0.7
$n = 200$	0.5	3.0
$n = 300$	0.6	3.0
$n = 400$	0.9	4.0
$n = 500$	1.0	5.0

certain subset size. The subset size for each gossip is recommended to be small (2, 3 or 4).

The TPR achieved by setting the optimal step interval increases approximately linearly with the group size. The number of gossip steps needed increases approximately logarithmically with the group size. These results are not surprising because we have seen in Section 4.2.5 that the upper bound of the number of steps is  $O(\log(n))$ .

## 5.4 Comparison of Various Protocols in Dense Groups with 50 Senders

After determining the optimal step interval and subset size for the **Gossip** protocol, we compare all the stability detection protocols in the same network environment. We conduct two categories of tests for each group size  $n$  as in Section 5.3.1: the dense test and the sparse test. In the dense test, a balanced bounded-degree tree of size  $n$  is built, where every node in the tree is a member of the multicast group. In the sparse test, a balanced bounded-degree tree of size 1000 is built.  $n$  nodes are randomly chosen to be in the multicast group, and the remaining  $1000 - n$  are routers. Notice when  $n$  is small, the number of routers is relatively large compared with the group size. But the transport level protocol does not need to use all the routers necessarily. The number of routers actually used is much smaller than  $1000 - n$  for a small  $n$ .

The **Gossip** protocol is simulated under the condition that every node in the tree only has enough buffer space to store 64 gossip messages for each connected link. Gossip messages arriving at a node with a full buffer are dropped. Additionally, two random gossip messages in each step interval are dropped at their senders. The reason is that the **Gossip** protocol tolerates message loss well, and dropping messages can actually improve its performance as shown in Section 5.3.1.

The rest of the protocols are simulated with no message losses and every node in the tree having infinite buffer space, because we are interested in the best performance of the protocols.

Simulations in this section first investigate the best attainable performance

of various protocols. Under this guideline, the logical trees in **S\_CoordP** and **S\_Train** are built to match the underlying network topology, that is, each logical tree has a fixed branching factor of 3. The benefit of **Gossip** lies in its lack of structure, and therefore its robustness to message losses. To show the benefit of a hierarchical approach — **S\_Gossip**, we build a one-level tree where each sub-group has size 50.

Without loss of generality, the coordinator in **CoordP** is located at the root of the network tree.

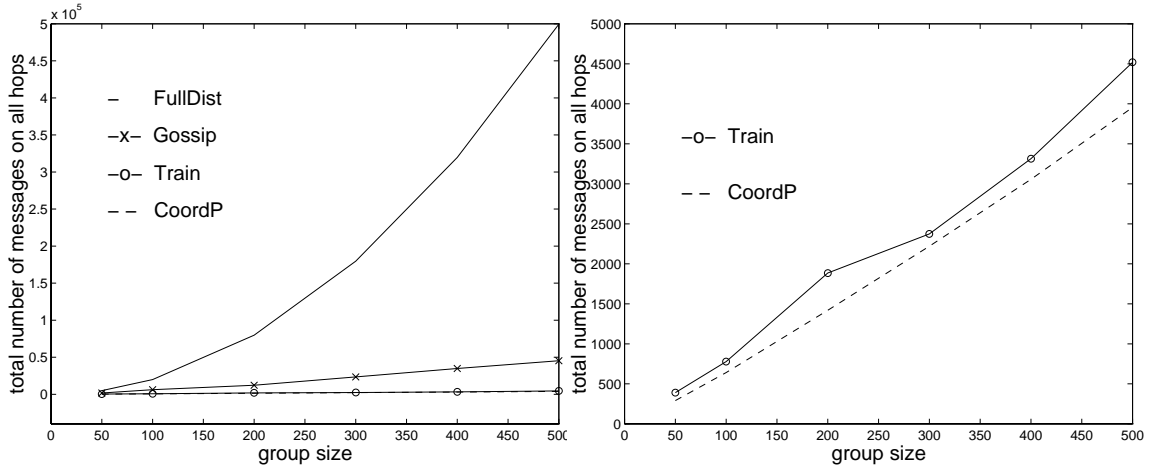
#### 5.4.1 Total number of messages on all hops in the system

In **CoordP**, each non-coordinator member sends a point-to-point message to the coordinator at the root of the underlying tree network. The number of hops each message traverses is at most the height of the tree which is  $O(\log(n))$ . Therefore, the total number of messages on all hops is  $O(n \log(n))$ .

In **FullDist**, each member multicasts a message in the group. Each multicast message traverses the entire tree to reach all the members. Therefore, the number of hops each multicast message traverses is  $O(n)$  and the total number of messages on all hops from  $n$  multicasts is  $O(n^2)$ .

In **Train**, each member sends a point-to-point message to the next member in line, since it takes  $O(1)$  hops to reach the next member, the total number of messages on all hops from  $n$  unicasts is  $O(n)$ .

In **Gossip**, each member unicasts a constant number of messages to a random set of members, therefore, the total number of messages in the system during each step is  $O(n)$ . The probability analysis in Section 4.2.5 shows the number of steps



(A): four basic protocols

(B): basic protocols (in detail)

Figure 5.12: Total number of messages on all hops for the four basic protocols in dense groups with 50 senders.

needed is bounded by  $O(\log(n))$ , thus the total number of messages is bounded by  $O(n \log(n))$ .

The simulation results for the four basic protocols shown in Figure 5.12 confirm the above analysis.

It also shows **Gossip** uses more messages in the system than **CoordP** and **Train**. Let  $P_{CoordP}$  and  $P_{Gossip}$  designate the number of messages used in **CoordP** and **Gossip** respectively. It is interesting to observe that the number of messages in both **Gossip** and **CoordP** are in the same order  $O(n \log(n))$ . However, Figure 5.12(A) shows approximately  $P_{Gossip} = k \times P_{CoordP}$ , with  $k > 1$ . Notice in **CoordP**, each unicast message from non-coordinators traverses at most  $\log(n)$  hops, therefore  $n \log(n)$  is a tight upper bound for  $P_{CoordP}$ . In **Gossip**, there are  $n \log(n)$  messages, assume on average each message traverses  $k$  hops be-

Table 5.2: Total number of messages on all hops for various protocols

basic protocols	<b>CoordP</b>	<b>FullDist</b>	<b>Train</b>	<b>Gossip</b>
# of messages	$O(n \log(n))$	$O(n^2)$	$O(n)$	$O(n \log(n))$
structured protocols	<b>S_CoordP</b>	<b>S_CoordP</b>	<b>S_Train</b>	<b>S_Gossip</b>
# of messages	$O(n)$	$O(n)$	$O(n)$	$O(n \log(50))$

fore arriving at its destination, the total number of hops would be  $kn \log(n)$  with  $k > 1$ . Comparing **CoordP** and **Gossip** in Figure 5.12(A), we can clearly see the factor  $k$ .

In **S\_CoordP** and **S\_Train**, each member needs to send out one message, and this message traverses a constant number of hops, thus the total number of messages is  $O(n)$ . Adding a structure does not change the fact that each member needs to send a constant number of messages in **S\_Gossip**. Gossip is conducted in local groups of size 50 in parallel, the number of steps needed is therefore  $\log(50)$ , as a result, the total number of messages becomes  $O(n \log(50))$ . The results are summarized in Table 5.2.

The simulation results for structured protocols presented in Figure 5.13 show the number of messages in **S\_Gossip** is larger than the other two, even though all of their increase is linear. This is expected from the  $\log(50)$  factor in  $O(n \log(50))$ .

The comparison between the four basic protocols and their respective structured protocols is presented in Figures 5.14 and 5.15. And we see adding a structure reduces the total number of messages needed in all the protocols.

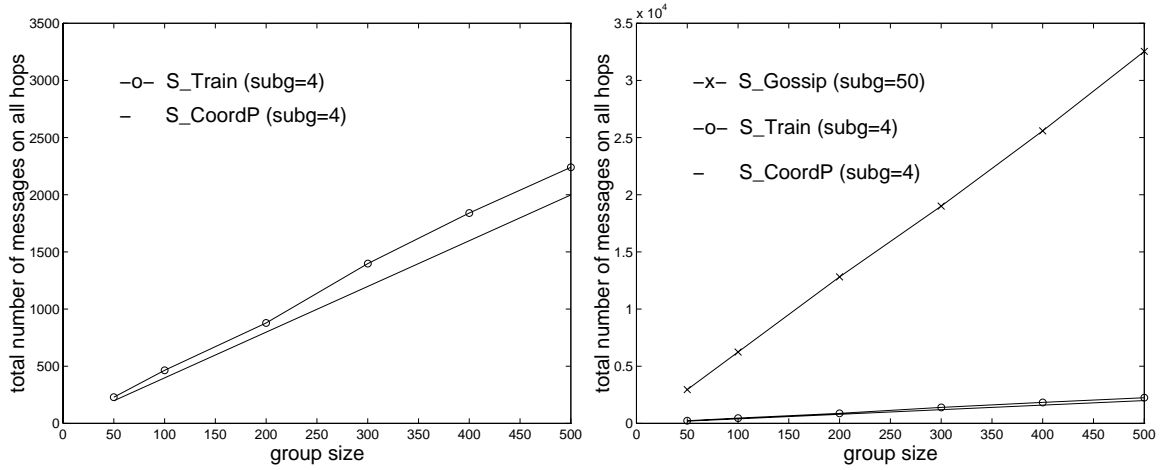
(A): without **S\_Gossip**(B): with **S\_Gossip**

Figure 5.13: Total number of messages on all hops for the three structured protocols in dense groups with 50 senders.

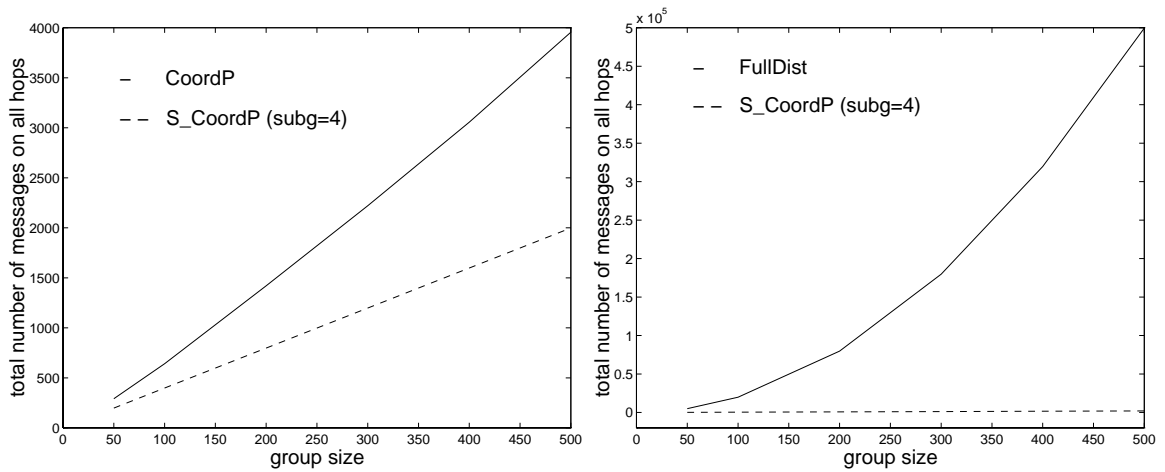
(A): **CoordP** and **S\_CoordP**(B): **FullDist** and **S\_CoordP**

Figure 5.14: Total number of messages on all hops for the basic and their corresponding structured protocols in dense groups with 50 senders (part I).

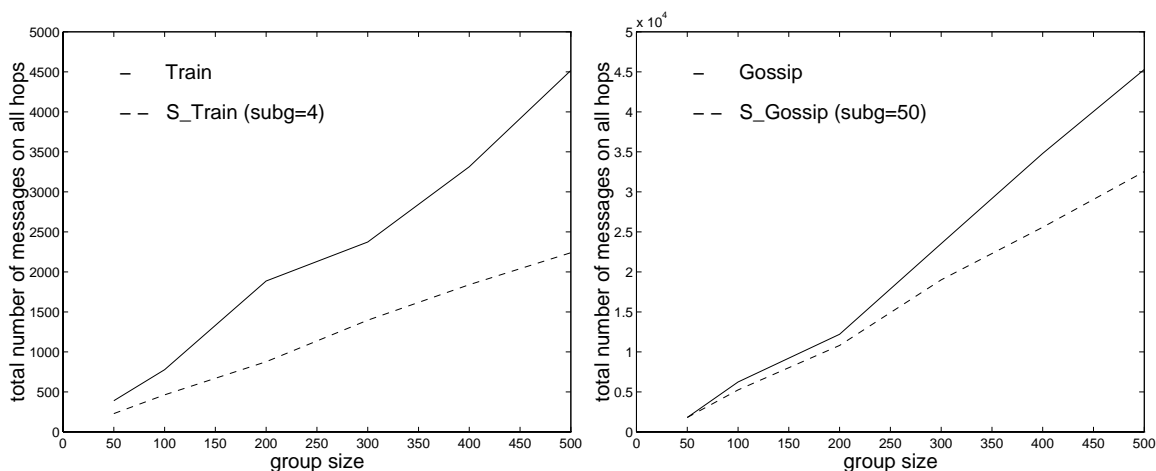
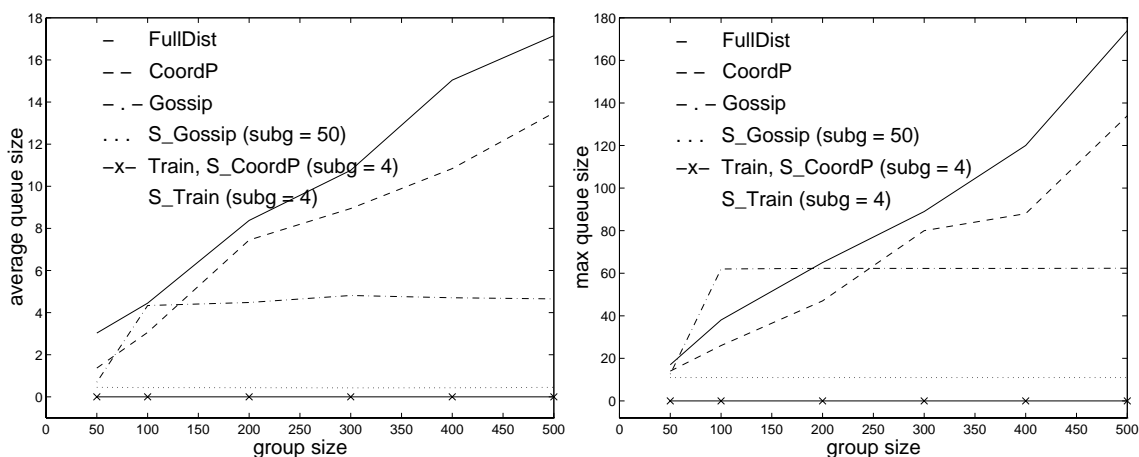
(A): **Train** and **S\_Train**(B): **Gossip** and **S\_Gossip**

Figure 5.15: Total number of messages on all hops for the basic and their corresponding structured protocols in dense groups with 50 senders (part II).

#### 5.4.2 Average and maximum queue sizes over all the nodes in the system

The average and maximum queue sizes recorded over all the network nodes during the simulation are reported in this section. These numbers are indicators of processing load at each node. They are different from the total number of messages processed by a node, since they reflect how many messages are accumulated at each node at any moment.

Figure 5.16(A) shows that out of the four basic protocols, **FullDist** has the largest value for average queue size, followed by **CoordP**, both of which are increasing with the group size, while the number for **Train** stays flat at 0. For any internal node in protocol **CoordP**, the ACK messages from all of its descendants



(A): average queue size

(B): maximum queue size

Figure 5.16: Average and maximum queue sizes over all the nodes for the basic and structured protocols in dense groups with 50 senders.

need to go through it to reach the root. Therefore the average queue size increases with  $n$ . The number for **FullDist** also increases with  $n$  simply because of the  $O(n^2)$  number of messages in the system. The number for **Train** stays at 0 since each node only needs to handle 2 messages.

The average queue size stays constant at 0 in the two structured protocols **S\_CoordP** and **S\_Train** which is the result of the number of messages handled by each node being a small number. In **S\_CoordP**, each node only needs to handle 3 ACK messages from its children. In **S\_Train**, each node needs to handle up to 5 messages.

In **Gossip** and **S\_Gossip**, the buffer size at each node is limited to 64 messages. As a result, the average queue size for **Gossip** increases to 4 when group size increases to 100 and stays at 4 with further increases of group size, whereas the

average queue size for **S\_Gossip** stays at 0.5.

The same trend is observed for maximum queue size in Figure 5.16(B).

### 5.4.3 Time-per-round (TPR)

As introduced in Section 5.2, time-per-round (TPR) is the time needed for one round of a stability detection protocol. It is defined as the duration of time between the start of a protocol and the moment the first member constructs the stability array. For **FullDist**, each member multicasts its sequence number array and each member constructs its sequence number matrix separately. We measure the time for the first and last member to detect stability. That is, we measure the smallest and the largest TPR.

#### The basic protocols

Figure 5.17 shows TPR for the four basic protocols. Among them, **Train** has the longest TPR, which is the result of the  $n$  protocol steps needed for the first member to collect message stability information. **CoordP** and **FullDist** both show increase of TPR as  $n$  increases, but since both of them have constant protocol steps, their increment is not as significant as **Train**. It is interesting to notice that the TPR for **CoordP** and the smallest TPR for **FullDist** are almost identical, whereas the largest TPR for **FullDist** is twice as large as the smallest.

#### Comparing FullDist and Gossip

Both **FullDist** and **Gossip** are robust against message losses and process crashes in the sense that in **FullDist**, each member constantly multicasts sequence number

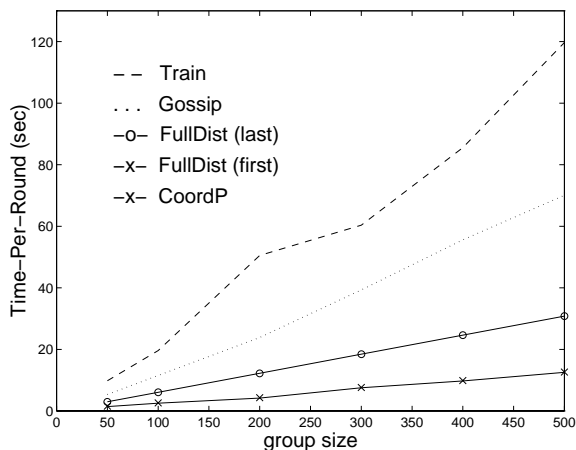


Figure 5.17: Time-per-round (TPR) for the four basic protocols in dense groups with 50 senders.

information in the entire group and in **Gossip**, each member constantly sends its information to a random subset of members. In our simulation for **FullDist**, we assume each network node has infinite buffer space and there are no message losses. **FullDist** requires the buffer size to be able to hold 180 INFO messages when the group size is 500. This is close to 42K bytes<sup>4</sup> of buffer space. As group size increases, the maximum required buffer size increases approximately linearly as presented in Figure 5.16(B). It is unreasonable to reserve buffer space large enough to process all incoming messages. Therefore, messages are dropped because of the lack of buffer space.

