# Performance Evaluation of Distributed Algortihms over the Internet

by

**Omar Bakr**

S.B. in Computer Science and Engineering, MIT (2001)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

**Master of Engineering in Electrical Engineering and Computer Science**

at the

**Massachusetts Institute of Technology**

February 2003

© 2003 Omar Bakr. All rights reserved.

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

Author_____

Department of Electrical Engineering and Computer Science
February 11, 2003

Certified by_____

Dr. Idit Keidar
Senior Lecturer, The Technion – Israel Institute of Technology
Thesis Supervisor

Accepted by_____

Professor Arthur C. Smith
Department of Electrical Engineering and Computer Science
Chairman, Department Committee on Graduate Theses

**Performance Evaluation of**
**Distributed Algortihms over the Internet**

by

Omar Bakr

Submitted to the Department of Electrical Engineering and Computer Science
on February 11, 2003, in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

We study the running time of distributed algorithms deployed in a widely distributed setting over the Internet using TCP. We consider two simple primitives. Both primitives corresponds to a *communication round* which is employed by many different algorithms and systems. In the first primitive, every host sends information to every other host. The second primitive propagates information from a quorum of hosts to a quorum of hosts. Both primitives occur in numerous distributed algorithms. We experiment with four algorithms that typically implement the first primitive and two that implement the second. We run our experiments on twenty-eight hosts at geographically disperse locations over the Internet. We observe that message-loss has a large impact on algorithm running times, which causes leader-based algorithms to usually outperform decentralized ones. We also observe that algorithms, in which hosts need only to hear from a quorum, are more reliable, efficient, and tolerant to bad links than algorithms where every host is required to hear from every other host in the system.

Thesis Supervisor: Idit Keidar, Ph.D.
Title: Senior Lecturer, The Technion – Israel Institute of Technology

## Acknowledgments

My principle thanks and appreciation rests with my thesis advisor, Idit Keidar. She not only introduced me to this problem area and suggested my research methods, but also encouraged me to believe that I could make an original contribution of my own. I am particularly grateful that she was willing to usher me through the initial presentation of these finding at PODC '02. A graduate student could not hope for a more enthusiastic and supportive adviser. I am indebted to Professor Nancy Lynch for welcoming me to her Theory of Distributed Systems Group. During my research I had the good fortune of receiving assistance from Carl Livadas and Roger Khazan.

I thank all those who are hosting our experiments on their machines. Many of the machines we use belong to the RON project [3] at MIT, which is funded by DARPA; I am especially thankful to Dave Andersen for technical assistance with these machines. Keith Marzullo and Geoff Voelker gave us access to the machine in UCSD, and Yuh-Jzer Joung let us use his machine in Taiwan. Danny Dolev provided me with access to the computer networks of the Hebrew University of Jerusalem, and Ilya Shanayderman and Danny Bickson were gracious enough to provide me with technical assistance whenever I needed it. The account in New Zealand was made available by Mark Moir, and the account in Australia by Alan Fekete and Greg Ryan. Rachid Guerraoui provided us with the account in Switzerland and Sebastien Baehni was always there when I needed support. Lars Engebretsen gave us access to a machine in Sweden, and Roberto De Prisco gave us access to his machine in Italy. Ricardo Jimenez Peris gave access to several machines in Spain. Luis Rodigues provided us with an account in Portugal, Cevat Sener provided us with an account in Turkey, and Francisco Brasileiro gave us an account in Brazil.

During the final stages of writing this thesis I am grateful for the proof reading and humor of Dr. Eric Kupferberg. I am particularly indebted to my parents and my family. Words cannot begin to express my love and respect for them. And above all, I am grateful that the most gracious one has commanded humble men to want to know.

This research was performed at the Theory of Distributed Systems Group of the Massachusetts Institute of Technology. This work was supported by Air Force Aerospace Research (OSR) contract F49620-00-1-0097 and MURI award F49620-00-1-0327, and Nippon Telegraph and Telephone (NTT) grant MIT9904-12.

# Contents

**Contents** . . . . . 7
**List of Figures** . . . . . 9
**List of Tables** . . . . . 11

**1 Introduction and Background** **13**
1.1 Introduction . . . . . 13
1.2 Related Work . . . . . 18
1.2.1 Internet Measurments . . . . . 18
1.2.2 Metrics . . . . . 19
1.2.3 Quorums . . . . . 19

**2 Methodology** **21**
2.1 The Hosts . . . . . 21
2.2 Server Implementation . . . . . 22
2.3 Running Times and Clock Skews . . . . . 23
2.4 Running Time Distribution over TCP/IP . . . . . 25
2.5 Pseudo Code of the Algorithms . . . . . 25
2.5.1 The Gather-All Primitive . . . . . 27
2.5.2 The Gather-Quorum Primitive . . . . . 31

**3 The Gather-All Primitive** **35**
3.1 The Effect of Message Loss . . . . . 35
3.2 Experiment I . . . . . 35
3.3 Experiment II: Excluding the Lossiest Host . . . . . 39
3.4 The Impact of Latency . . . . . 40
3.5 The Running Time of Ring . . . . . 41
3.6 Latency Changes over Time . . . . . 42
3.7 Latency and Loss . . . . . 43

3.8 The Triangle Inequality . . . . . 43

**4 The Gather-Quorum Primitive** **47**

4.1 Comparing the Two Primitives . . . . . 47

4.1.1 Complete Probe Sets . . . . . 48

4.1.2 Minimal Probe Sets . . . . . 51

4.2 The Size of the Probe Set . . . . . 53

4.2.1 Quorum-Leader . . . . . 53

4.2.2 Quorum-to-Quorum . . . . . 58

**5 Conclusions** **63**

**References** **65**

# List of Figures


1.1 The message flow of the four algorithms. Initiator shown in gray. . . . . 14

1.2 The Table quorum system of 6 x 6, with one quorum shaded. . . . . 15

2.1 Variables used by the algorithms. . . . . 26

3.1 Histograms of overall running times, Experiment I, runs up to 1.3 seconds. 44

3.2 Histograms of local running times, Experiment I, runs up to 1.3 seconds. . . 45

3.3 Histograms of overall running times, runs up to 2 seconds, Experiment III. 46

4.1 Comparing the gather-all and gather-quorum primitives using the leader based algorithm and complete probe sets (results from experiment IV). . . . 49