To avoid the implosion problem in **FullDist**, it is common practice to spread the multicast of INFO messages from each member. This technique is used in the management of session messages of SRM [FJL<sup>+</sup>96]. In the following simulation, the multicast from each member is scattered randomly and the average distance

<sup>4</sup>The size of the INFO message in the simulation is  $32 + 4 \times m = 32 + 4 \times 50 = 232$  bytes. 180 messages will occupy  $180 \times 232 = 41760$  bytes of buffer space.

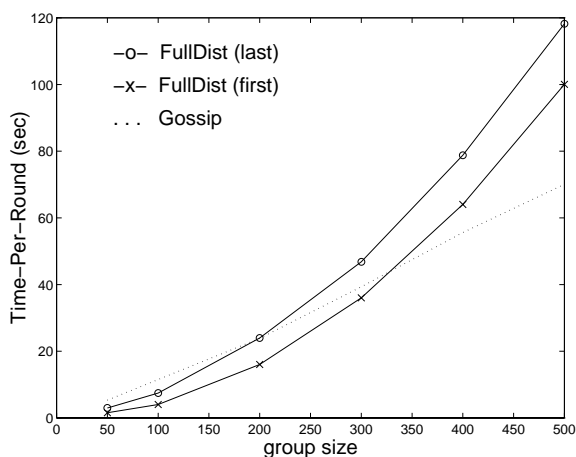


Figure 5.18: Time-per-round (TPR) for **Gossip** and **FullDist** employing the scattering mechanism in dense groups with 50 senders.

between each multicast increases linearly with the group size such that a fixed number of messages are sent out in the system per unit time. When the group size is 50, 100, 200, 300, 400 and 500, the average distance between each multicast is set to 0.02, 0.04, 0.08, 0.12, 0.16 and 0.2 seconds respectively, and on average 25 messages are sent out every 10 milliseconds. Figure 5.18 shows that when the group size is over 200, **Gossip** actually performs better than **FullDist**. The TPR for **FullDist** would be even larger if message loss is taken into consideration.

### The structured protocols

Figure 5.19 displays TPR for the three structured protocols. For both **S\_CoordP** and **S\_Train**, TPR consists of the time for ACKs to go up each level. As group size increases from 50 to 500, the height of the underlying bounded-degree tree with degree 4 only increases from 4 to 6. This is why their TPR only increases slightly as the group size increases. It takes one protocol step for the ACKs to go

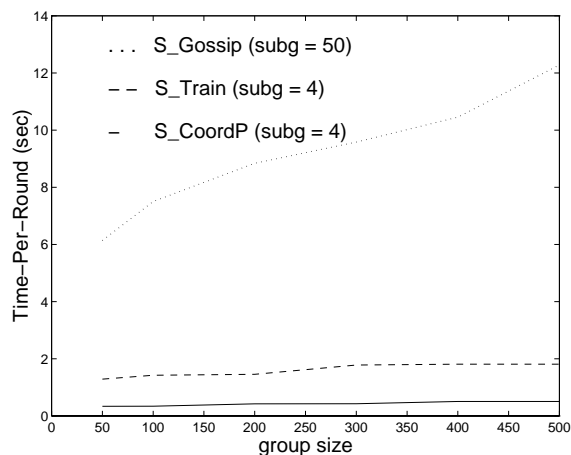


Figure 5.19: Time-per-round (TPR) for the three structured protocols in dense groups with 50 senders.

up a level in the tree in **S\_CoordP**, versus  $b = 3$  steps in **S\_Train**. This is why **S\_CoordP** has a smaller TPR than **S\_Train**. TPR for **S\_Gossip** is much larger than the other two protocols and it shows approximately linear increase with the group size. It is because only a one-level tree is used in the simulation.

### Comparing the basic and their corresponding structured protocols

The TPR results for the basic protocols and their corresponding structured protocols are displayed in Figures 5.20 and 5.21. We see that a hierarchical structure offers significant improvement in TPR to all the basic protocols.

Figure 5.20(A) shows the TPR for **CoordP** and **S\_CoordP**. There are 3 steps in **CoordP**, but  $p + 2$  steps in **S\_CoordP**. Contrary to what the protocol steps suggest, we find that the TPR for **CoordP** increases dramatically as  $n$  increases, while the TPR for **S\_CoordP** stays almost flat. In **CoordP**, for any node in the tree, ACK messages from all its descendants need to travel through the node in

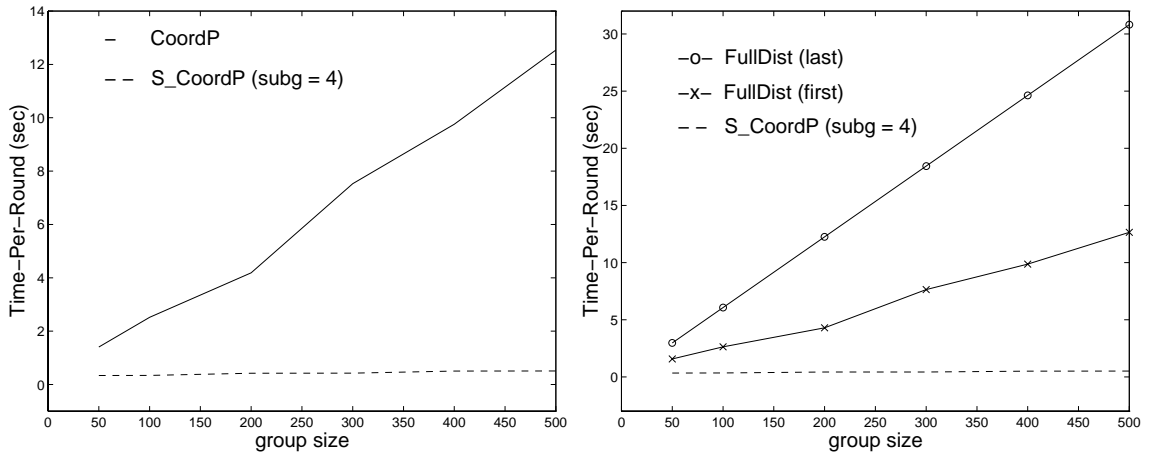
(A): **CoordP** and **S\_CoordP**(B): **FullDist** and **S\_CoordP**

Figure 5.20: Time-per-round (TPR) for the basic and their corresponding structured protocols in dense groups with 50 senders (part I).

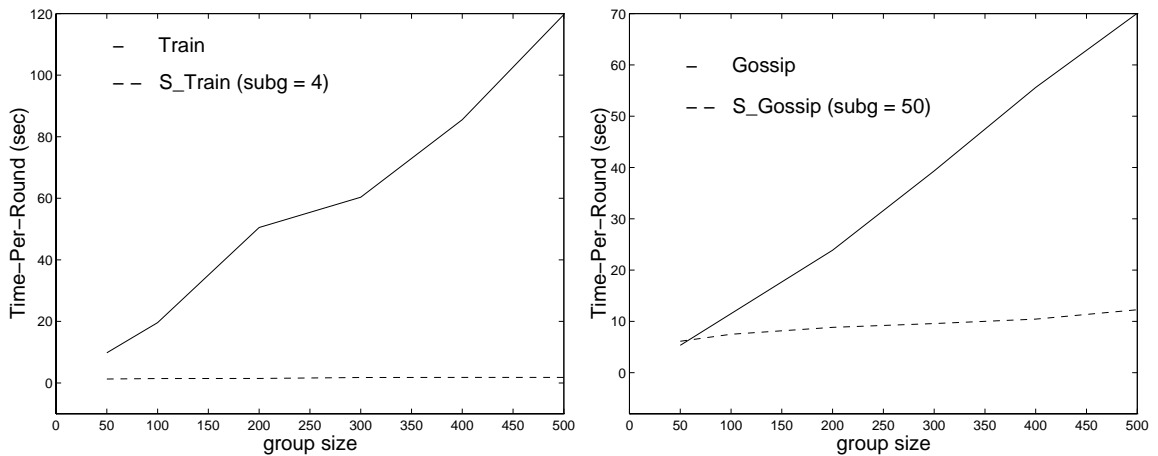
(A): **Train** and **S\_Train**(B): **Gossip** and **S\_Gossip**

Figure 5.21: Time-per-round (TPR) for the basic and their corresponding structured protocols in dense groups with 50 senders (part II).

order to reach the root. The closer a node is to the root, the more ACK messages it needs to handle. This scheme will overload the root node, its neighbors and the links connecting them when  $n$  becomes large. As a result, ACK implosion will appear at the root and those nodes close to it. On the other hand, in **S\_CoordP**, each ACK message travels only one hop to its parent. There is only one ACK message on any hop in the tree network. Each node only needs to handle up to  $b$  ACK messages, and, when the branching factor of the tree  $b$  is small (3 in our simulation), this will not cause an implosion problem at any node. The TPR for **S\_CoordP** is essentially the time spent for the START message to travel  $p$  hops down the tree plus the time for the ACKs to travel  $p$  hops up the tree to reach the root. Since there is no contention for the links, this time is proportional to the tree height  $p$ . As the tree height varies from 4 to 6 when the group size ranges from 50 to 500 in our simulation, we see an almost flat line for the TPR of **S\_CoordP**. When  $n = 500$ , **S\_CoordP** offers a 25-time improvement in TPR over **CoordP**.

Figure 5.20(B) shows TPR for **FullDist** and its corresponding structured protocol **S\_CoordP**. We see a sharp increase in TPR for **FullDist** because of implosion at every member when  $n$  is large. TPR for **S\_CoordP** stays flat because the reduced number of messages each member needs to handle eliminates the implosion problem.

The results for **Train** and its structured version **S\_Train** are presented in Figure 5.21(A). There is no message implosion problem in either protocol. The TPR is proportional to the number of steps in the protocol. **Train** takes  $n$  steps, whereas **S\_Train** takes  $bp + 2$  steps, where the branching factor of the tree  $b = 3$  and the height of the tree  $p$  varies from 4 to 6 in the simulation. This explains why

**S\_Train** scales significantly better than **Train**. When  $n = 500$ , **S\_Train** offers over 66-fold improvement in TPR over **Train**.

Even though only a one-level tree hierarchy is used in **S\_Gossip**, its TPR shows almost 6-fold improvement over **Gossip** when  $n = 500$  as shown in Figure 5.21(B).

## 5.5 Comparison of Various Protocols in Sparse Groups with 50 Senders

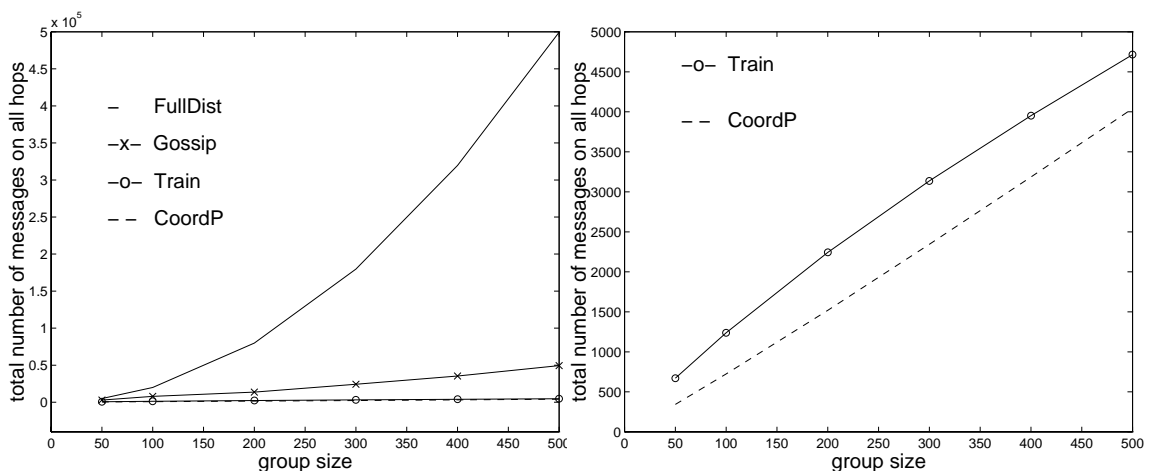
In the sparse test, a balanced bounded-degree tree of size 1000 is built.  $n$  nodes are randomly chosen to be in the multicast group, and the remaining  $1000 - n$  are routers.

The same set of simulations are conducted in the sparse test as in the dense test. In the sparse test, we choose a more realistic approach where the logical tree in the structured protocols does not match the underlying network topology. To see the effect of the branching factor of the logical tree on the performance of the structured protocols, we conducted the following two tests.

In the first test, we build logical trees with a branching factor of 3. This means that the size of each sub-group is 4. In the second test, we build a two-level tree where at the bottom level, each sub-group contains 50 members, consisting of 49 siblings and one parent. Each local group reports to its parent at the middle level, and the middle level members report to the root. In this case, the branching factor at the bottom level is fixed to 49 and varies at the middle level.

All the logical trees in the sparse test are constructed randomly without respect to the underlying network topology. We still can observe the similar trends as in the dense test.

For each group size, we built 20 random sparse groups in the 1000-node tree and conducted simulation for all the protocols for each of the sparse groups. The results presented in this section are the average of the 20 tests.



(A): four basic protocols

(B): basic protocols (in detail)

Figure 5.22: Total number of messages on all hops for the basic protocols in sparse groups with 50 senders.

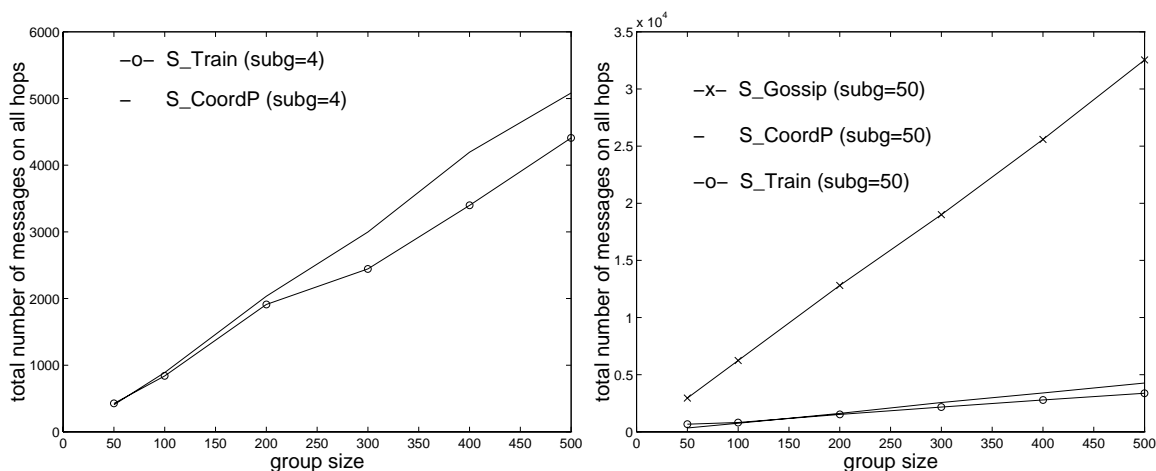
### 5.5.1 Total number of messages on all hops in the system

The total number of messages in the setting of sparse groups follow the same trend as in the dense groups as summarized in Table 5.2.

The simulation results for the basic protocols are presented in Figure 5.22. Their behavior is similar to the dense group case.

When the sub-group size is 4, **S\_CoordP** uses more messages than **S\_Train** as shown in Figure 5.23(A). This is not surprising, because the construction of the logical tree with degree 4 does not match the underlying network tree, therefore on average, a message sent from an interior node of the tree to its parent traverses slightly more hops than messages to its neighboring sibling.

When the sub-group size is 50 in the logical tree, on the other hand, a message



(A): sub-group size = 4

(B): sub-group size = 50

Figure 5.23: Total number of messages on all hops for the three structured protocols in sparse groups with 50 senders.

from **S\_Train** traverses slightly more hops than in **S\_CoordP** as displayed in Figure 5.23(B).

Figures 5.24 and 5.25 display the total number of messages for each basic protocol and its corresponding structured protocol with different sub-group sizes 4 and 50.

The smaller the sub-group size, the larger the height of the logical tree, and fewer hops a message need to traverse going up one level of the tree, thus fewer messages are used. As a result, we are expecting that the basic protocol uses more messages than its structured protocol with sub-group size 50 which in turn uses more messages than that with sub-group size 4.

It is interesting to notice in Figure 5.24(A), **S\_CoordP** with either group size actually uses more messages than the basic **CoordP**! Again, this is because the

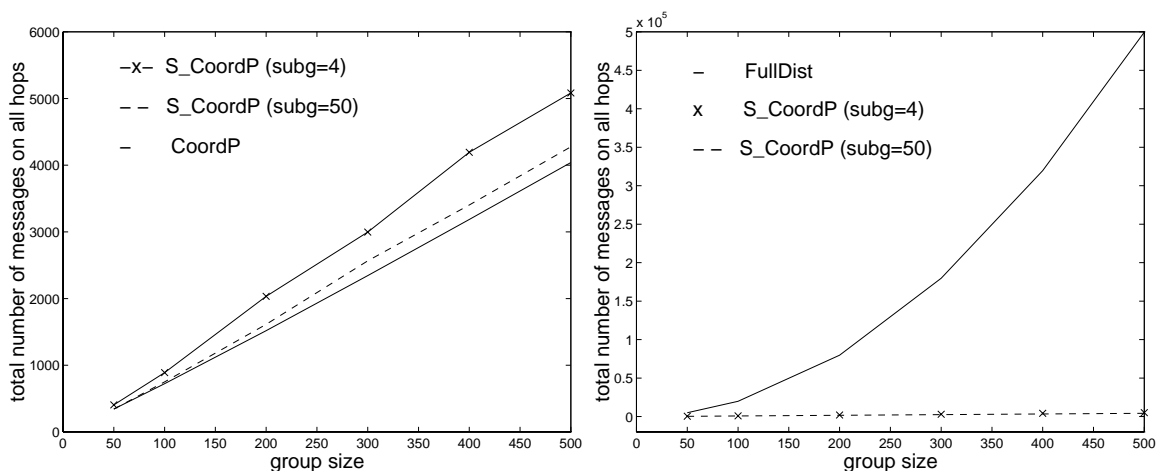
(A): **CoordP** and **S\_CoordP**(B): **FullDist** and **S\_CoordP**

Figure 5.24: Total number of messages on all hops for the basic and their corresponding structured protocols in sparse groups with 50 senders (part I).

logical tree in **S\_CoordP** does not match the underlying network tree.

### 5.5.2 Average and maximum queue sizes over all the nodes in the system

The average queue size shows the same trend in sparse groups as in dense groups. Figure 5.26 shows that out of the four basic protocols, **FullDist** has the largest value for average queue size, followed by **CoordP**, both of which are increasing with the group size, while the number for **Train** stays flat at 0.

Figure 5.27(A) shows that when the sub-group size is 4, the average queue size increases with the group size for both **S\_CoordP** and **S\_Train**.