4.2 Comparing the gather-all and gather-quorum primitives using the all-to-all algorithm and complete probe sets (results from experiment V). . . . . 50

4.3 Comparing the gather-all and gather-quorum primitives using the leader based algorithm and minimal probe sets (results from experiment IV). . . . 52

4.4 Comparing the gather-all and gather-quorum primitives using the all-to-all algorithm and minimal probe sets (results from experiment V). . . . . 53

4.5 The cumulative distribution of local running times of quorum-leader algorithms (using the majority quorum system) initiated at different hosts during Experiment IV, runs up to 4 seconds ($n$ denotes the number of hosts in the probe set). . . . . 54

4.6 The cumulative distribution of local running times of quorum-leader algorithms (using the table quorum system) initiated at different hosts during Experiment IV, runs up to 4 seconds ($n$ denotes the number of table rows in the probe set). . . . . 59

4.7 The cumulative distribution of overall running times of majority for all-to-all initiated by MIT during Experiment V, runs up to 2 seconds ($n$ denotes the number of hosts the probe set). . . . . 60

4.8 The cumulative distribution of local running times of majority for all-to-all initiated by MIT during Experiment V, runs up to 2 seconds ($n$ denotes the number of hosts in the probe set). . . . . 60

4.9 The cumulative distribution of overall running times of table for all-to-all initiated by MIT during Experiment V, runs up to 2 seconds ($n$ denotes the number of table rows the in the probe set). . . . . 62

# List of Tables


3.1 Network characteristics during Experiment I. . . . . 36

3.2 Measured running times, milliseconds, Experiment I. . . . . 37

3.3 Network characteristics during Experiment II. . . . . 39

3.4 Measured overall and local running times, Experiment II. . . . . 40

3.5 Network characteristics during Experiment III. . . . . 41

3.6 Measured running times, milliseconds, Experiment III. . . . . 42

4.1 Table quorum system in experiment IV. . . . . 48

4.2 Table quorum system in experiment V. . . . . 48

4.3 Failure percentages for quorum-leader algorithms using the majority quorum system at different initiators for all possible probe set sizes (size is measured in terms the number of hosts). Results compounded during Experiment IV. 50

4.4 Failure percentages for quorum-leader algorithms using the table quorum system at different initiators for all possible probe set sizes (size is measured in terms table rows). Results were compounded during Experiment IV. . . 51

4.5 Link characteristics from TW to other hosts during experiment IV. . . . . 55

# Chapter 1

# Introduction and Background

## 1.1 Introduction

It is challenging to predict the end-to-end running time of a distributed algorithm operating over TCP/IP in a wide-area setting. It is often not obvious which algorithm would work best in a given setting. For instance, would a decentralized algorithm outperform a leader-based one? Answering such questions is difficult for a number of reasons. Firstly, because end-to-end Internet performance itself is extremely hard to analyze, predict, and simulate [8]. Secondly, end-to-end performance observed on the Internet exhibits great diversity [22, 33], and thus different algorithms can prove more effective for different topologies, and also for different time periods on the same topology. Finally, different algorithms can perform better under different performance metrics.

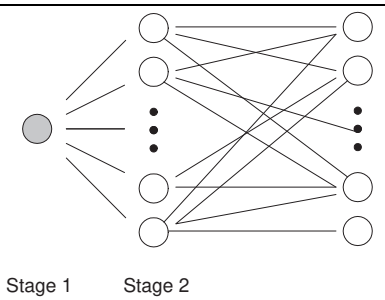
In this thesis, we study the running time of distributed algorithms over the Internet. Our experiments span twenty-eight hosts, widely distributed over the Internet – in Korea, Taiwan, Israel, Australia, New Zealand, and several hosts across Europe and North America. Some of the hosts reside at academic institutions and others on commercial ISP networks. We present data that was gathered over several months. The hosts communicate using TCP/IP. TCP is a commonly used protocol on the Internet, and therefore evaluating systems that use it is of interest. Moreover, it was feasible for us to deploy a TCP-based system because TCP does not generate excessive traffic at times of congestion, and because firewalls at some of our hosts block UDP traffic.

We consider a fixed set of hosts engaged in a distributed algorithm. We evaluate two simple primitives that correspond to a *communication round* executed by a distributed algorithm. Both primitives are employed by many different algorithms and systems, e.g., Byzantine agreement [18], atomic commit [9, 13, 29], state-machine replication [17], group membership [15], and updates of routing tables. Thus, our study has broad applicability. First we look at *gather-all*, which propagates a small amount of information from every

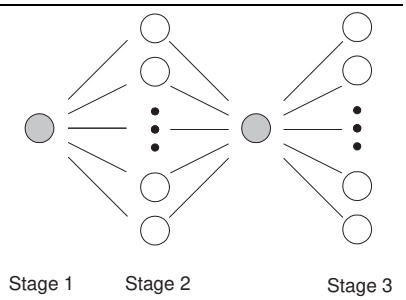
host to all other hosts that are connected to it. The primitive can be initiated by any one of the hosts, called the *initiator*, and it terminates once information from every host has propagated to all of the hosts.

We evaluate the following commonly used algorithms implementing this primitive:

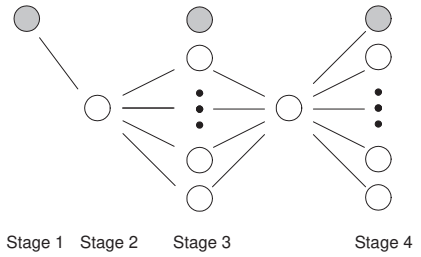
**Figure 1.1** The message flow of the four algorithms. Initiator shown in gray.



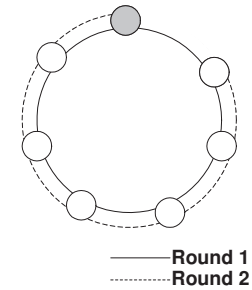

(a) The all-to-all algorithm.




(b) The leader algorithm.




(c) The secondary leader algorithm.




(d) The ring algorithm.

- *all-to-all*, where the initiator sends a message to all other hosts, and each host that learns that the algorithm has been initiated sends messages to all the other hosts. This algorithm is structured similar to decentralized two-phase commit, some group membership algorithms (e.g., [15]), and the first phases in decentralized three-phase commit algorithms, (e.g., [10, 29]). The algorithm flow is depicted in Figure 1.1(a).

- *leader*, where the initiator acts as the leader. After the initiator sends a message to all other hosts, the hosts respond by sending messages to the leader. The leader *aggregates* the information from all the hosts, and sends a message summarizing all the inputs to all the hosts. This algorithm is structured like two-phase commit [9], and like the first two of three communication phases in three-phase commit algorithms, e.g., [13, 29]. The algorithm flow is depicted in Figure 1.1(b).

- *secondary leader*, where a designated host (different from the initiator) acts as the leader. The initiator sends a message to the leader, which then initiates the leader-based algorithm. The algorithm flow is depicted in Figure 1.1(c). This algorithm

structure is essentially a spanning tree of depth one, with the secondary leader being the root and all other hosts being leaves.

- *logical ring*, where messages propagate along the edges of a logical ring. This algorithm structure occurs in several group communication systems, e.g., [1]. The algorithm flow is depicted in Figure 1.1(d).

The second primitive we evaluate is the *gather-quorum* primitive. This primitive involves the use of quorum systems. A *quorum system* for a universe of servers is a collection of subsets (called quorums) of servers, each pair of which have a nonempty intersection. Quorum systems are typically used to increase the availability and efficiency of replicated services.The gather-quorum primitive propagates a small amount of information from a quorum to a quorum. Like the gather-all primitive, This primitive can also be initiated by any one of the hosts, and it terminates once information from a quorum of hosts has propagated to a quorum of hosts.

We evaluate this primitive with two widely used quorum systems:

- *Majorities*, where every set that includes a majority of servers is a quorum.
- *Table*, Suppose that the number of servers $n = k^2$ for some integer $k$. Arrange the servers into a $k$ x $k$ table, as shown in Figure 1.2. A quorum is a union of a full row and one element from each row above the full row. In reality the number of servers is not always a perfect square, in which case some rows might be larger than others. In any case, the quorums in the Table quorum system are of size $O(\sqrt{n})$.

**Figure 1.2** The Table quorum system of 6 x 6, with one quorum shaded.

| | | | | | |
|---|---|---|---|---|---|
| Host1 | Host2 | Host3 | Host4 | Host5 | Host6 |
| Host7 | Host8 | Host9 | Host10 | Host11 | Host12 |
| Host13 | Host14 | Host15 | Host16 | Host17 | Host18 |
| Host19 | Host20 | Host21 | Host22 | Host23 | Host24 |
| Host25 | Host26 | Host27 | Host28 | Host29 | Host30 |
| Host31 | Host32 | Host33 | Host34 | Host35 | Host36 |

Initiators that invoke an algorithm that implements the gather-quorum primitive have a decision to make. One option is to probe every host and wait for a quorum to respond.

A second option is to probe exactly one quorum and wait for that particular quorum to respond. These two cases are the extreme cases. Another option is to probe multiple quorums and wait for one of them to respond. We call the set of hosts probed by the initiator the *probe set*. The probe set has a lot implications on the running time of an algorithm. One the one hand, the initiator prefers to keep the probe set as small as possible to reduce the overall load on the system. One the other hand, the initiator also prefers to probe more hosts in order to reduce the probability of failure. We study the impact of the choice of probe set on the algorithm running time and avilability.

We evaluate the following commonly used algorithms implementing this primitive:

- *quorum-to-quorum*, where the initiator sends a message to every host in the probe set, and each host that hears from the initiator sends messages to all hosts in the probe set. The algorithm terminates as soon as quorum hears from a quorum and fails otherwise. This algorithm is similar to the all-to-all algorithm presented above. This algorithm is structured like some atomic broadcast algorithms (e.g.,Corel [12]), some consensus algorithms (e.g., [27]), and the final phase in decentralized three-phase commit algorithms, (e.g., [10]).

- *quorum-leader*, where the initiator acts as the leader. After the initiator sends a message to every host in the probe set, the hosts respond by sending messages to the leader. Once the leader hears from a quorum, it aggregates the information from hosts in that quorum, and sends a message summarizing all the inputs to all the hosts in the probe set. This algorithm is similar to the leader algorithm presented above. The algorithm is structured similar to Lamport's Paxos algorithm [16], the algorithm used in the Frangipani distributed file system [31], some consensus algorithms (e.g., [6]), and like the final two of three communication phases in three-phase commit algorithms, e.g., [13, 29].

We run a single process at each geographical location. We do not address issues related to scaling the number of processes, as we believe that such issues are orthogonal to our study. Using a 2-level hierarchy, algorithms of the sort we consider can be made to work effectively with several hundreds of processes. Such a hierarchy is used, e.g., in [11, 15], where the top level of the hierarchy consists of 5–20 representatives (servers) at disperse geographical locations. Each representative gathers information from and propagates information to processes that are proximate to it. Algorithms like those considered here are typically run among the representatives. Thus, our study is applicable to systems that implement scalability in this manner. Our study is also somewhat applicable to large scale client-server systems like Fleet [19] and SBQ-L [20] where clients contact a quorum of the servers. Our study is however not applicable to systems that implement massive scalability, e.g., using gossip-based or peer-to-peer algorithms.