In **S\_CoordP** and **S\_Train**, the number of messages handled by each node is

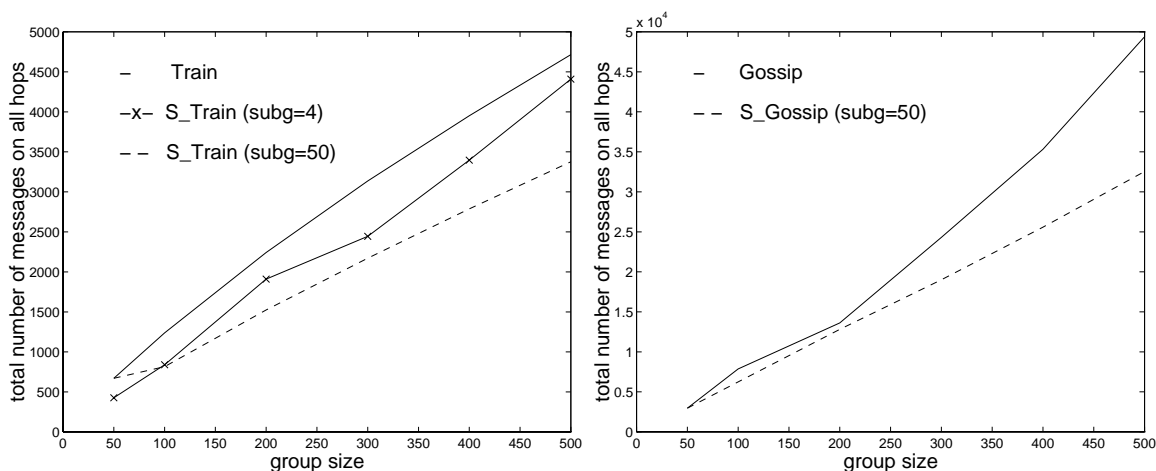
(A): **Train** and **S\_Train**(B): **Gossip** and **S\_Gossip**

Figure 5.25: Total number of messages on all hops for the basic and their corresponding structured protocols in sparse groups with 50 senders (part II).

a small number: in **S\_CoordP**, each node only needs to handle 3 ACK messages from its children, and in **S\_Train**, each node needs to handle up to 5 messages. But because the logical tree does not match the physical network tree, each message needs to travel more than one hop before arriving at its destination. This causes more congestion at the routers as the group size increases, which results in an increase of the average queue size.

Figure 5.27(B) shows that when the sub-group size is 50, the average queue size increases significantly with the group size for **S\_CoordP**, but increases very slowly for **S\_Train**. This is because the larger sub-group size requires the interior nodes in **S\_CoordP** to handle 49 ACK messages, while the nodes in **S\_Train** still only need to handle up to 5 messages.

In **Gossip** and **S\_Gossip**, the buffer size at each node is limited to 64 messages.

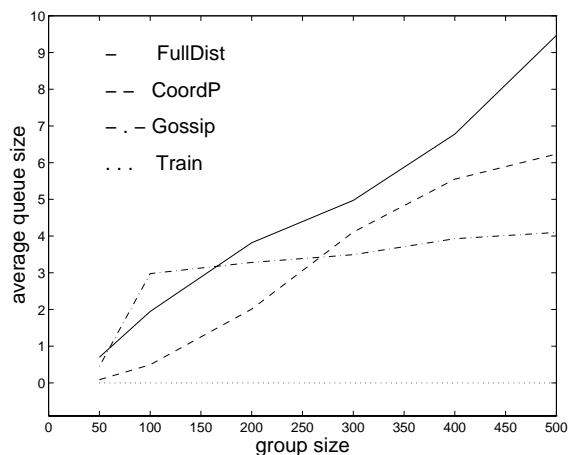


Figure 5.26: Average queue size over all the nodes for the four basic protocols in sparse groups with 50 senders.

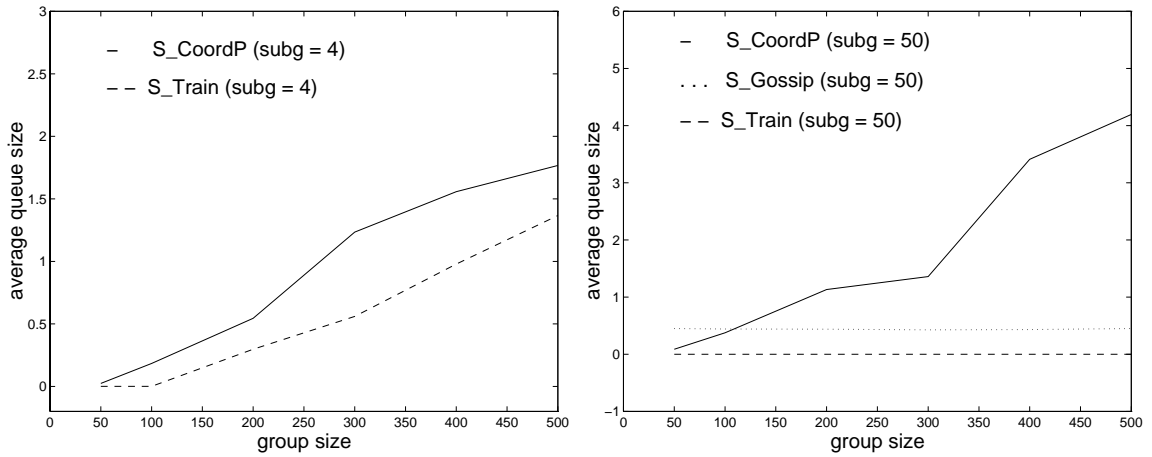
As a result, the average queue size for both **Gossip** and **S\_Gossip** only increases slowly.

The same trend is observed for maximum queue size in Figures 5.28 to 5.29.

### 5.5.3 Time-per-round (TPR)

#### The basic protocols

From Figure 5.30, we can see that TPR results for the four basic protocols exhibit the same trend in the sparse group test as in the dense group test in Section 5.4.3. For all the basic protocols, the TPR increases as the group size increases. Out of these four, **Train** has the largest TPR, followed by **Gossip**, **FullDist** and **CoordP**.



(A): sub-group size = 4

(B): sub-group size = 50

Figure 5.27: Average queue size over all the nodes for the structured protocols in sparse groups with 50 senders.

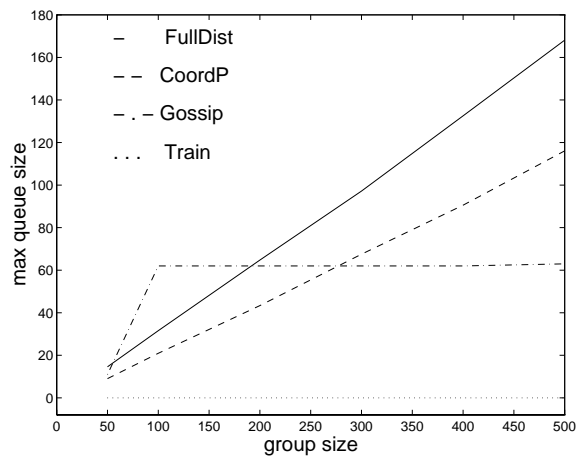
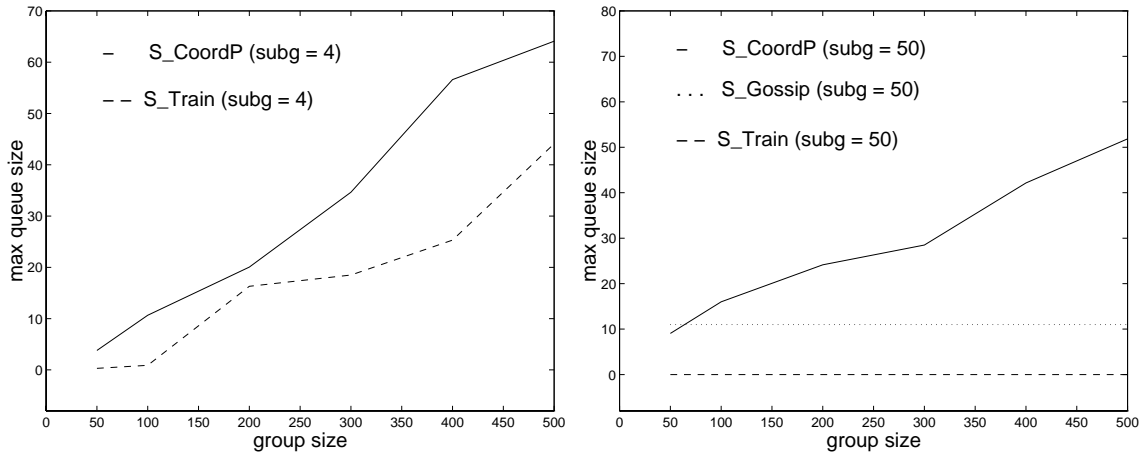


Figure 5.28: Maximum queue size over all the nodes for the four basic protocols in sparse groups with 50 senders.



(A): sub-group size = 4

(B): sub-group size = 50

Figure 5.29: Maximum queue size over all the nodes for the structured protocols in sparse groups with 50 senders.

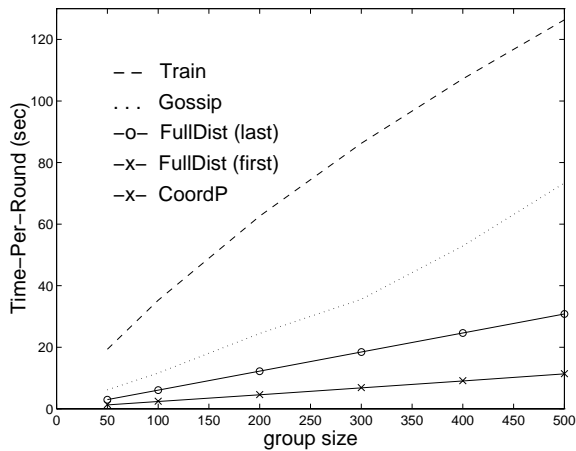
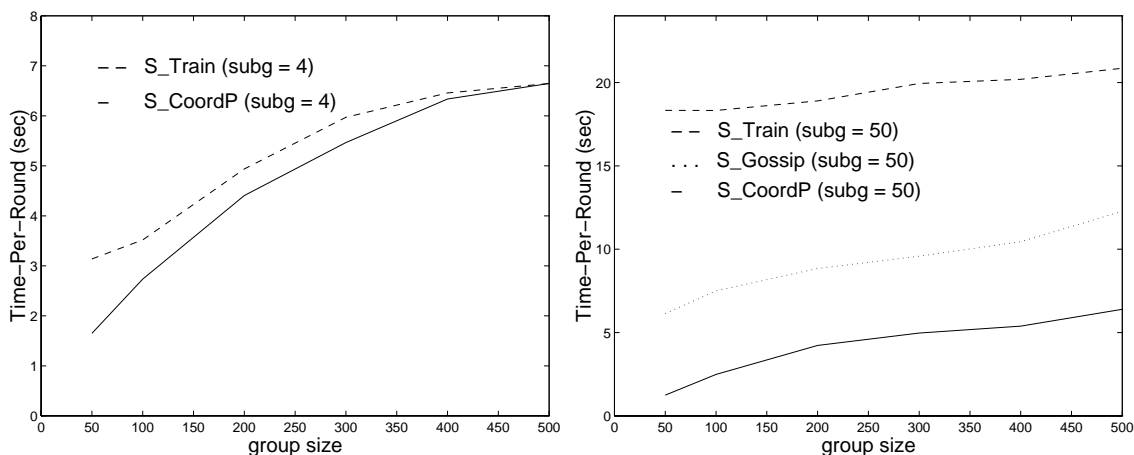


Figure 5.30: Time-per-round (TPR) for the four basic protocols in sparse groups with 50 senders.



(A): sub-group size = 4

(B): sub-group size = 50

Figure 5.31: Time-per-round (TPR) for the structured protocols in sparse groups with 50 senders.

### The structured protocols

Figure 5.31(A) displays the TPR results for the two structured protocols **S\_Train** and **S\_CoordP** with sub-group size 4.

For both protocols, TPR consists of the time for ACKs to go up each level. As group size increases from 50 to 500, the height of the underlying bounded-degree tree with degree 4 only increases from 4 to 6. Because the logical trees do not match the physical network tree, the time it takes to go up one level in the tree increases as the group size increases. This results in the increase of TPR with the group size. It takes one protocol step for the ACKs to go up a level in the tree in **S\_CoordP**, versus  $b = 3$  steps in **S\_Train**. This is why **S\_CoordP** has a slightly smaller TPR than **S\_Train**.

Figure 5.31(B) displays the TPR results for the three structured protocols

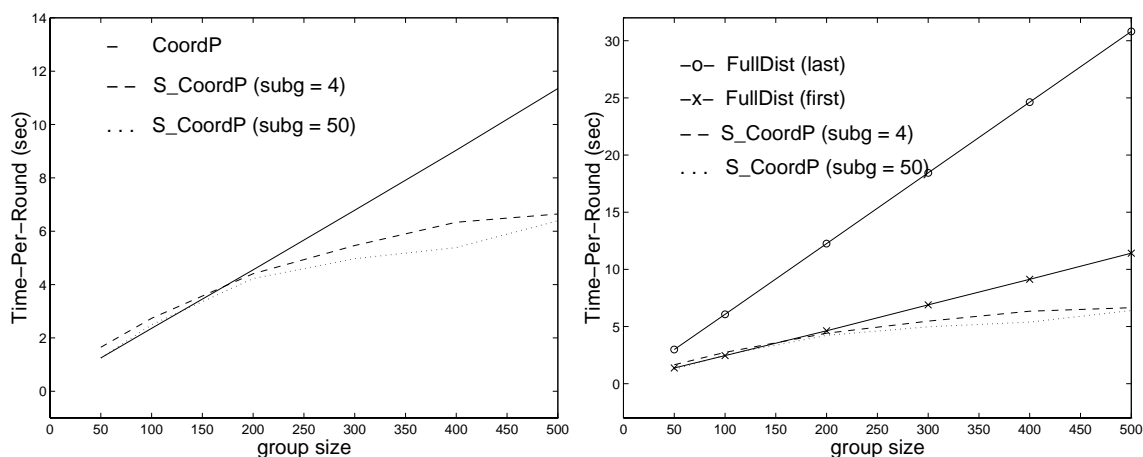
(A): **CoordP** and **S\_CoordP**(B): **FullDist** and **S\_CoordP**

Figure 5.32: Time-per-round (TPR) for the basic and their corresponding structured protocols in sparse groups with 50 senders (part I).

**S\_Train**, **S\_CoordP**, and **S\_Gossip** with sub-group size 50. In this case, the logical tree has only one level. It takes one protocol step for the ACKs to go up a level in the tree in **S\_CoordP**, versus  $b = 49$  steps in **S\_Train**. This is why **S\_Train** has a significantly larger TPR than **S\_CoordP**. TPR for **S\_Gossip** is slightly larger than **S\_CoordP**, but it is a lot smaller than **S\_Train**.

### Comparing the basic and their corresponding structured protocols

The TPR results for the basic protocols and their corresponding structured protocols are displayed in Figures 5.32 and 5.33. We see that a hierarchical structure offers significant improvement in TPR to all the basic protocols.

Figure 5.32(A) shows the TPR for **CoordP** and **S\_CoordP** with two different sub-group sizes: 4 and 50. Over all the group sizes, **S\_CoordP** with sub-group size

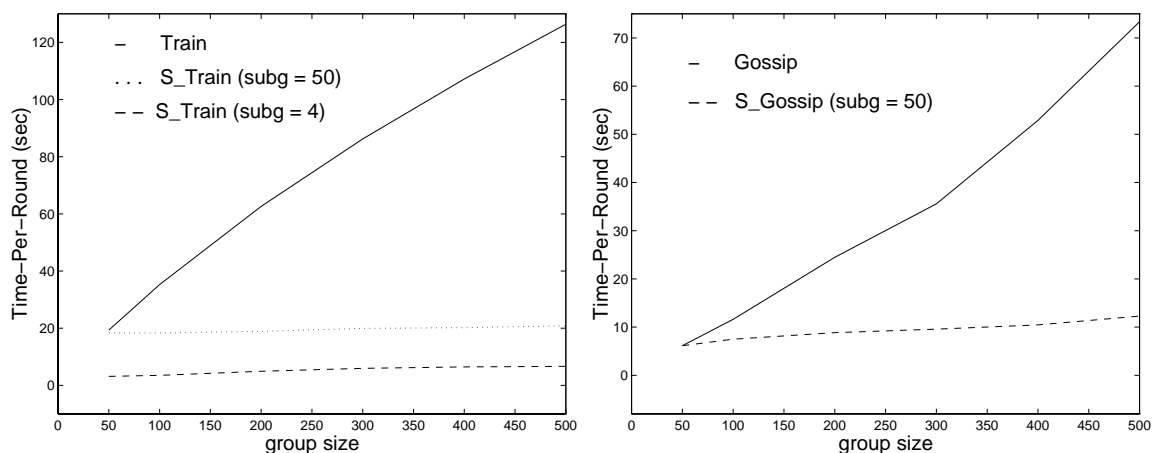
(A): **Train** and **S\_Train**(B): **Gossip** and **S\_Gossip**

Figure 5.33: Time-per-round (TPR) for the basic and their corresponding structured protocols in sparse groups with 50 senders (part II).

50 exhibits a smaller TPR than the same protocol with sub-group size 4. When the sub-group size is 50, the height of the logical tree is 2, it takes 2 protocol steps for ACK messages from all the members to arrive at the root of the logical tree. When the sub-group size is 4, the height of the logical tree ranges from 4 to 6, so it takes 4 to 6 protocol steps for ACK messages from all the members to arrive at the root. Apparently from the simulation results, the effect of the ACK implosion problem is not significant in the case for sub-group size of 50. The saving of protocol steps makes 50 a better sub-group size than 4 for **S\_CoordP**. The recommended sub-group size for **S\_CoordP** when the logical tree does not match the physical network tree is as large as possible as long as the ACK implosion problem does not occur.

When the group size is less than 150, **CoordP** even performs better than

**S\_CoordP** with either sub-group sizes. Again this observation shows that there is a trade-off between the sub-group size and the height of the logical tree. A large sub-group size might trigger the ACK implosion problem which will increase TPR whereas a small tree height decreases the number of protocol steps which will in turn decrease TPR. For a fixed group size, the larger the sub-group size, the smaller the tree height will be. Our suggested principle is to use a sub-group size as large as possible in order to reduce the tree height.

Figure 5.32(B) shows TPR for **FullDist** and its corresponding structured protocol **S\_CoordP** with sub-group sizes being 4 and 50. We see a sharp increase in TPR for **FullDist** because of implosion at every member when  $n$  is large. TPR for **S\_CoordP** only increases slightly because the reduced number of messages each member needs to handle eliminates the implosion problem.

The results for **Train** and its structured version **S\_Train** are presented in Figure 5.33(A). There is no message implosion problem in any of the protocols. The TPR is proportional to the number of steps in the protocol. **Train** takes  $n$  steps with  $n$  ranging from 50 to 500. **S\_Train** with sub-group size 4 takes  $bp + 2$  steps, where the branching factor of the tree  $b = 3$  and the height of the tree  $p$  varies from 4 to 6, as a result the number of steps ranges from 14 to 20. **S\_Train** with sub-group size 50 takes 49 steps to go up the bottom level of the logical tree, and  $n/50$  steps to get information from all the sub-groups. Therefore the total number of steps is  $49 + n/50$  which ranges from 50 to 59.

This explains why **S\_Train** scales significantly better than **Train**. And between the two versions of **S\_Train**, we recommend the one with a smaller sub-group size because the number of steps needed is smaller therefore the TPR is smaller.

The results for **Gossip** and its structured version **S\_Gossip** are presented in Figure 5.33(B). Even though only a one-level tree hierarchy is used in **S\_Gossip**, its TPR shows almost 6-fold improvement over **Gossip** when  $n = 500$ .

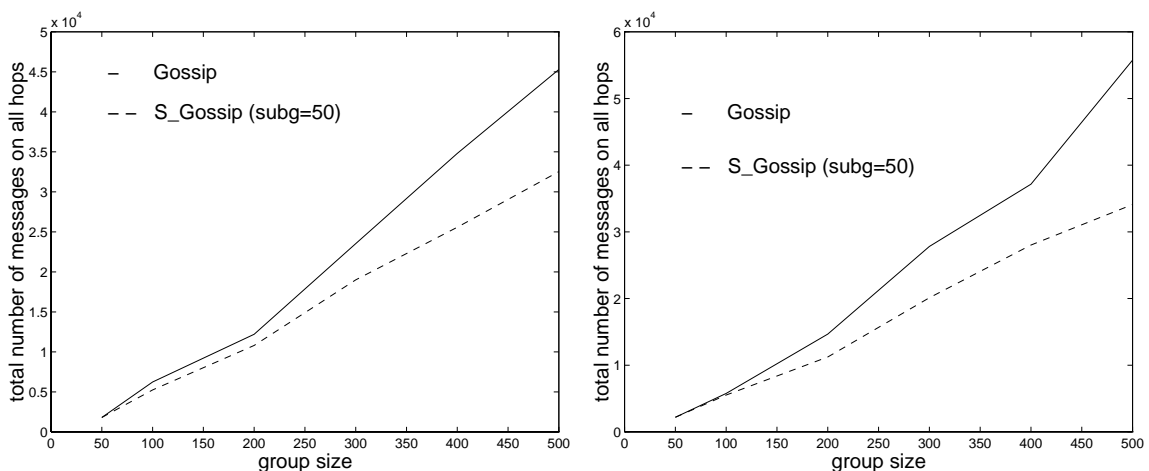
## 5.6 Comparison of Various Protocols in Dense Groups with One Sender

In the following two sections, we present the simulation results when there is only one sender. The protocols are still the same except the message sizes are reduced by  $4 \times 49 = 196$  bytes.

### 5.6.1 Total number of messages on all hops in the system

As mentioned in Section 5.3.3, we conducted simulations for the **Gossip** protocol using the optimal step interval for each group size. When there is only one sender, because the message size is smaller, the step interval is smaller as shown in Table 5.1 and Figure 5.11. Comparing Figures 5.10(B) and 5.11(B), we also notice that more steps are needed to detect stability when there is only one sender. Since each member sends out one random gossip message during each step interval, and on average each message traverses the same number of hops, the total number of messages on all hops is slightly larger when there is only one sender. This observation is true for both **Gossip** and **S\_Gossip** as shown in Figure 5.34.

For the rest of the protocols, the number of messages sent out in the system and the number of hops each message traverses are independent of the message size. Therefore the total number of messages is identical for different numbers



(A): with 50 senders

(B): with one sender

Figure 5.34: Total number of messages on all hops for **Gossip** and **S\_Gossip** with different number of senders in dense groups.

of senders. The simulation results for the rest of the protocols are presented in Figures C.1 to C.4 of Appendix C.

### 5.6.2 Average and maximum queue sizes over all the nodes in the system

The average and maximum queue sizes recorded over all the network nodes during the simulation are reported in this section. These numbers reflect how many messages are accumulated at each node at any moment. We present the average queue size for different numbers of senders in Figure 5.35. We can observe the average queue size is slightly larger for **Gossip**, **S\_Gossip**, **CoordP** and **FullDist** when there is one sender.

For **Gossip** and **S\_Gossip**, the step interval is smaller when the number of senders  $m = 1$ , and the number of messages sent out in the system during each step is the same:  $n$ . Therefore, there are more messages sent out per unit time. Recall in Section 5.1, we set the router processing time for a message to be 1 millisecond independent of message size. Routers still process messages with the same speed, but there are more messages in the system per unit time. The natural consequence of this is that more messages are accumulated at the routers, resulting in a larger average queue size.

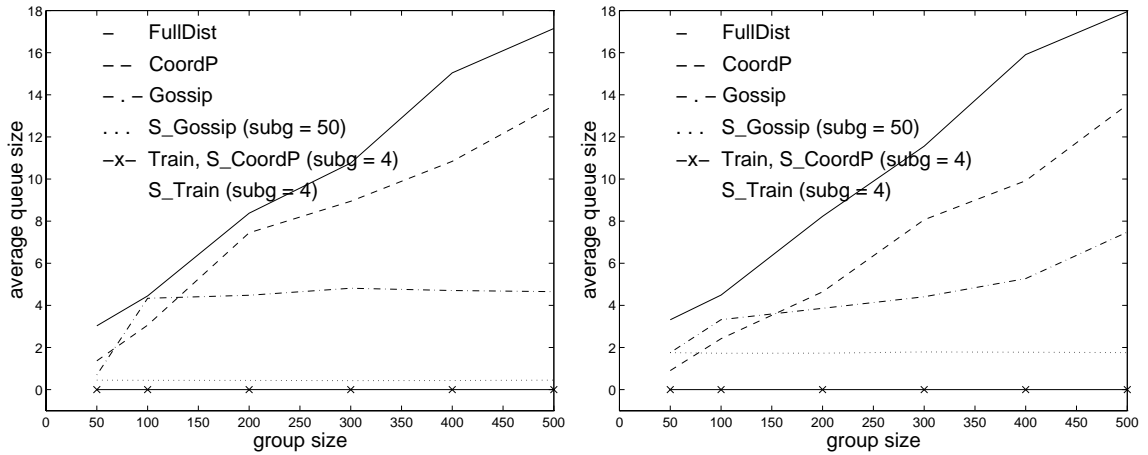
For **CoordP** and **FullDist**, the number of messages sent out in the system is identical for different numbers of senders. When  $m = 1$ , the size of the ACK message is smaller, therefore a message traverses a link in a shorter amount of time. The router processing time is the same independent of message size. During the 1 millisecond it takes a router to process a message, more messages arrive at the router because of the shorter time period messages spent on the links. This causes more messages to accumulate at the routers waiting for processing.

For **Train**, **S\_Train**, and **S\_CoordP**, the average queue size is 0 for either number of senders because of the small number of messages each node needs to handle.

The maximum queue size shows the same trend as the average queue size and is displayed in Figure 5.36.

### 5.6.3 Time-per-round (TPR)

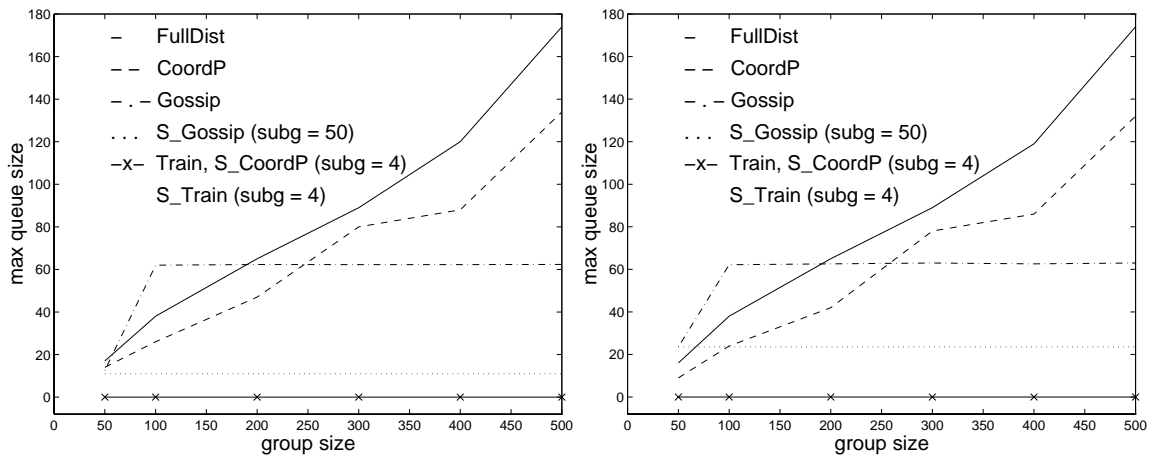
Compared with 50 senders, when there is only one sender, for **Gossip** and **S\_Gossip** the step interval is smaller, the gossip message size is smaller, a message traverses



(A): with 50 senders

(B): with one sender

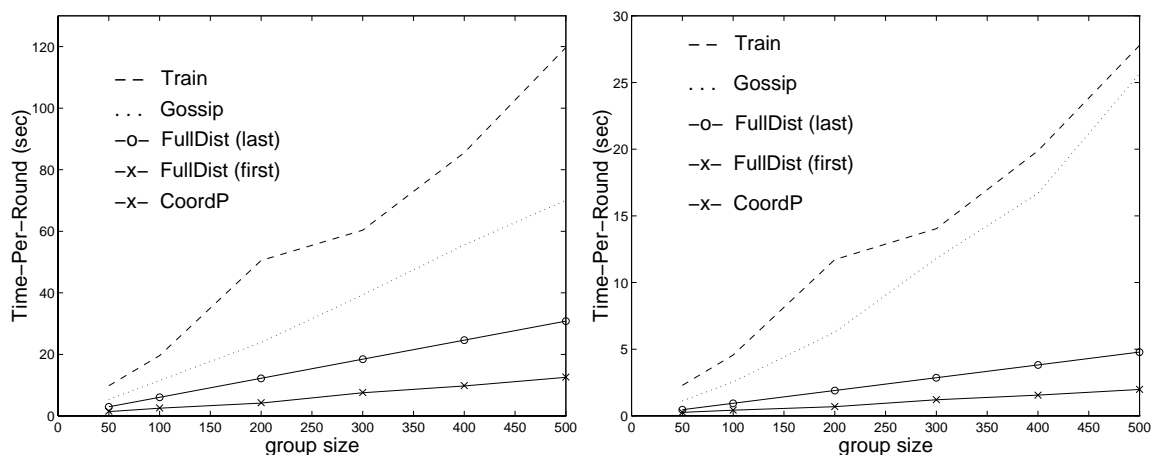
Figure 5.35: Average queue size over all the nodes with different numbers of senders for the basic and structured protocols in dense groups.



(A): with 50 senders

(B): with one sender

Figure 5.36: Maximum queue size over all the nodes with different numbers of senders for the basic and structured protocols in dense groups.



(A): with 50 senders

(B): with one sender

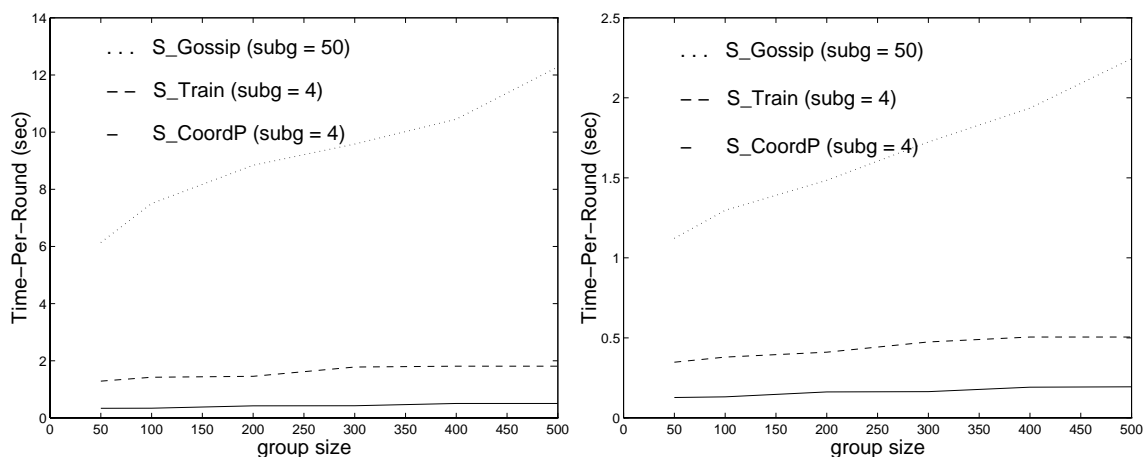
Figure 5.37: Time-per-round (TPR) with different numbers of senders for the four basic protocols in dense groups.

a link in a shorter amount of time, and a message is processed by a node faster. All these factors lead to the result that TPR is significantly smaller when the number of senders  $m = 1$ .

For the rest of the protocols, the message size is smaller, a message traverses a link faster and is processed by a node faster. Therefore, the resulting TPR is a lot smaller when  $m = 1$  than when  $m = 50$ .

The TPR results for the four basic protocols are shown in Figure 5.37, and the results for the three structured protocols are shown in Figure 5.38.

The TPR results for the basic and their corresponding structured protocols are presented in Figures C.7 and C.8 of Appendix C.



(A): with 50 senders

(B): with one sender

Figure 5.38: Time-per-round (TPR) with different numbers of senders for the three structured protocols in dense groups.

## 5.7 Comparison of Various Protocols in Sparse Groups with One Sender

The simulation results for sparse groups with the number of senders  $m = 1$  follow the same trend as the results for dense groups with  $m = 1$ .

Compared with results for sparse groups with  $m = 50$ , the  $m = 1$  case exhibits the same characteristic as presented in Section 5.6. The total number of messages on all hops in the system is slightly larger when  $m = 1$  for **Gossip** and **S\_Gossip**, and remains the same for the rest protocols. The maximum and average queue sizes are slightly larger when  $m = 1$  for all the protocols. And the TPR is significantly smaller when  $m = 1$  for all the protocols.

The detailed results are presented in Figure D.1 to D.12 of Appendix D.

## 5.8 Summary

This chapter compares different stability detection protocols under a set of WAN environments using simulation. A set of performance metrics are used to evaluate the protocols.

There are two input parameters for the **Gossip** protocol — the subset size and the step interval. Simulation results show that for each subset size, there is a smallest acceptable step interval such that any step interval smaller than this limit would cause significant increase of message burstiness which in turn will damage the performance of the **Gossip** protocol. Simulation results also show that for any subset size, there is a window of step intervals in which one can select any step interval to achieve a near-minimum TPR. An algorithm is presented to find this optimal range of step intervals.

Under different simulation tests, there are trade-offs between different protocols, nevertheless we can draw certain conclusions for protocols under different conditions.

The basic protocols can be listed in the following order according to decreasing number of messages on all hops in the system: **FullDist**, **Gossip**, **Train** and **CoordP**. They can be listed in the decreasing order of maximum and average queue sizes as follows: **FullDist**, **CoordP**, **Gossip** and **Train**. The queue size in **Gossip** is smaller than **CoordP**, because in all the simulations, the maximum queue size is limited to 64 messages for the **Gossip** protocol. According to TPR, these protocols can be listed in the following decreasing order: **Train**, **Gossip**, **FullDist** and **CoordP**.

The structured protocols always perform better than their corresponding basic

protocols, as they use less number of messages and cause less congestion on the network.

When the multicast group size is small, there is no danger of the message implosion problem, the desired stability detection protocols are **CoordP**, **FullDist** and **Train**. The reason is that these protocols are deterministic, in the sense that after a predetermined number of steps, they are guaranteed to detect message stability. On the other hand, when the group size is large, the recommended protocols are **Gossip** and the structured protocols, because they effectively eliminate the implosion problem.

# Chapter 6

## Stability Triggering Mechanism

After studying the various stability detection algorithms in detail in the previous chapters, we now look at the triggering problem for these protocols. The triggering mechanism addresses the question of how often the protocols should be executed in order to release stable messages from buffers before buffer overflow happens.

Assume that all members are sending messages at the same rate  $r$  messages per second, and that each member has an output buffer of size  $g$  messages and an input buffer of size  $ng$  messages. Notice that when each member is sending at the same rate, during the time each sender sends  $g$  messages, it could at most receive  $ng$  messages from all the  $n$  members. The corresponding time-per-round (TPR) (in seconds) for the stability detection protocol is expressed as  $f = f(r)$ , a function of data message rate  $r$  messages per second. This is a naive assumption. But we do know that the time for one round of the stability detection protocol to complete is influenced by traffic load generated by data messages. We can say that  $f(r)$  is a complicated function of which  $r$  is an important input variable.

We designate one node in the group to trigger the stability detection proto-

col. The triggering mechanism works as follows. When the output buffer at the triggering node reaches  $X$  messages, the stability detection protocol is triggered. During the time period  $f(r)$  of a TPR, the number of data messages sent out will be  $Q = rf(r)$ . Then the number of messages in the triggering node's output buffer will be  $X + Q$ , where  $X$  is the number of existing messages before the trigger, and  $Q$  is the number of newly sent messages since the trigger.

Suppose  $\alpha X$  ( $0 \leq \alpha \leq 1$ ) messages are stable at the end of a round of the stability detection protocol. In other words,  $1 - \alpha$  is the failure ratio which stands for the percentage of data messages lost<sup>1</sup>. Then  $\alpha X$  out of the  $X$  messages can be released from the top of the output buffer. If  $(1 - \alpha)X + Q < X$ , then the triggering node waits until the number of messages in its output buffer reaches  $X$ . If  $(1 - \alpha)X + Q \geq X$ , then the triggering node starts the next round of the stability detection protocol. When the data messages fill up the buffer, all the senders stop sending data messages.

In this triggering mechanism, the message number  $X$  in the triggering node's output buffer should be chosen, given the measured TPR  $f(r)$  for different protocols. If  $X$  is too large, multicast group members will constantly be forced to stop sending data messages since their buffers are full. If  $X$  is too small, the stability detection protocol will be triggered too often and the total number of messages in the system will increase unnecessarily. Therefore an optimal value of  $X$  should be determined according to the following set of constraints enforced on  $X$  by the buffer size:

- The first round of stability detection protocol starts when there are  $X$  mes-

---

<sup>1</sup>We do not assume that the FIFO property is provided by the underlying protocols here. When FIFO is assumed,  $\alpha = 1$ , and it is a special case in the following discussion.

sages in the buffer, therefore,

$$X \leq g \tag{6.1}$$

is a necessary condition.

- At the end of the first round, there are  $X + rf$  messages in the buffer, so

$$X + rf \leq g \tag{6.2}$$

must be satisfied. This makes Equation 6.1 redundant.