We measure the *overall* running time of an algorithm from the time it starts at some host until it terminates at all hosts (or until it terminates at a quorum of hosts in the cases where we are considering the performance of quorum systems), as well as the *local* running time at a given host.

The typical theoretical metric used to analyze the running time of distributed algorithms is the number of message exchange rounds the algorithm performs, or the number of *communication steps* in case of a non-synchronous system (e.g., [14, 15, 27]). According to this metric, we get the following overall running times: 2 communication steps for the all-to-all and quorum-to-quorum algorithms; 3 communication steps for the leader and quorum-leader algorithms; 4 communication steps for secondary leader; and $2n - 1$ steps for the ring algorithm in a system with $n$ hosts. In contrast to what this metric suggests, in Section 3.1 we observe that in certain settings the secondary leader algorithm achieves the best overall running time, whereas all-to-all often performs the worst. The running time of ring was usually less than double the running times of the other algorithms.

Why does the communication step metric fail to capture actual algorithm behavior over the Internet? First, not all communication steps have the same cost, e.g., a message from MIT to Cornell can arrive within 20 ms., while a message from MIT to Taiwan may take 125 ms. Second, the latency on TCP links depends not only on the underlying message latency, but also on the loss rate. If a message sent over a TCP link is lost, the message is retransmitted after a timeout which is larger than the average round-trip time (RTT) on the link. Therefore, if one algorithm message is lost, the algorithm's overall running time can be more than doubled. Since algorithms that exchange less messages are less susceptible to message loss, they are more likely to perform well when loss rates are high. This explains why the overall running time of all-to-all is miserable in the presence of lossy links. Additionally, message latencies and loss rates on different communication paths on the Internet often do not preserve the triangle inequality [3, 15, 26] because the routing policies of Internet routers often do not choose an optimal path between two hosts. This explains why secondary leader can achieve better performance by refraining from sending messages on very lossy or slow paths.

We analyze our experimental results, and explain the observed algorithm running times in terms of the underlying network characteristics – latency and loss rates. Due to the great variability of running times, the average running time is not indicative of an algorithm's typical behavior. We therefore focus on the *distribution* of running times.

The communication step metric is widely used due to its ease of use. Several other performance models, e.g., [7, 24, 32], have been used to analyze distributed or parallel algorithms (cf. Section 1.2). However, these do not realistically model algorithm behavior over the Internet. At the end of this thesis, we suggest a refinement to the standard metric, which gives a more realistic account of an algorithm's efficiency, and at the same time is easy to

employ.

The rest of this thesis is organized as follows: Section 1.2 discusses related work. Chapter 2 describes the experiment setup and methodology. The following two chapters analyze experimental results: Chapter 3 discusses gather-all algorithms and how their running times are influenced by latency and message loss. Chapter 4 compares the two primitives and studies the relationship between the probe set and the running time of gather-quorum algorithms. Chapter 5 concludes the thesis and suggests an alternative performance metric.

## 1.2 Related Work

### 1.2.1 Internet Measurments

Obtaining data on different aspects of Internet communication is an emerging research field. Some present research in this area focuses on measuring and analyzing the constancy of Internet path characteristics such as routing, loss, and throughput [22, 33]. Such research focuses primarily on point-to-point communication, and not on the performance of distributed algorithms. A related project, pursued by Chandra et al. [5], studies the nature of communication failures, including duration and location, and how they effect the end-to-end availability of wide-area services. Another study, by Amir and Wool [2], evaluates the availability of different quorum systems over the Internet. These research efforts are orthogonal and complementary to ours.

#### Triangle Inequality and Overlay Networks

The fact that Internet routing often does not select optimal paths was previously observed by a number of projects – Detour [25, 26], Moshe [15], and RON [3]. These projects construct overlay networks and improve performance by routing messages over these overlays on better paths than would be chosen by Internet routing. In contrast, we neither assume an overlay infrastructure, nor route messages through hosts that are not participating in the current instance of the algorithm. Moreover, the aforementioned projects use overlays in order to find better paths for *point-to-point* communication only. When an overlay is used at the routing level, as in these projects, messages from the same source that are routed through the same host to different destinations are not merged into a single message. For example, let us consider the all-to-all algorithm running over an overlay that routes messages from Taiwan to the Netherlands via Cornell. Taiwan would send identical messages to Cornell and the Netherlands, which would be sent as two separate messages on the link from Taiwan to Cornell. Likewise, the overlay would not combine the information sent from Taiwan and Cornell to the Netherlands into a single message. Such sending of multiple messages

increases the probability of some message being lost, which increases the average running time.

### 1.2.2 Metrics

Another line of research focuses on providing a theoretical framework for predicting and evaluating the performance of parallel and distributed algorithms. A number of papers, e.g. [7, 24, 28, 32], focus on settings where message processing overhead is significant, and show that this favors algorithms that send fewer messages. While our results also illustrate the advantage of sending fewer messages, the reasons for this are different: in our setting, it is due to high variability of message latency (due to loss) rather than processing overhead, which is negligible in our setting. The conclusions from such studies do not, in general, apply to our setting. For example., leader has a high processing overhead (at the leader), but this does not hamper its performance in our setting. Moreover, these analyses assume that the evaluated algorithm is the only source of overhead in the system. In contrast, over the Internet, the evaluated algorithms have little impact on the total overhead of the system.

### 1.2.3 Quorums

The primary foci for research on evaluation of quorum systems are availability and load. The load typically is evaluated assuming a probe set consisting of a single quorum. A common metric used to evaluate the availability of quorum systems is the probability of failure. Quorum systems fail when no quorum exists. Different approaches have been used to study this metric. One approach is using probabilistic models to obtain theoretical results on the probability of failure [21, 23]. While this approach is most rigorous, it makes oversimplifying assumptions about the underlying network. Assumptions such as independent failures and full connectivity generally do not hold in the Internet. Another approach used by Amir and Wool [2] involves running experiments consisting real hosts connected to the Internet and using a group membership protocol to track failures and network partitions. The availability of different quorum systems is then evaluated based on the traces the were collected from the experiments. However, none of these studies provide an adequate framework for evaluating the running times of distributed algorithms that use quorum systems.