- After the first round,  $\alpha X$  messages can be released from the sender's output buffer. If  $(1 - \alpha)X + rf < X$ , then the sender waits until  $X$  is filled to start the next round, so  $X + rf \leq g$  is required. If  $(1 - \alpha)X + rf \geq X$ , the second round is trigger immediately,  $rf$  messages are newly sent during the second round, in order to prevent buffer overflow,

$$(1 - \alpha)X + rf + rf = (1 - \alpha)X + 2rf \leq g \tag{6.3}$$

is needed.

Assume each member sends  $K$  messages in the test, and the sending rate is  $r$  when the output buffer is not full. The optimization problem can be described as follows. Given output buffer size  $g$ , message sending rate  $r$ , and TPR  $f = f(r)$ , choose  $X$  to minimize the total number rounds of the stability detection protocol  $T(X) = \frac{K}{\alpha X}$ , or, equivalently, to maximize  $X$ , subject to the following two constraints:

$$X + rf \leq g \tag{i}$$

$$(1 - \alpha)X + 2rf \leq g \quad (\text{ii})$$

(i) and (ii) can be simplified to

$$X \leq g - rf \quad (\text{iii})$$

$$X \leq \frac{g - 2rf}{1 - \alpha} \quad (\text{iv})$$

which result in the optimal  $X = \min(g - rf, \frac{g-2rf}{1-\alpha})$ . When given some special values of  $\alpha$ , the optimal  $X$  becomes the following:

- When  $\alpha = 0$ ,  $X = \min(g - rf, g - 2rf) = g - 2rf$ .
- When  $\alpha = 0.5$ ,  $X = \min(g - rf, 2(g - 2rf))$ .
- When  $\alpha = 1$ ,  $2rf \leq g$  is satisfied by the assumption, hence  $X = g - rf$ .

In general,

- When  $\alpha = 0$ , the optimal  $X = g - 2rf$ .
- When  $\alpha = 1$ , the optimal  $X = g - rf$ .
- When  $0 < \alpha < 1$ , the optimal  $X = \min(g - rf, \frac{g-2rf}{1-\alpha})$ .

Given the total number of messages  $K$  sent by each member, failure rate  $(1 - \alpha)$  and the optimal  $X$ , the minimum number of tests needed is  $T(X) = \frac{K}{\alpha X}$ . If a smaller time-to-stable (TTS) is needed,  $X$  has to be reduced, which results in an increase of  $T(X)$ .

In order to gain more intuition about  $X$ , the optimal number of messages in the output buffer when a round of the stability detection protocol should be triggered, we examine the behavior of  $X$  for a given set of parameters. Most messages on the Internet are around 500 bytes. For simplicity, we assume each data message has

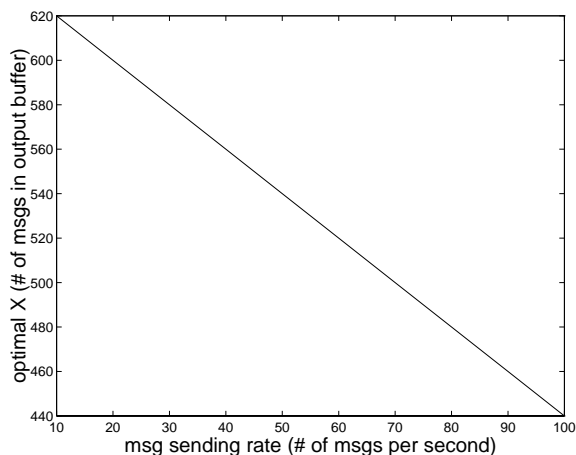


Figure 6.1: Optimal number of messages in the output buffer when the stability detection protocol should be triggered ( $\alpha = 1$ ).

a fixed size of 500 bytes. Assume the output buffer size for each group member is 320K bytes, which can store  $g = 320 \times 10^3 / 500 = 640$  data messages. Assume the TPR for the stability detection protocol is  $f = 2$  seconds. We also assume the message sending rate varies from 10 to 100 messages per second.

When  $\alpha = 1$ , the optimal value of  $X$  is plotted in Figure 6.1. When  $\alpha = 0.2$ ,  $X$  is plotted in Figure 6.2.

In a real network environment, messages might get lost in the network because of buffer overflow at some intermediate nodes, and later get retransmitted. This and other non-deterministic factors will add more delay to the TPR and TTS. Therefore, to use the triggering algorithm efficiently in practice, users should measure the  $f(r)$  in the current system, then set  $X$ ,  $Q$ , and  $g$  accordingly. Our simulation results presented in Chapter 5 are useful because they provide a lower bound in TPR  $f(0)$  under the given tree shaped network structure. They also provide some insight on how to choose stability detection algorithms under different

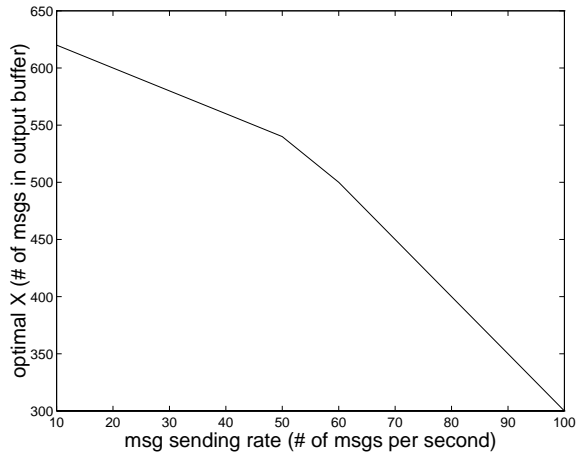


Figure 6.2: Optimal number of messages in the output buffer when the stability detection protocol should be triggered ( $\alpha = 0.2$ ).

circumstances.

## 6.1 Summary

After studying the various stability detection algorithms in detail in the previous chapters, this chapter studies the triggering problem for these protocols. The triggering mechanism addresses the question of how often the protocols should be executed in order to release stable messages from buffers before buffer overflow happens.

This triggering mechanism is formulated as an optimization problem and the solution is derived accordingly.

# Chapter 7

## Discussion and Conclusions

Message stability detection is an integral part of garbage collection in reliable multicast protocols. In addition, it can support atomic message ordering as mentioned in Section 2.3. For example, in a stock trading application, to be fair to all the traders, a message is not delivered until it is received by all the group members. Message stability detection protocols also play an important role in distributed data base management systems, and parallel computing systems.

The basic stability detection protocols have their limitations in scalability. By employing a tree structure in the basic protocols, we have derived three structured protocols with significant increase of scalability.

There are some techniques commonly used to improve the performance of stability detection protocols. Messages used for stability detection can be piggybacked on data messages to reduce traffic load in the network and the time processors spend handling messages received; only the changed sequence numbers need to be exchanged among group members in order to reduce the message size; some senders can be grouped together sharing one series of sequence numbers to reduce

the size of the sequence number array (or stability matrix) kept at each member, therefore the size of ACK and INFO messages. In the extreme case, when the stability detection protocol is combined with a total ordering protocol, only 4 bytes of information is needed from each member, and the message size is reduced to a minimum. These techniques can be applied to both the basic and the structured protocols to further improve their performance.

Message stability detection protocols are designed to discover the largest sequence number from each sender such that each message with a smaller sequence number has been received by all the group members. All the protocols follow the same general procedure:

- Each member keeps a sequence number array in which it stores the largest sequence number from each sender such that all messages with smaller sequence numbers have been received.
- A stability array is the element-wise minimum of the sequence number arrays from all the members. Each protocol uses different mechanisms to collect this stability array and to distribute it in the group.

In reliable multicast protocols, there are three ways for message receivers to detect missing messages.

1. Receivers detect gaps in sequence number space.
2. The sender periodically multicasts the sequence number of the last message sent out in a burst, because when the last message is lost, it is not detectable using the first method.

3. Receivers exchange information about the largest sequence number they have seen from each sender.

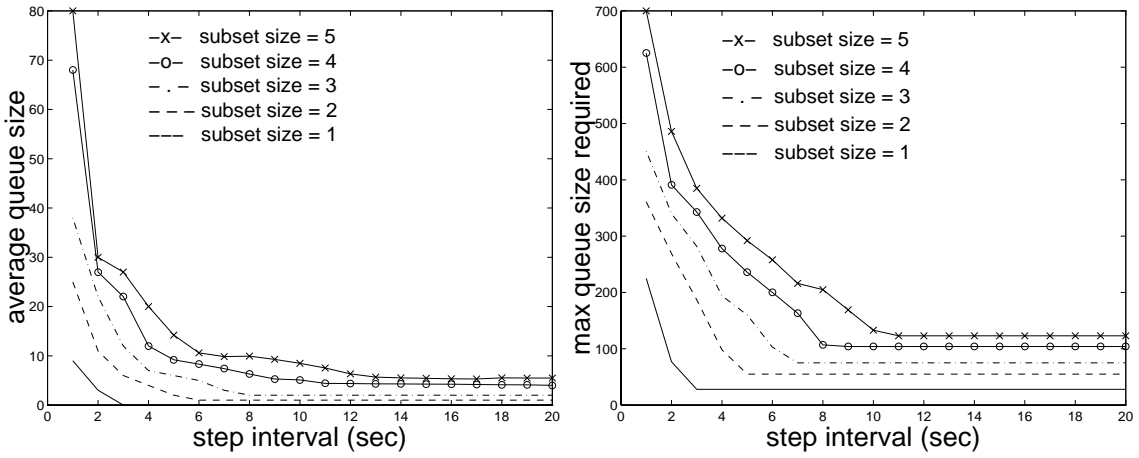
With a slight change in the definition of sequence number array, the stability detection protocols become error detection protocols. The general error detection mechanism works as follows:

- Each member keeps a high-end sequence number array in which it stores the largest sequence number from each sender.
- An error detection array is the element-wise *maximum* of the high-end sequence number arrays from all the members. Each protocol uses different mechanism to collect this error detection array and distributed it in the group.

We have presented simulations of the set of representative stability detection protocols in a WAN environment with restricted bandwidth. Different network characteristics will affect the behavior of these protocols, in terms of TPR and queue sizes recorded over the nodes in the network. However, we believe that the analysis of scalability of different protocol categories is valid in other types of network environment as well.

# Appendix A

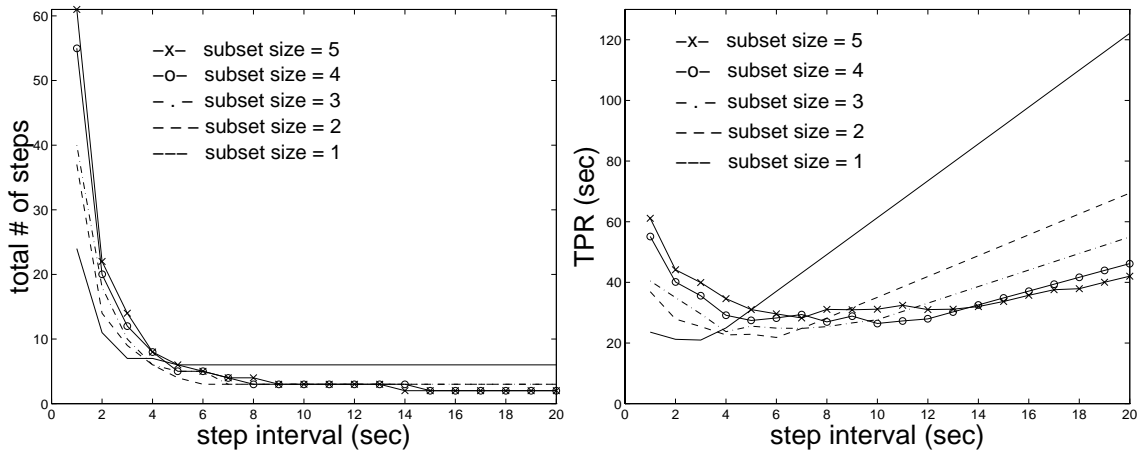
## Simulation Results for Gossip in Dense Groups



(A): average queue size

(B): maximum queue size

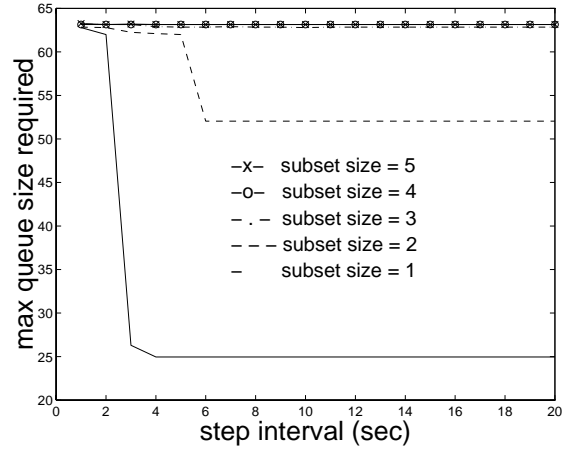
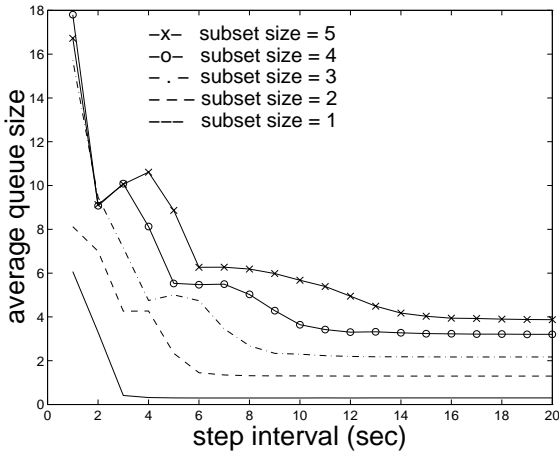
Figure A.1: Simulation I (no message loss) with a dense group of size  $n = 200$  (part I).



(A): number of steps in a round

(B): time-per-round (TPR)

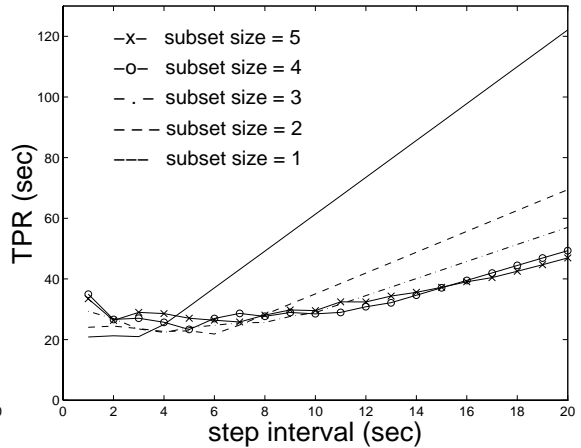
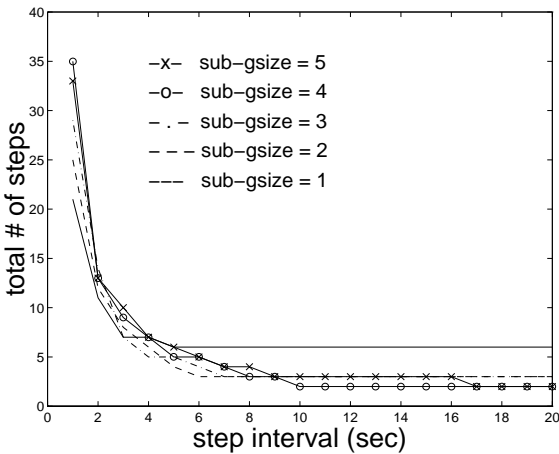
Figure A.2: Simulation I (no message loss) with a dense group of size  $n = 200$  (part II).



(A): average queue size

(B): maximum queue size

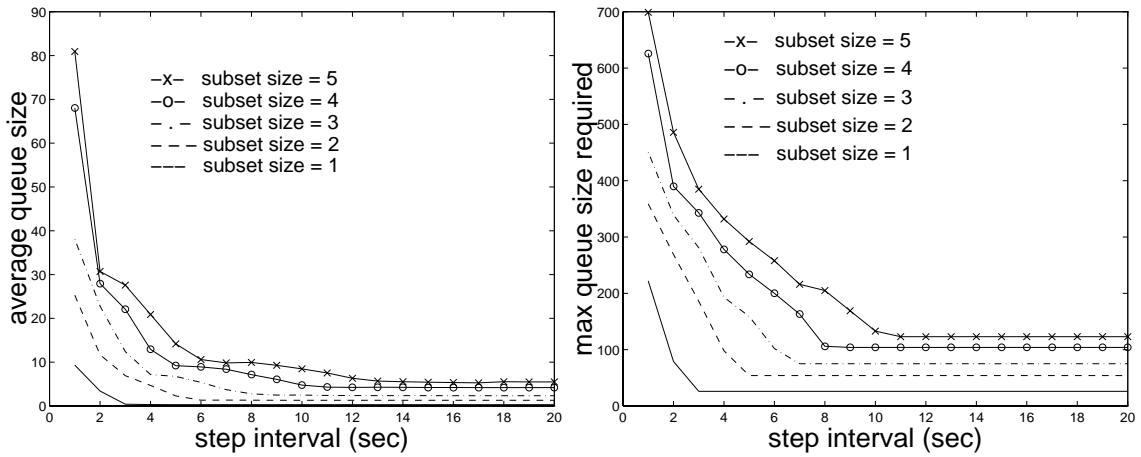
Figure A.3: Simulation II (queue size = 64 and 2 lost messages per step) with a dense group of size  $n = 200$  (part I).



(A): number of steps in a round

(B): time-per-round (TPR)

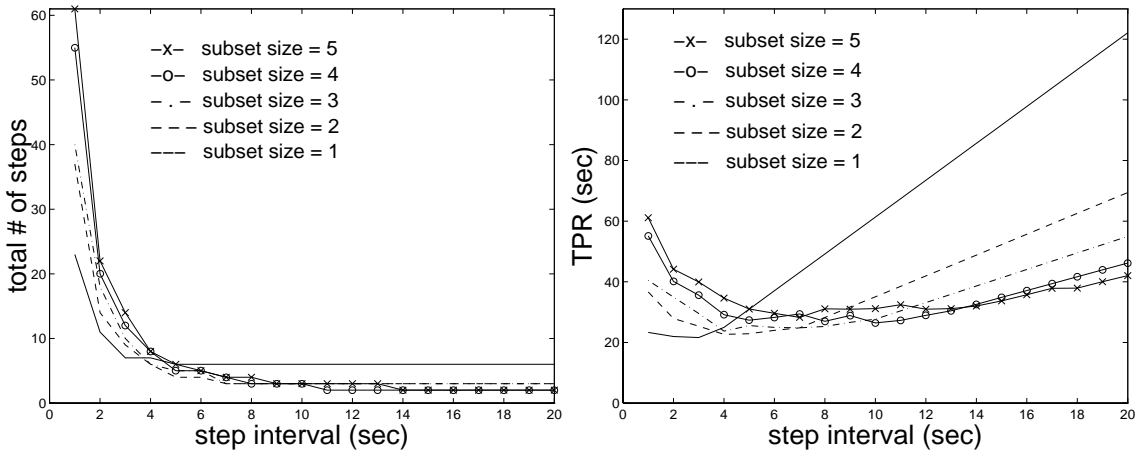
Figure A.4: Simulation II (queue size = 64 and 2 lost messages per step) with a dense group of size  $n = 200$  (part II).