# Chapter 2

# Methodology

## 2.1 The Hosts

We use the following 28 hosts (the majority of which is part of the RON testbed [3]) in our experiments:

- *MIT*, at the Massachusetts Institute of Technology, Cambridge, MA, USA;
- *MA1*, at a commercial ISP in Cambirdge, MA, USA;
- *MA2*, at a second commercial ISP in Cambridge, MA, USA;
- *MA3*, at a commercial ISP in Martha's Vineyard, MA, USA;
- *NYU*, at New York University, New York, NY, USA;
- *CU*, at Cornell University, Ithaca, NY, USA;
- *NY*, at a commercial ISP in New York, NY, USA;
- *CMU*, at Carnegie Mellon University, Pittsburgh, PA, USA;
- *NC*, at a commercial ISP in Dhuram, NC, USA;
- *Emulab* , at the University of Utah, Salt Lake City, UT, USA;
- *UT1*, at a commercial ISP in Salt Lake City, UT, USA;
- *UT2*, at a second commercial ISP in Salt Lake City, UT, USA;
- *UCSD*, at the University of California San Diego, San Diego, CA, USA;
- *CA1*, at a commercial ISP in Foster City, CA, USA;

- *CA2*, at Intel Labs in Berkeley, CA, USA;
- *CA3*, at a commercial ISP in Palo Alto, CA, USA;
- *CA4*, at a commercial ISP in Sunnyvale, CA, USA;
- *Canada (CND)*, at a commercial ISP in Nepean, ON, Canada;
- *Sweden (SWD)*, at Lulea University of Technology, Lulea, Sweden;
- *Netherlands (NL)*, at Vrije University, Amsterdam, the Netherlands;
- *Greece (GR)*, at the National Technical University of Athens, Athens, Greece;
- *Switzerland (Swiss)*, at the Swiss Federal Institute of Technology, Lausanne, Switzerland;
- *Israel (ISR1)*, at the Israel Institute of Technology (Technion), Haifa, Israel;
- *Israel (ISR2)*, at the Hebrew University of Jerusalem, Jerusalem, Israel;
- *Korea (KR)*, at Korea Advanced Institute of Science and Technology, Daejon, South Korea;
- *Taiwan (TW)*, at National Taiwan University, Taipei, Taiwan;
- *Australia (AUS)*, at the University of Sydney, Sydney, Australia; and
- *New Zealand (NZ)*, at Victoria University of Wellington, Wellington, New Zealand.

All the hosts run either FreeBSD or Linux or Solaris operating systems.

## 2.2 Server Implementation

At every host we run a server, implemented in Java, optimized with the GCJ compiler. Each server has knowledge of the IP addresses and ports of all the potential servers in the system. Every server keeps an active TCP connection to every other server that it can communicate with. We disable TCP's default waiting before sending small packets (cf. Nagle algorithm, [30, Ch. 19]). The system implements asynchronous I/O using threads. Each server periodically attempts to set up connections with other servers to which it is not currently connected. A `crontab` monitors the status of the server, and restarts it if it is down. Thus, when either a server or communication failure is repaired, connection is promptly reestablished. In case the communication is not transitive, different hosts can have different views of the current set of participants. For gather-all, we present performance results only for periods during which all the hosts had identical perceptions of the set of

connected hosts. In case of host or communication failures, an instance of the algorithm may fail to terminate. This situation can be detected by the failure of a TCP connection or by a timeout.

Each server has code implementing the algorithms in Section 1.1. The server periodically invokes each algorithm: it sleeps for a random period, and then invokes one of the algorithms, in round-robin order. Each invocation of an algorithm is called a session. We use randomness in order to reduce the probability of different sessions running at the same time and delaying each other; this is easier than synchronizing the invocations, as the hosts do not have synchronized clocks.

We constantly run ping (and traceroute in the last two experiments) from each host to each of the other hosts, periodically sending ICMP packets, in order to track the routing dynamics, the latency and loss rate between every pair of hosts in the underlying network. The ping and traceroute processes are also monitored by a `crontab`.

## 2.3 Running Times and Clock Skews

We use two measures of running time:

- The *local running time* of a session at a particular host is the clock time elapsing from when this host begins this session and until the same host terminates the session. Where we present performance measurements, we give local running times at the initiator only.
- The *overall running time* of a session is the time elapsing from when the initiator begins this session until all the hosts terminate this session.

Each host writes to log its starting time and termination time for each session, according to its local clock. Since we do not own the hosts used in our experiments, we were not able to synchronize their clocks. Therefore, in order to deduce the overall running time from the log files, we need to know the skews between different hosts' clocks.

We now explain how we estimate the clock differences. Whenever a host A sends a message to host B, it includes in the message its local clock time. When host B receives the message, it computes the difference between its local clock time and the time in the message, and writes this value to log. Denote this value by $\Delta_{AB}$. Assume that B's clock is $d_{AB}$ time ahead of A's, and assume that the average message latency from A to B and from B to A is $l_{AB}$. Then on average, $\Delta_{AB} = l_{AB} + d_{AB}$ and symmetrically, $\Delta_{BA} = l_{AB} - d_{AB}$. Therefore, $\Delta_{AB} - \Delta_{BA}$ is, on average, $2d_{AB}$. We approximate the clock difference between A and B

as:

$$(average(\Delta_{AB}) - average(\Delta_{BA}))/2$$

This approximation method has some limitations: since messages are exchanged over TCP, the latency can vary substantially in case of message loss. Therefore, if a pair of hosts communicate over a lossy link, this method can give a bad approximation for the clock difference. Furthermore, high variations in the (loss-free) latency between a pair of hosts make it harder to distinguish between high TCP latencies that are due to message loss and those that due to variations (routing delays). However, we can detect (and avoid) some of the cases in which lost messages occur in bursts by setting a threshold. If the round-trip time of a sample exceeds the given threshold, then we can discard that sample when computing the clock difference. From our observation, for pairs of hosts with loss rates less than 10%, most losses are clustered around short periods of time (bursts). This increases the accuracy of the above method in detecting losses. The higher we set the threshold, the less likely we are to discard samples that do not reflect a loss, but message losses are also more likely to go undetected. For each pair of hosts, we set the threshold to be twice the average round-trip time plus four times the standard deviation. Moreover, we discovered that when the average clock skew is computed over a long interval, results can be inconsistent, because some hosts experience clock drifts. So instead of taking the average over all samples, we compute the average over samples obtained in shorter intervals (we used intervals that are one hour long).

We fix a base host $h$, and compute the clock differences between $h$ and every other host per every 15 minute time interval. Then, all logged running times in this interval are adjusted to $h$'s clock, and the overall running time is inferred from the adjusted initiation and termination times. In order to minimize the effect of TCP retransmission delays, it is preferable to choose a host that has reliable links to every other host. In order to check the consistency of our results, we computed the overall running times using three different base hosts: MIT, Emulab, and Cornell. We chose these hosts since the links to them from all hosts were fairly reliable and exhibited a low variation of latency.

Having computed the running times three different ways, we found the results to be fairly consistent. The *distributions* of overall running times as computed with each of the three hosts were almost identical. Moreover, for over 90% of the sessions with overall running times up to 2 seconds, the three computed running times were within 20 ms. of each other.

## 2.4 Running Time Distribution over TCP/IP

We now explain the mathematical model that underlies the analysis of the experimental results in this paper.

After TCP sends a message, it waits for an acknowledgement. If an acknowledgement does not arrive for a designated retransmission time-out, TCP retransmits the message. TCP's initial retransmission time-out is the estimated average RTT on the link plus four times the mean deviation of the RTT, where both the average and the mean deviation are computed over recent values. If the second copy is also lost, TCP waits twice the amount of time it waited before retransmitting the first lost copy, and this continues to grow exponentially with number of lost copies. [30, Ch. 21]

We estimate the distribution of the TCP latency based on the underlying link latency $d$ and loss probability $p$. Assume first that $d$ is half the RTT, that losses are independent, and that the latency does not vary, so the RTT's mean deviation is 0. Then the TCP latency is $d$ with probability $1-p$, $3d$ with probability $p(1-p)$, $7d$ with probability $p^2(1-p)$, and so on. This is a rough estimate, as it does not address variations in latency and loss. Correlated loss causes the first peak (at latency $d$) to occur with higher probability, and causes the tail of the distribution to be sparser; this will be most significant on links with high loss rates. A high variation of latency will shift all the peaks except the first.

We use this estimate to analyze the distribution of the running time of a stage of an algorithm. Let $p_i$ be the probability that the latency of a message sent on link $i$ is at most $D$ (as computed above). Then the probability that an algorithm stage takes at most $D$ time is the product of the probabilities $p_i$ for all the links traversed in this stage. More generally, the running time of a stage is a random variable representing the maximum value of the random variables representing the TCP link latencies, with distributions defined by the RTT and loss rate as explained above. As the number of random variables over which the maximum is computed grows, the expected maximum value increases. This explains why all-to-all, which sends $O(n^2)$ messages in each stage performs much worse than leader, which sends $O(n)$.

## 2.5 Pseudo Code of the Algorithms

Each server runs the following threads:

- A thread for monitoring connections to other hosts (reconnects if necessary).
- One thread per live connection used to exchange messages with the server at the other end of the connection.

- A thread for accepting connections from other hosts.
- A thread for administrating the server.

We now present the server in pseudo code. In defining variables, we use the following data types: `connection` is an IP address and port number identifying a server; `sid` is the type for session identifiers; `time` is for real time values; `alg-type` is either leader, quorum-leader, all-to-all, quorum-to-quorum, secondary leader, or ring.

Messages Exchanged between different servers have the following format:

```
<sid session-id, alg-type alg, connection initiator, connection sender, int stage>
```

where `session-id` is a globally unique session identifier; `alg` is the algorithm being run; `initiator` is the server that initiates this session; `sender` is the server sending the message; and `stage` denotes the communication step, which can be 1, 2, or 3 in case of a leader (or quorum-leader) session, 0, 1, 2, or 3 in case of a secondary leader session, and 1 or 2 in case of an all-to-all or a quorum-to-quorum or a ring session (see Figure 1.1).

**Figure 2.1** Variables used by the algorithms.

```
connection          my-id
connection          my-neighbor
hashtable           neighbor: connection → connection
set of connections  live-connections
hashtable           sent-to: sid → set of connections
hashtable           received-from: sid → set of connections
hashtable           start-time: sid → time
```

We list the variables used by the server in Figure 2.1. The server holds its own IP address and port in a variable `my-id`. Each server is assigned a neighbor that specifies the flow of messages in a ring session. The server holds its neighbor's IP address and port in a variable `my-neighbor`. Each server tracks every other server's neighbor as well as its own in the hash table `neighbor` mapping connections to their neighboring connections. The server keeps track of the set of connections that are up and running in the variable `live-connections`. This variable is maintained by the thread that tracks connections; we do not give the pseudo code for this thread. Thus, in the pseudo code we give below, `live-connections` is a read-only variable. A server $s$ associates each session $x$ with a start time, a set of remote servers that received a session $x$ message from $s$, and a set of remote servers that sent a message to $s$ in session $x$. These are held in the hash tables `start-time`, `sent-to`, and `received-from`, resp.

### 2.5.1 The Gather-All Primitive

In the subsections below we give the pseudo code for the four algorithms that implement the gather-all primitive.

#### All-to-All

The following procedure is used to initiate an all-to-all session:

```
procedure run-all-to-all
    if (live-connections = ∅)
        abort
    endif
    message m = <session-id, all-to-all, my-id, my-id, 1>
    start-time[session-id] ← clock
    sent-to[session-id] ← ∅
    received-from[session-id] ← ∅
    ∀ c ∈ live-connections do
        send m to c
        sent-to[session-id] ← sent-to[session-id] ∪ {c}
    od
```

All-to-all sends two types of messages: stage 1 messages from the initiator to non-initiators; and stage 2 messages from non-initiators to all servers. Both messages are handled using the following event handler:

```
Upon recv m with m.alg-type = all-to-all
    if (start-time[m.session-id] = null)
        start-time[session-id] ← clock
        sent-to[session-id] ← ∅
        received-from[session-id] ← {m.sender}
        ∀ c ∈ live-connections do
                send <m.session-id, m.alg-type, m.initiator, my-id, 2> to c
                sent-to[session-id] ← sent-to[session-id] ∪ {c}
        od
    else
        received-from[session-id] ← received-from[m.session-id] ∪ {m.sender}
    endif
    if ((sent-to[m.session-id] ∩ live-connections) ⊆ received-from[m.session-id])
        end-time ← clock
        write start-time[m.session-id], end-time to log file
    endif
```

### Leader

The following procedure is used to initiate a leader session:

```
procedure run-leader
    if (live-connections = ∅)
        abort
    endif

    message m = <session-id, leader, my-id, my-id, 1>
    start-time[session-id] ← clock
    sent-to[session-id] ← ∅
    received-from[session-id] ← ∅
    ∀ c ∈ live-connections do
        send m to c
        sent-to[session-id] ← sent-to[session-id] ∪ {c}
    od
```

Leader sends three types of messages: stage 1 and stage 3 messages are sent by the initiator and received by non-initiators, and stage 2 messages are sent by non-initiators to the initiator. The following are event handlers for these messages:

```
Upon recv m with m.alg-type = leader and stage = 1
    start-time[m.session-id] = clock
    send <m.session-id, m.alg-type, m.initiator, my-id, 2> to m.initiator

Upon recv m with m.alg-type = leader and stage = 2
    received-from[m.session-id] ← received-from[m.session-id] ∪ {m.sender}
    if (sent-to[m.session-id] ∩ live-connections ⊆ received-from[m.session-id])
        ∀ c ∈ received-from[session-id] ∩ live-connections do
            send <m.session-id, m.alg-type, m.initiator, my-id, 3> to c
        od
        end-time ← clock
        write start-time[m.session-id], end-time to log file
    endif

Upon recv m with m.alg-type = leader and stage = 3
    end-time ← clock
    write start-time[m.session-id], end-time to log file
```

### Secondary Leader

The following procedure is used to initiate a secondary leader session:

```
procedure run-secondary-leader
```

```
    if (live-connections = ∅)
        abort
    endif
    pick a host h from live-connections to be the secondary leader
    message m = <session-id, secondary-leader, my-id, my-id, 0>
    start-time[session-id] ← clock
    send m to h
```

In this algorithm, stage 0 messages are sent from the initiator to the secondary leader; the secondary leader sends a stage 1 message to non-initiators; servers send stage 2 messages to the secondary leader; and the secondary leader sends stage 3 messages to non-initiators. Messages of type 1,2, and 3 are handled exactly the same way as in the leader algorithm. The following is the event handler for messages of type 0:

```
Upon recv m with m.alg-type = secondary-leader and stage = 0
    start-time[m.session-id] ← clock
    sent-to[m.session-id] ← ∅
    received-from[m.session-id] ← ∅
    ∀ c ∈ live-connections - {m.initiator} do
        send <m.session-id, m.alg-type, m.initiator, my-id, 1> to c
        sent-to[session-id] = sent-to[session-id] ∪ {c}
    od
    if (sent-to[m.session-id] = ∅)
        send <m.session-id, m.alg-type, m.initiator, my-id, 3> to m.initiator
        end-time ← clock
        write start-time[m.session-id], end-time to log file
    endif
```

### Ring

The following procedure is used to initiate a ring session:

```
procedure run-ring
    if (live-connections = ∅)
        abort
    endif
    message m = <session-id, ring, my-id, my-id, 1>
    n ← my-neighbor
    while (n ∉ live-connections) do
        n ← neighbor[n]
    od
    start-time[session-id] ← clock
    send m to n
```

In this algorithm, all servers receive stage 1 messages from their neighbors, and all servers except the initiator receive stage 2 messages from their neighbors. The following are event handlers for these messages:

```
Upon recv m with m.alg-type = ring and stage = 1
    if (m.initiator = my-id)
        n ← my-neighbor
        while (n ∉ live-connections) do
            n ← neighbor[n]
        od
        if (n.id  = my-id)
            end-time ← clock
            write start-time[m.session-id], end-time to log file
        else
            m.stage ← 2
            m.sender ← my-id
            send m to n
            end-time ← clock
            write start-time[m.session-id], end-time to log file
        endif
    else
        start-time[session-id] ← clock
        n ← my-neighbor
        while (n ∉ live-connections) do
            n ← neighbor[n]
        od
        m.sender ← my-id
        send m to n
    endif

Upon recv m with m.alg-type = ring and stage = 2
    n ← my-neighbor
    while (n ∉ live-connections) do
        n ← neighbor[n]
    od
    if (n.session-id != initiator)
        m.sender ← my-id
        send m to n
    endif
    end-time ← clock
    write start-time[m.session-id], end-time to log file
```

### 2.5.2 The Gather-Quorum Primitive

In the subsections below we give the pseudo code for the two algorithms that implement the gather-quorum primitive*. The two algorithms presented in this section, quorum-to-quorum and quorum-leader, behave very closely to the all-to-all and leader algorithms presented in Section 2.5.1. There are a some notable differences however. First, both algorithms in this section take an additional parameter, probe-set, which is the set of connections that will be probed in this session. Second, both algorithms in this session terminate as soon as a quorum of hosts hear from a quorum of hosts. Finally, unlike the all-to-all algorithm, in the quorum-to-quorum algorithm, hosts do not begin to send message to other hosts until they have been probed by the initiator. The following code assumes a generic quorum system.