(A): average queue size

(B): maximum queue size

Figure A.5: Simulation III (2 lost messages per step) with a dense group of size  $n = 200$  (part I).



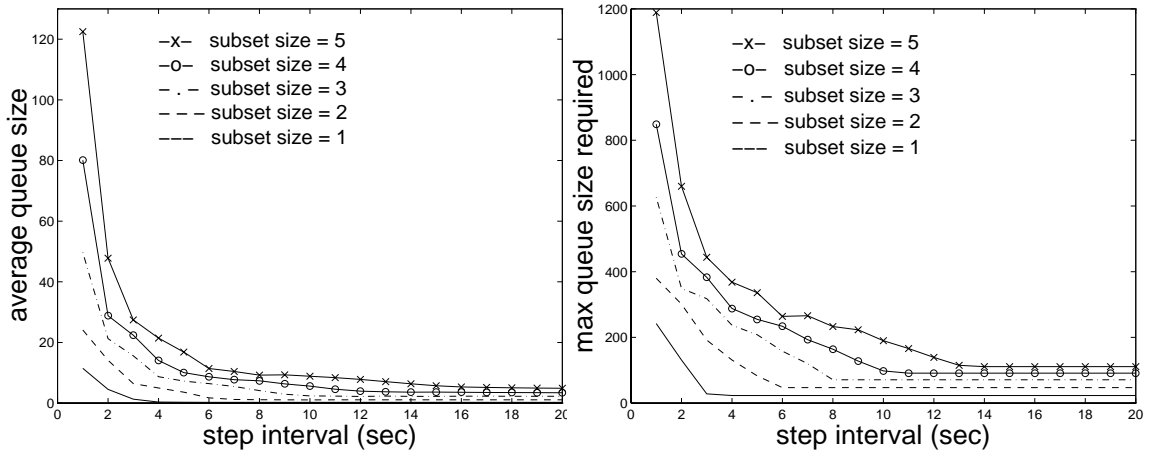
(A): number of steps in a round

(B): time-per-round (TPR)

Figure A.6: Simulation III (2 lost messages per step) with a dense group of size  $n = 200$  (part II).

## **Appendix B**

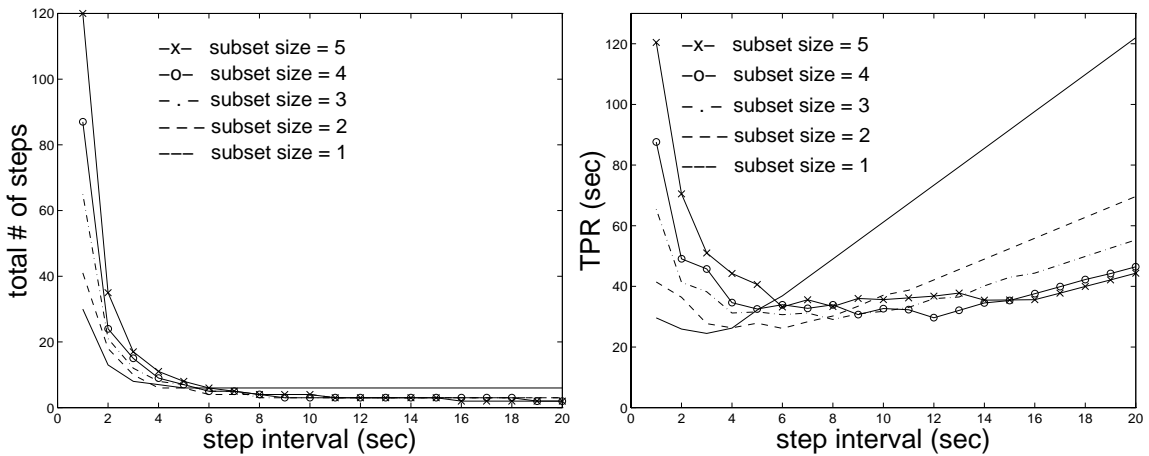
# **Simulation Results for Gossip in Sparse Groups**



(A): average queue size

(B): maximum queue size

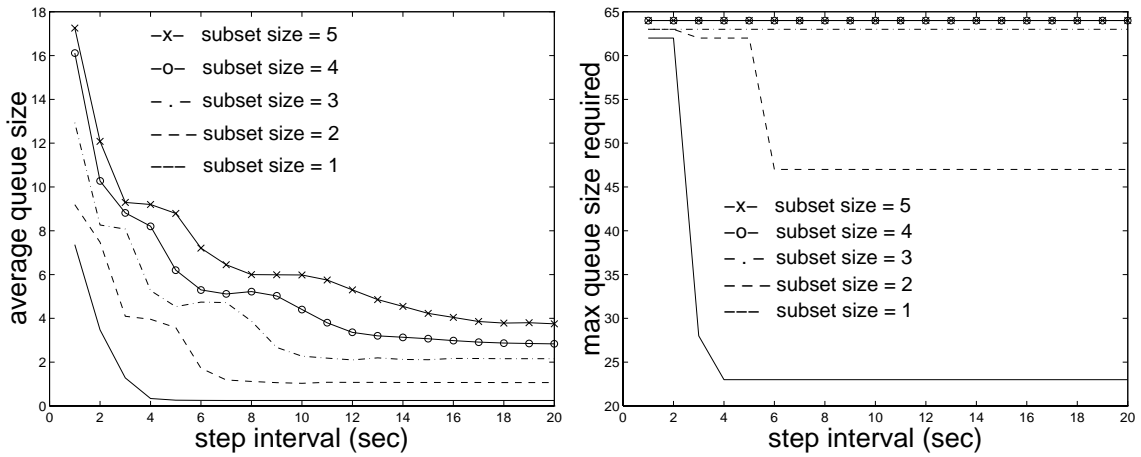
Figure B.1: Simulation I (no message loss) with 20 sparse groups of size  $n = 200$  (part I).



(A): number of steps in a round

(B): time-per-round (TPR)

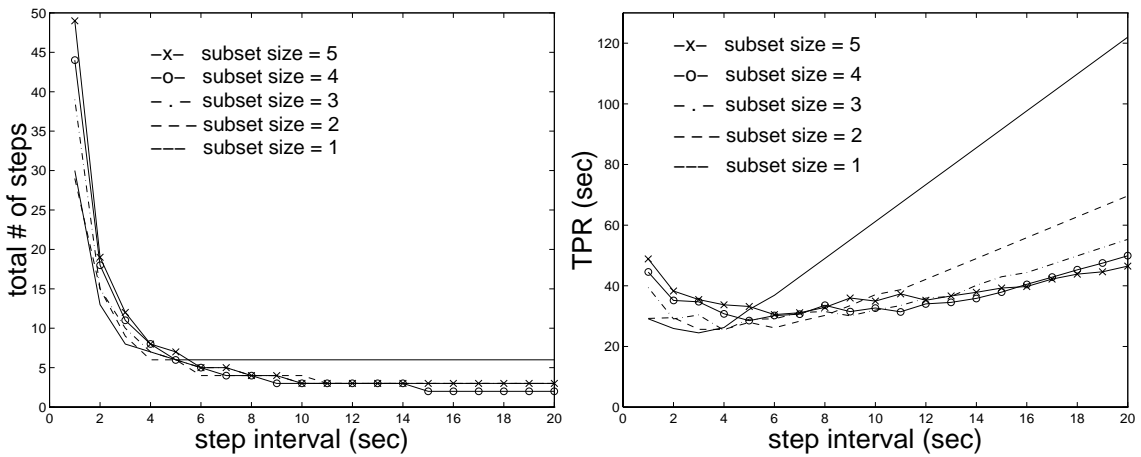
Figure B.2: Simulation I (no message loss) with 20 sparse groups of size  $n = 200$  (part II).



(A): average queue size

(B): maximum queue size

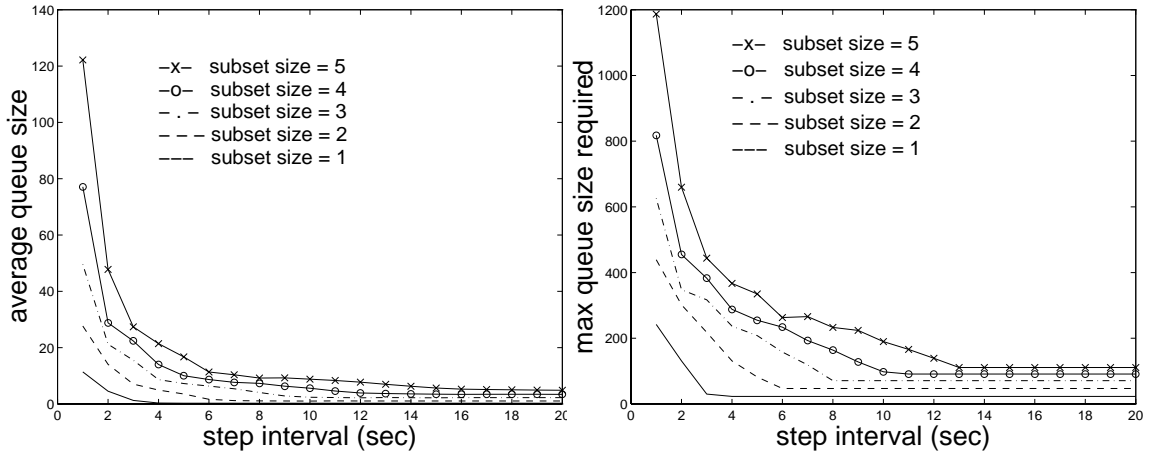
Figure B.3: Simulation II (queue size = 64 and 2 lost messages per step) with 20 sparse groups of size  $n = 200$  (part I).



(A): number of steps in a round

(B): time-per-round (TPR)

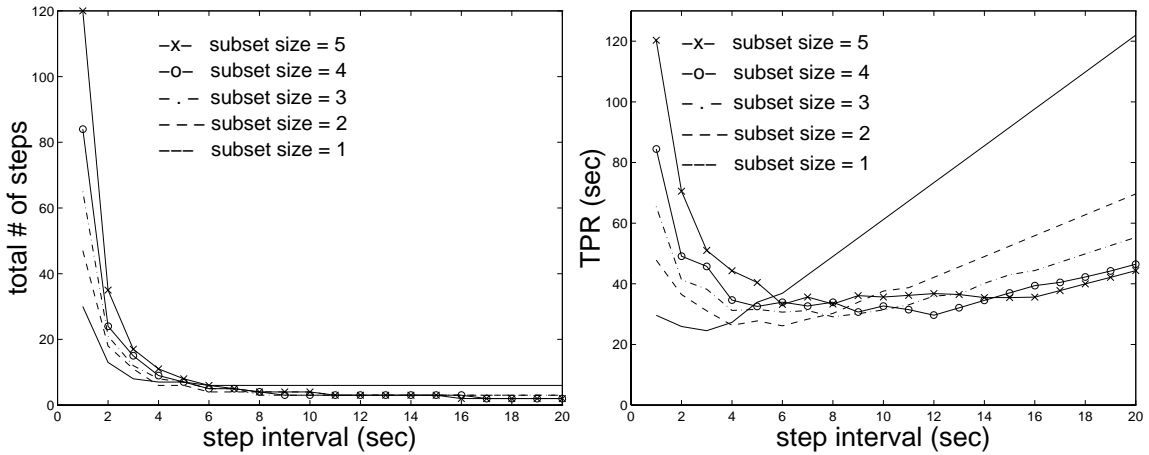
Figure B.4: Simulation II (queue size = 64 and 2 lost messages per step) with 20 sparse groups of size  $n = 200$  (part II).



(A): average queue size

(B): maximum queue size

Figure B.5: Simulation III (2 lost messages per step) with 20 sparse groups of size  $n = 200$  (part I).



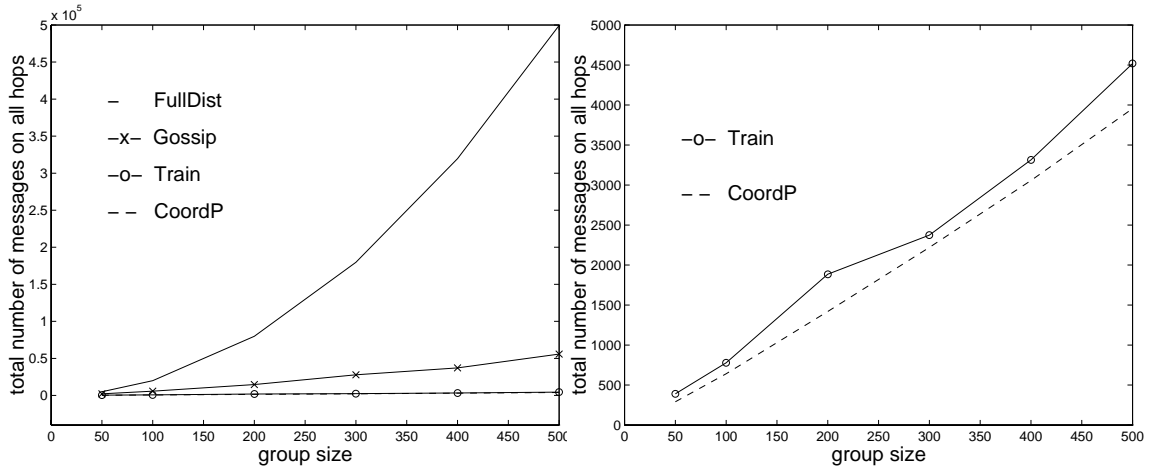
(A): number of steps in a round

(B): time-per-round (TPR)

Figure B.6: Simulation III (2 lost messages per step) with 20 sparse groups of size  $n = 200$  (part II).

# Appendix C

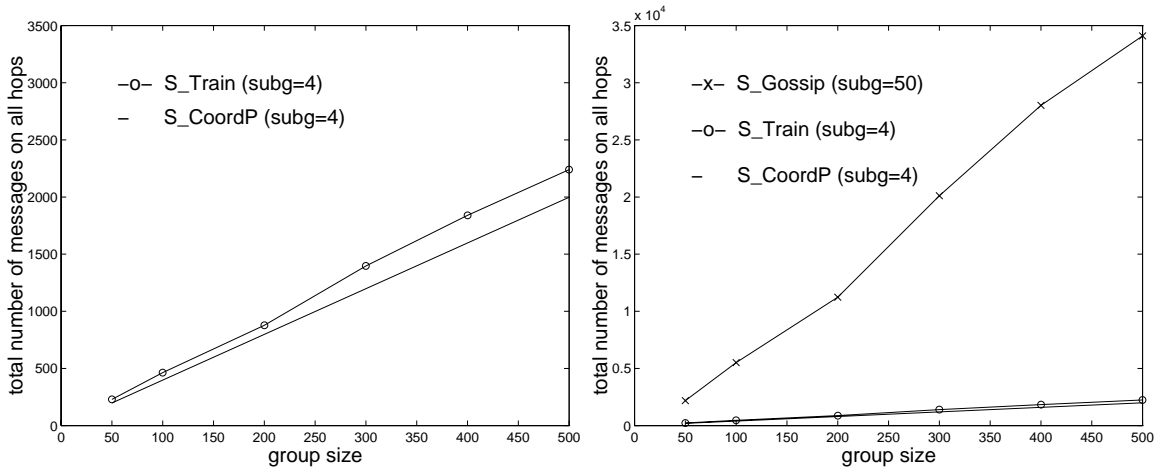
## Simulation Results for Dense Groups with One Sender



(A): four basic protocols

(B): basic protocols (in detail)

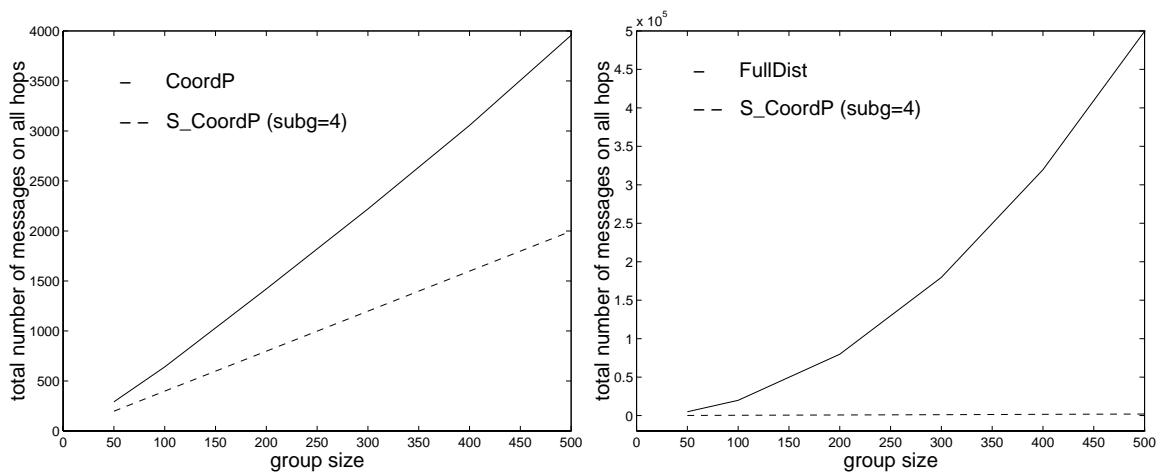
Figure C.1: Total number of messages on all hops for the four basic protocols in dense groups with one sender.



(A): sub-group size = 4

(B): sub-group size = 50

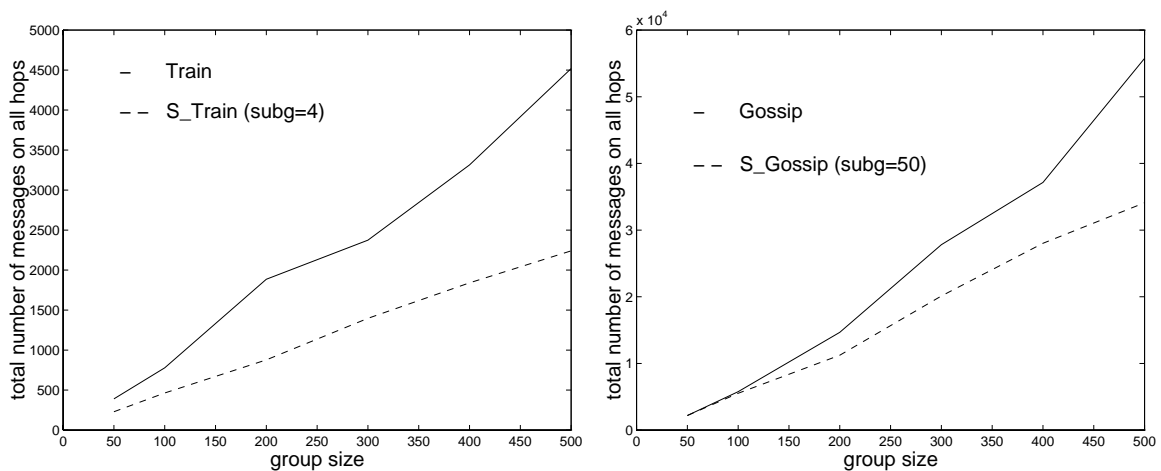
Figure C.2: Total number of messages on all hops for the three structured protocols in dense groups with one sender.



(A): CoordP and S\_CoordP

(B): FullDist and S\_CoordP

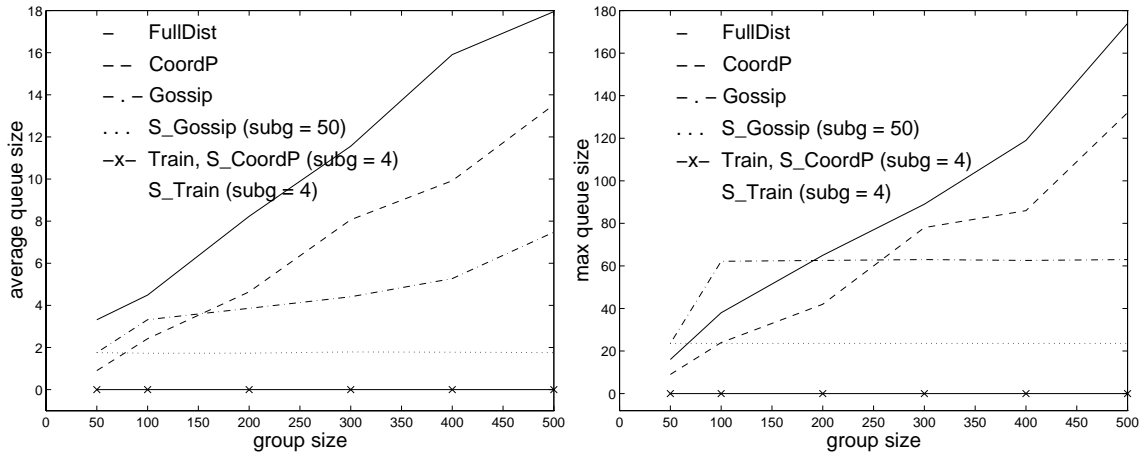
Figure C.3: Total number of messages on all hops for the basic and their corresponding structured protocols in dense groups with one sender (part I).



(A): Train and S\_Train

(B): Gossip and S\_Gossip

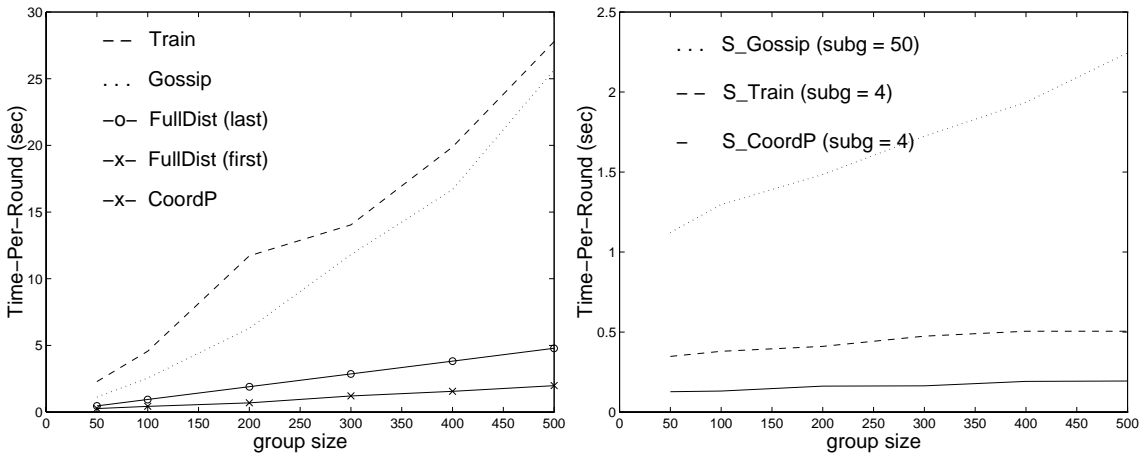
Figure C.4: Total number of messages on all hops for the basic and their corresponding structured protocols in dense groups with one sender (part II).



(A): average queue size

(B): maximum queue size

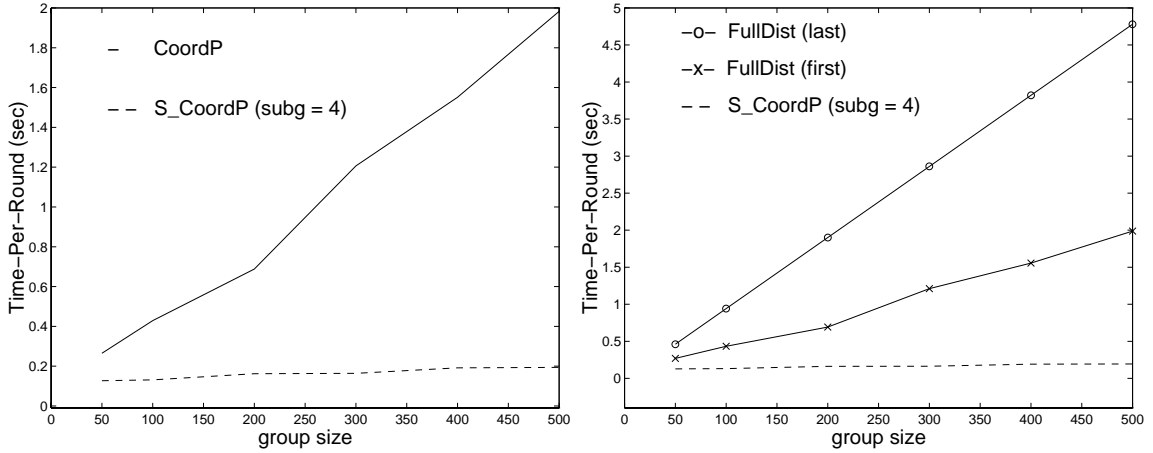
Figure C.5: Average and maximum queue sizes over all the nodes for the basic and structured protocols in dense groups with one sender.



(A): four basic protocols

(B): three structured protocols

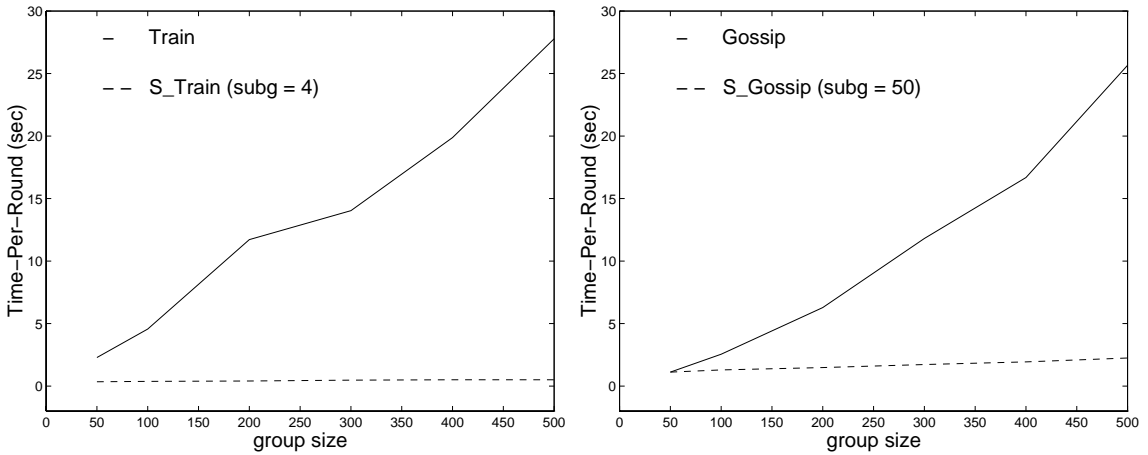
Figure C.6: Time-per-round (TPR) for the basic and structured protocols in dense groups with one sender.



(A): CoordP and S\_CoordP

(B): FullDist and S\_CoordP

Figure C.7: Time-per-round (TPR) for the basic and their corresponding structured protocols in dense groups with one sender (part I).



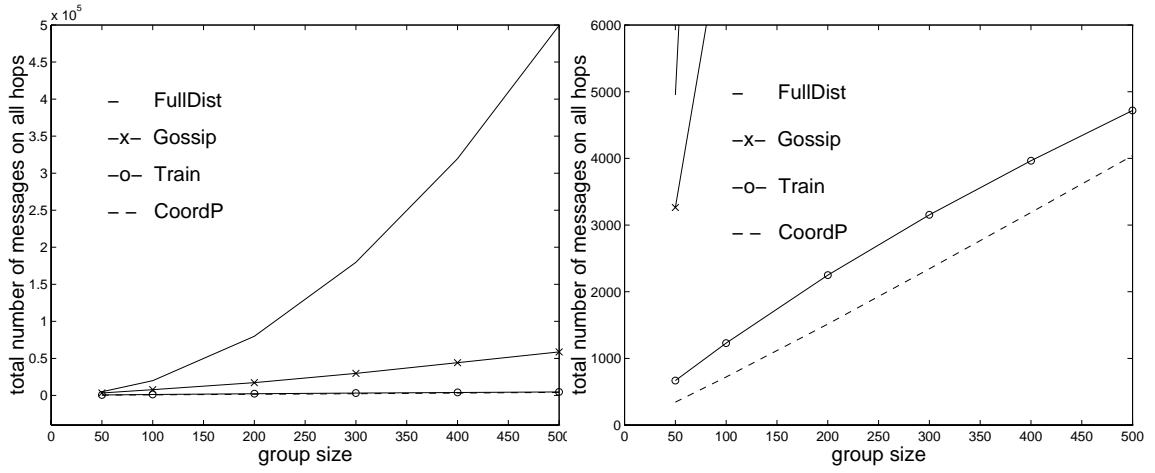
(A): Train and S\_Train

(B): Gossip and S\_Gossip

Figure C.8: Time-per-round (TPR) for the basic and their corresponding structured protocols in dense groups with one sender (part II).

# Appendix D

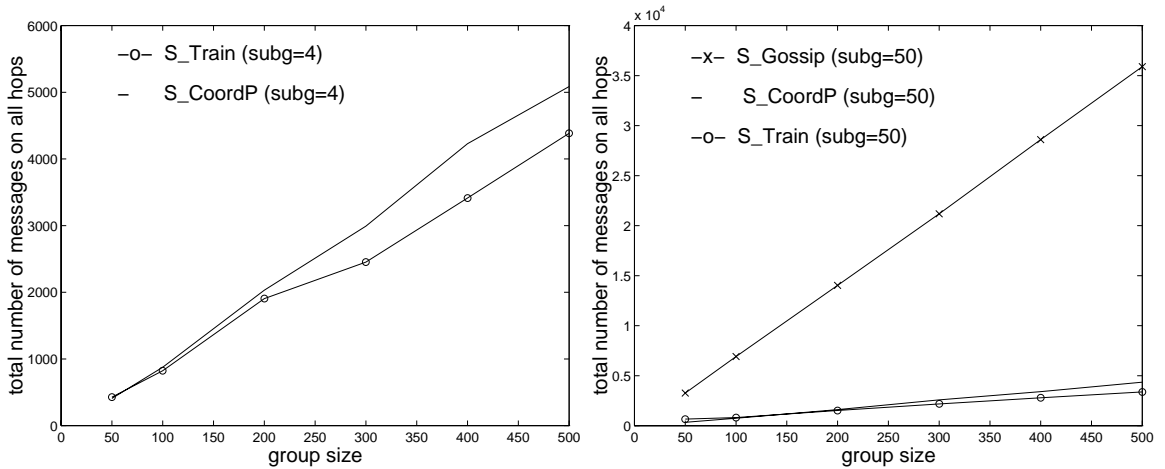
## Simulation Results for Sparse Groups with One Sender



(A): four basic protocols

(B): basic protocols (in detail)

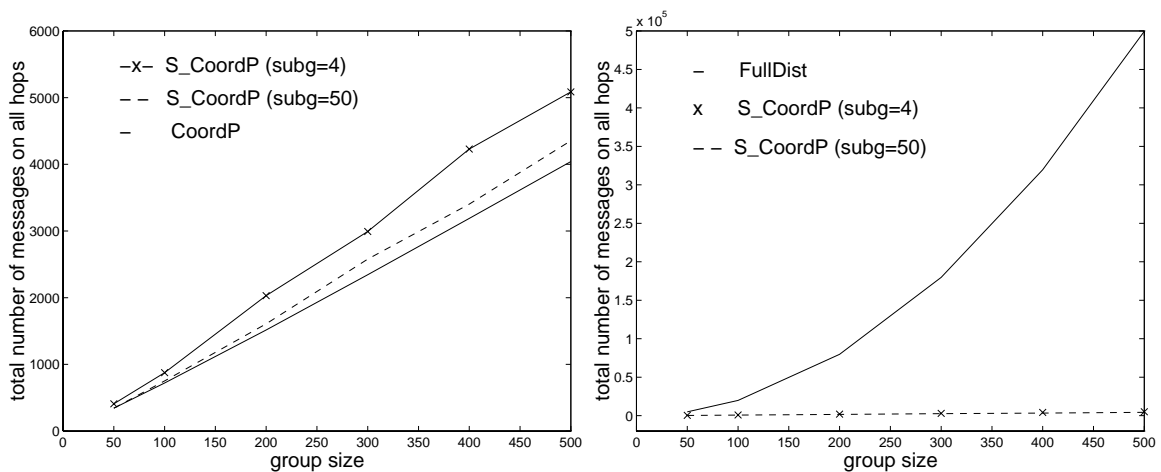
Figure D.1: Total number of messages on all hops for the four basic protocols in sparse groups with one sender.



(A): sub-group size = 4

(B): sub-group size = 50

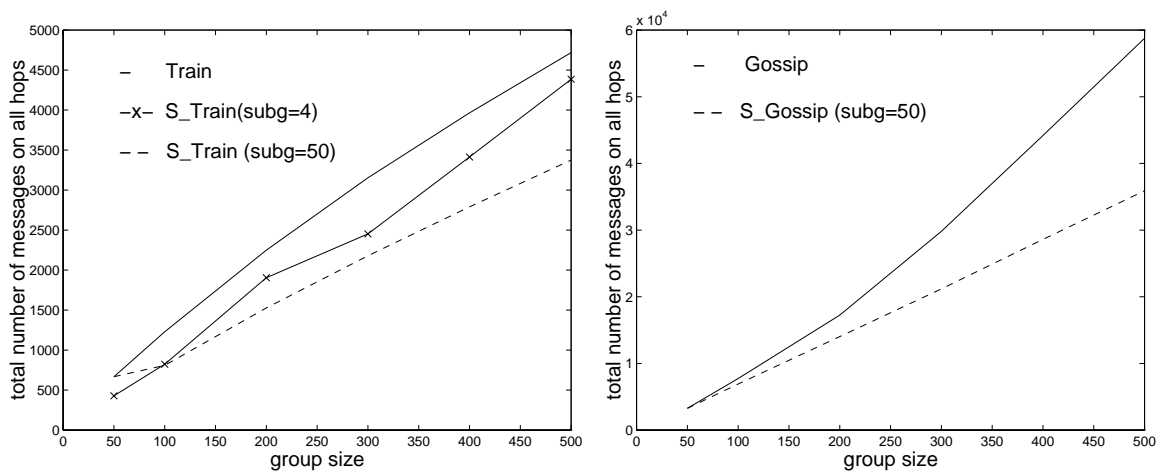
Figure D.2: Total number of messages on all hops for the three structured protocols in sparse groups with one sender.



(A): CoordP and S\_CoordP

(B): FullDist and S\_CoordP

Figure D.3: Total number of messages on all hops for the basic and their corresponding structured protocols in sparse groups with one sender (part I).



(A): Train and S\_Train

(B): Gossip and S\_Gossip

Figure D.4: Total number of messages on all hops for the basic and their corresponding structured protocols in sparse groups with one sender (part II).

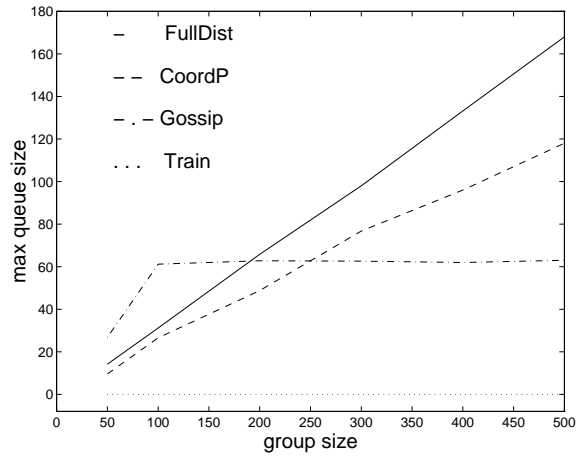
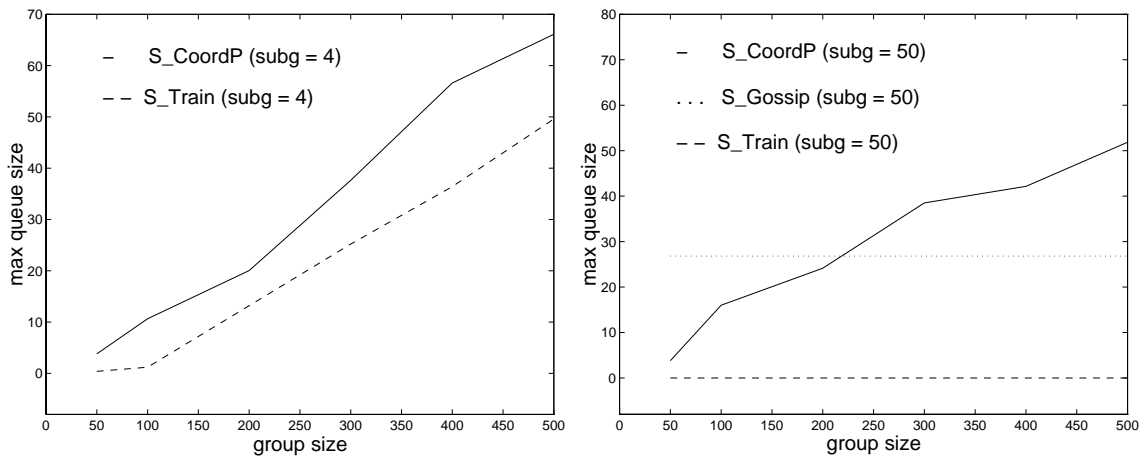


Figure D.5: Maximum queue size over all the nodes for the four basic protocols in sparse groups with one sender.



(A): sub-group size = 4

(B): sub-group size = 50

Figure D.6: Maximum queue size over all the nodes for the structured protocols in sparse groups with one sender.

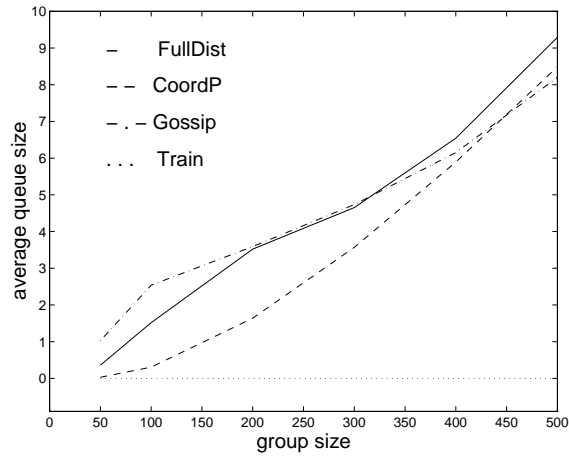
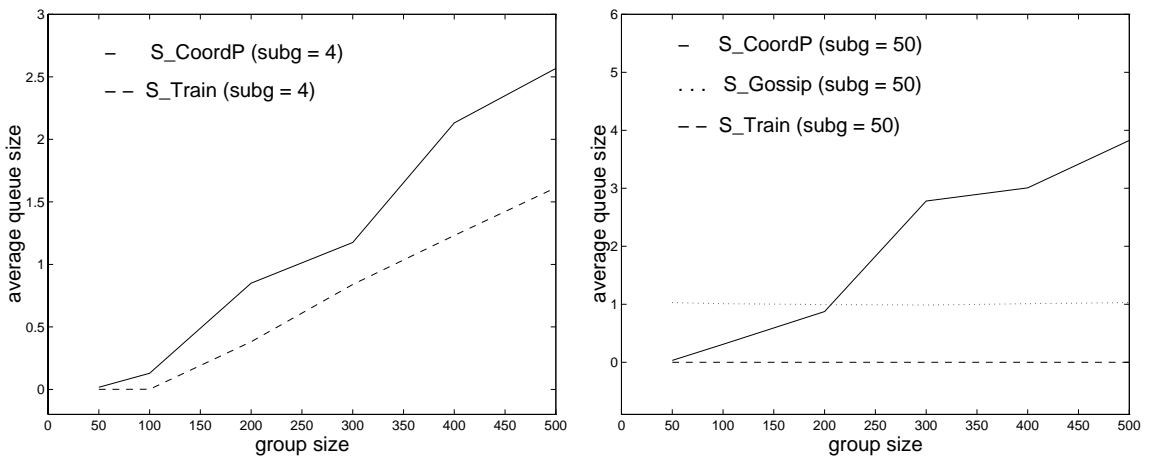


Figure D.7: Average queue size over all the nodes for the four basic protocols in sparse groups with one sender.



(A): sub-group size = 4

(B): sub-group size = 50

Figure D.8: Average queue size over all the nodes for the structured protocols in sparse groups with one sender.

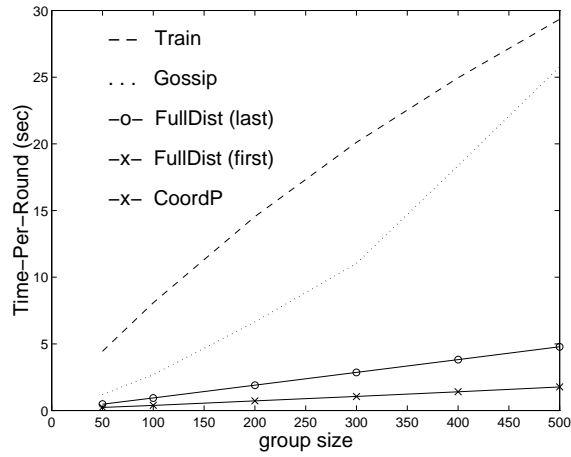
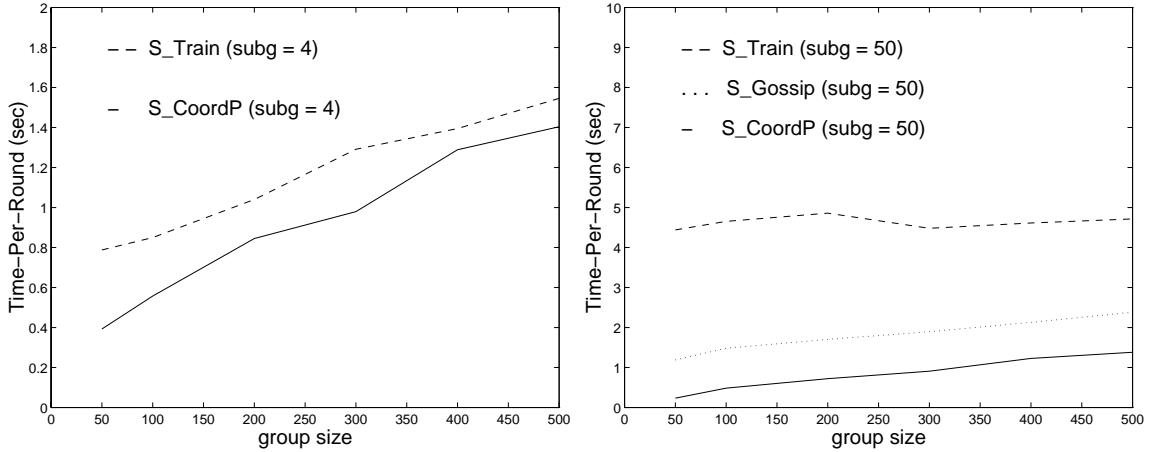


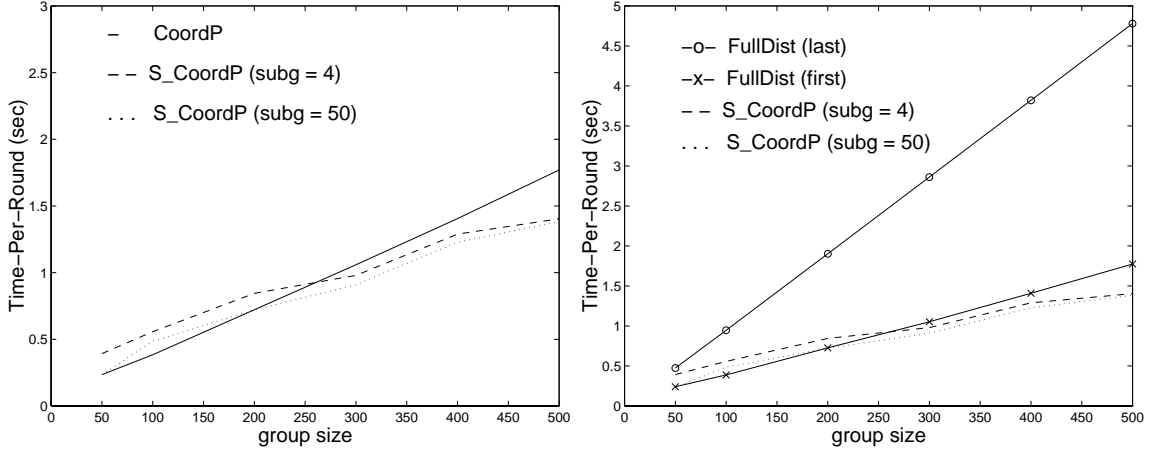
Figure D.9: Time-per-round (TPR) for the four basic protocols in sparse groups with one sender.



(A): sub-group size = 4

(B): sub-group size = 50

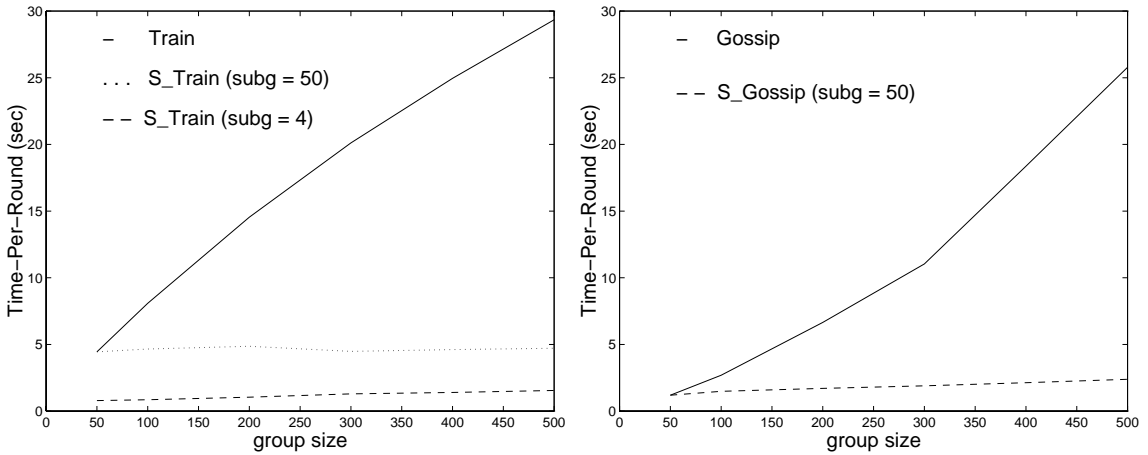
Figure D.10: Time-per-round (TPR) for the structured protocols in sparse groups with one sender.



(A): CoordP and S\_CoordP

(B): FullDist and S\_CoordP

Figure D.11: Time-per-round (TPR) for the basic and their corresponding structured protocols in sparse groups with one sender (part I).



(A): Train and S\_Train

(B): Gossip and S\_Gossip

Figure D.12: Time-per-round (TPR) for the basic and their corresponding structured protocols in sparse groups with one sender (part II).

# Bibliography

- [ADKM92] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *Digest of Papers, The 22nd IEEE International Symposium on Fault-Tolerant Computing Systems*, pages 76–84, July 1992.
- [AKS96] R. Ahuja, S. Keshav, and H. Saran. Design, implementation, and performance of a native mode ATM transport layer. *IEEE/ACM Transactions on Networking*, 4(4):502–515, August 1996.
- [AMMS<sup>+</sup>95] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, November 1995.
- [AMMSB95] D. A. Agarwal, L. E. Moser, P. M. Melliar-Smith, and R. K. Budhia. Reliable ordered delivery across interconnected Local-Area Networks. In *Proceedings of the International Conference on Network Protocols*, pages 365–374, Tokyo, Japan, November 1995.
- [Bai75] N. T. J. Bailey. *The Mathematical Theory of Infectious Diseases and its Applications (second edition)*. Hafner Press, 1975.
- [BFC93] T. Ballardie, P. Francis, and J. Crowcroft. Core based trees (CBT): An architecture for scalable inter-domain multicast routing. In *Proceedings of ACM SIGCOMM'93*, pages 85–95, San Francisco, October 1993.
- [BFvR97] R. Baldoni, R. Friedman, and R. van Renesse. The hierarchical daisy architecture for causal delivery. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 570–577, Baltimore, Maryland, May 1997.
- [BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

- [BHO<sup>+</sup>98] K. P. Birman, M. Hayden, O. Ozkasap, M. Budiu, and Y. Minski. Bimodal multicast. Technical Report CSTR98-1665, Cornell University, Department of Computer Science, January 1998.
- [Bir93] K. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 9(12):36–53, December 1993.
- [Bla90] R. E. Blahut. *Digital Transmission of Information*. Addison-Wesley, 1990.
- [BS72] B. Baker and R. Shostak. Gossips and telephones. *Discrete Mathematics*, 2(3):191–193, June 1972.
- [Car85] R. Carr. The Tandem global update protocol, June 1985.
- [CdBM94] F. Cristian, R. de Beijer, and S. Mishra. A performance comparison of asynchronous atomic broadcast protocols. *Distributed Systems Engineering Journal*, 1(4):177–201, 1994.
- [CLZ87] D. D. Clark, M. L. Lambert, and L. Zhang. NETBLT: A high throughput transport protocol. In *Proceedings of ACM SIGCOMM Computer Communications Review*, pages 353–359, August 1987.
- [CM95] F. Cristian and S. Mishra. The pinwheel asynchronous atomic broadcast protocols. In *Proceedings of the Second International Symposium on Autonomous Decentralized Systems*, Phoenix, AZ, 1995.
- [Com95] Douglas E. Comer. *Internetworking with TCP/IP (3rd Edition)*. Prentice Hall, 1995.
- [Cri91] F. Cristian. Asynchronous atomic broadcast. *IBM Technical Disclosure Bulletin*, 33(9):115–116, February 1991.
- [CT90] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of ACM SIGCOMM'90*, pages 200–208, Philadelphia, PA, September 1990.
- [DC90] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.
- [DEF<sup>+</sup>94] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C.-G. Lue, and L. Wei. An architecture for wide-area multicast routing. In *Proceedings of ACM SIGCOMM'94*, pages 126–135, London, September 1994.

- [DGH<sup>+</sup>87] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 1–12, Vancouver, British Columbia, August 1987.
- [DL93] M. Doar and I. Leslie. How bad is naïve multicast routing? In *Proceedings of IEEE INFOCOM'93*, pages 82–89, San Francisco, CA, March 1993.
- [DMS94] D. Dolev, D. Malki, and R. Strong. An asynchronous membership protocol that tolerates partitions. Technical Report TR94-6, The Hebrew University of Jerusalem, Institute of Computer Science, March 1994.
- [DS93] J. E. Dennis, Jr. and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice Hall Series in Computational Mathematics. Prentice-Hall, Inc., 1993.
- [FJL<sup>+</sup>96] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, November 1996.
- [Fre] R. Frederick. “nv” manual pages.
- [GT92] R. Golding and K. Taylor. Group membership in the epidemic style. Technical Report UCSC-CRL-92-13, UC Santa Cruz, Department of Computer Science, May 1992.
- [GVvR96] K. Guo, W. Vogels, and R. van Renesse. Structured virtual synchrony: Exploring the bounds of virtually synchronous group communication. In *Proceedings of the 7th ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.
- [han79] *Handbook of Mathematics*. High Education Publishing House, Beijing, 1979.
- [Hay98] M. Hayden. *The Ensemble System*. Ph.D. dissertation, Cornell University, Ithaca, NY, January 1998. Also available as Department of Computer Science Technical Report CSTR98-1662.
- [Hof96a] M. Hofmann. Adding scalability to transport level multicast. In *Proceedings of Third COST 237 Workshop - Multimedia Telecommunications and Applications*, Barcelona, Spain, November 25-27 1996.

- [Hof96b] M. Hofmann. A generic concept for large-scale multicast. In *Broadband Communications (B. Plattner ed.), Lecture Notes in Computer Science, No. 1044, Proceedings of 1996 International Zurich Seminar on Digital Communications*. Springer Verlag, February 1996.
- [HSC95] H. Holbrook, S. Singhal, and D. Cheriton. Log-based receiver-reliable multicast for distributed interactive simulation. In *Proceedings of ACM SIGCOMM'95*, pages 328–341, Cambridge, MA, August 1995.
- [Jac93] V. Jacobson, September 1993. 1993 ARPA Networking PI Meeting.
- [JM] V. Jacobson and S. McCanne. “vat” manual pages.
- [Koz91] D. Kozen. *The Design and Analysis of Algorithms*. Springer Verlag, 1991.
- [KP93] J. Kay and J. Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *Proceedings of ACM SIGCOMM'93*, San Francisco, CA, September 1993.
- [KTHB89] M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and H. E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, October 1989.
- [Kum95] V. Kumar. *Mbone: Interactive Multimedia on the Internet*. New Riders Publishing, Indianapolis, Indiana, USA, 1995.
- [Lev96] B. N. Levine. A comparison of known classes of reliable multicast protocols. Technical report, University of California, Santa Cruz, June 1996. Master Thesis.
- [MAMSA94] L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 56–65, Poznan, Poland, June 1994.
- [MCS91] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991.
- [MF95] S. McCanne and S. Floyd. Ns (network simulator). Available via <http://www-nrg.ee.lbl.gov/ns>, 1995.
- [Mit97] S. Mittra. Iolus: A framework for scalable secure multicasting. In *ACM SIGCOMM'97*, pages 277–288, Cannes, France, September 1997.

- [MMSA<sup>+</sup>96] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.
- [MSMA91] P. M. Melliar-Smith, L. E. Moser, and D. A. Agarwal. Ring-based ordering protocols. In *Proceedings of the IEEE International Conference on Information Engineering (Singapore)*, pages 882–891, December 1991.
- [Mul93] S. Mullender, editor. *Distributed Systems*. ACM Press, Addison-Wesley, 1993.
- [OD83] D. Oppen and Y. Dalal. The clearinghouse: A decentralized agent for locating named objects in a distributed environment. *ACM Transaction on Office Information Systems*, 1(3):230–253, July 1983.
- [Pal85] E. Palmer. *Graphical Evolution: An Introduction to the Theory of Random Graphs*. John Wiley & Sons, 1985.
- [Pos81] J. B. Postel. Transmission control protocol. *RFC 793*, September 1981.
- [PTK94] S. Pingali, D. Towsley, and J. F. Kurose. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. In *SIGMETRICS'94, Performance Evaluation Review*, volume 22, May 1994.
- [RJ87] S. Ramakrishnan and B. N. Jain. A negative acknowledgement with periodic polling protocol for multicast over LANs. In *Proceedings of IEEE Infocom'87*, pages 502–511, March 1987.
- [Sch92] H. Schulzrinne. Voice communication across the internet: A network voice terminal. Technical Report UM-CS-1992-050, Dept. of Computer Science, University of Massachusetts, Amherst, July 1992.
- [SDW92] W. T. Strayer, B. J. Dempsey, and A. C. Weaver. *XTP The Express Transfer Protocol*. Addison-Wesley Publishing Company, Inc., 1992.
- [SKB89] M. Simmons, R. Koshela, and I. Bucher, editors. *Instrumentation for Future Parallel Computing Systems*. ACM Press, 1989.
- [SLPB97] K. Sabnani, J.C. Lin, S. Paul, and S. Bhattacharyya. Reliable Multicast Transport Protocol(RMTP). *IEEE Journal on Selected Areas in Communication*, April 1997.

- [Sta97] StarBurst MFTP compared to today's file transfer protocols: A white paper, 1997. StarBurst Communications Corporation. <http://www.starburstcom.com>.
- [The92] D. Theriault. BLAST, an experimental file transfer protocol, March 1992.
- [Vog96] W. Vogels. World wide failures. In *Proceedings of the 7th ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.
- [vR96] R. van Renesse. Masking the overhead of protocol layering. In *Proceedings of ACM SIGCOMM'96*, pages 96–104, August 1996.
- [vRBM96] R. van Renesse, K. P. Birman, and S. Maffei. Horus, a flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.
- [vRMH98] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. Technical Report In Preparation, Cornell University, Department of Computer Science, Ithaca, NY, 1998.
- [WM87] R. Watson and S. Mamrak. Gaining efficiency in transport services by appropriate design and implementation choices. *ACM Transactions on Computer Systems*, 5(2):97–120, May 1987.
- [WPD88] D. Waitzman, C. Partridge, and S. Deering. Distance vector multicast routing protocol. *RFC-1075*, November 1988.
- [XTP95] Xpress Transport Protocol specification (XTP revision 4.0). Available via <http://www.ca.sandia.gov/ntp/ntp.html>, March 1995.