#### Quorum-to-Quorum

The following procedure is used to initiate a quorum-to-quorum session:

```
procedure run-quorum-to-quorum (probe-set)
    if (live-connections ∩ probe-set does not contain a quorum)
        abort
    endif
    message m = <session-id, all-to-all, my-id, my-id, 1>
    start-time[session-id] ← clock
    received-from[session-id] ← {my-id}
    ∀ c ∈ live-connections ∩ probe-set do
        send m to c
    od
```

Quorum-to-quorum sends two types of messages: stage 1 messages from the initiator to non-initiators; and stage 2 messages from non-initiators to all servers. The following is the event handlers for these messages:

```
Upon recv m with m.alg-type = quorum-to-quorum
    if (start-time[m.session-id] = null)
        start-time[session-id] ← clock
        received-from[session-id] ← {my-id}
    endif
    received-from[session-id] ← received-from[session-id] ∪ {m.sender}
    if stage = 1
```

*In order to avoid running too many algorithms, we did not actually run any of the algorithms in this section. Instead, we can extrapolate the running times of these algorithms from the data obtained by running their counter-parts presented in Section 2.5.1. In each of these instances we are only interested in the times when a quorum hears from a quorum and we disregard data that is irrelevant.

```
        ∀ c ∈ live-connections ∩ probe-set do
            send <m.session-id, m.alg-type, m.initiator, my-id, 2> to c
        od
        if (received-from[m.session-id] ∩ probe-set contains a quorum)
            end-time ← clock
            write start-time[m.session-id], end-time to log file
        endif
    endif
```

### Quorum-Leader

The following procedure is used to initiate a quorum-leader session:

```
procedure run-quorum-leader (probe-set)
    if (live-connections ∩ probe-set does not contain a quorum)
        abort
    endif

    message m = <session-id, quorum-leader, my-id, my-id, 1>
    start-time[session-id] ← clock
    received-from[session-id] ← {my-id}
    ∀ c ∈ live-connections ∩ probe-set do
        send m to c
    od
```

Quorum-leader sends three types of messages: stage 1 and stage 3 messages are sent by the initiator and received by non-initiators, and stage 2 messages are sent by non-initiators to the initiator. The following are event handlers for these messages:

```
Upon recv m with m.alg-type = quorum-leader and stage = 1
    start-time[m.session-id] = clock
    send <m.session-id, m.alg-type, m.initiator, my-id, 2> to m.initiator

Upon recv m with m.alg-type = quorum-leader and stage = 2
    received-from[m.session-id] ← received-from[m.session-id] ∪ {m.sender}
    if (probe-set ∩ received-from[m.session-id] contains a quorum)
        ∀ c ∈ received-from[session-id] ∩ live-connections ∩ probe-set do
            send <m.session-id, m.alg-type, m.initiator, my-id, 3> to c
        od
        end-time ← clock
        write start-time[m.session-id], end-time to log file
    endif

Upon recv m with m.alg-type = quorum-leader and stage = 3
    end-time ← clock
```

```
write start-time[m.session-id], end-time to log file
```

# Chapter 3

# The Gather-All Primitive

This chapter presents three experiments, in which we ran the four algorithms that implement the total-gather primitive. A total of ten hosts participated in these experiments. For each experiment, we only present the periods in which every host that participated in that experiment was up and running. While the experiments were running, each host sent a ping probe to every other host once per minute. the results in this chapter were published in [4].

## 3.1 The Effect of Message Loss

This section presents Experiments I, II, each of which lasted three and a half days. Ring was not tested in these experiments. Each of the other three algorithms was initiated by each of the hosts every 7.5 minutes on average, and in total, roughly 650 times. Section 3.2, presents Experiment I, in which the TW host had two links with very high loss rates. We then excluded the TW host, and ran Experiment II, which we present in Section 3.3.

## 3.2 Experiment I

The following hosts participated in this experiment: MIT, CU, NYU, Emulab, UT2, CA1, UCSD, KR and TW. Table 3.1 presents the average RTT and loss rate from every host to every other host during the experiment, as observed by ping. The loss rates from TW to UT2 and CA1 are very high (37% and 42%, respectively), and all the other loss rates are up to 8%. Losses sometimes occur in bursts, where for a period of several minutes all the messages sent on a particular link are lost. The latencies generally vary less, but occasionally we observe periods during which the latency is significantly higher than average.

In this experiment MIT serves as the secondary leader for TW, KR, CU, UT2, NYU, and

**Table 3.1** Network characteristics during Experiment I.

| From | To | KR | TW | MIT | UCSD | CU | NYU | CA1 | UT2 | Emulab |
|---|---|---|---|---|---|---|---|---|---|---|
| KR | Avg. RTT | | 387 | 291 | 272 | 265 | 267 | 168 | 479 | 258 |
| | Loss Rate | — | 6% | 7% | 2% | 0% | 0% | 1% | 1% | 2% |
| TW | Avg. RTT | 388 | | 243 | 177 | 211 | 220 | 221 | 267 | 186 |
| | Loss Rate | 5% | — | 8% | 3% | 3% | 4% | 41% | 37% | 4% |
| MIT | Avg. RTT | 300 | 253 | | 115 | 40 | 34 | 112 | 99 | 80 |
| | Loss Rate | 6% | 8% | — | 5% | 6% | 6% | 6% | 5% | 5% |
| UCSD | Avg. RTT | 289 | 195 | 125 | | 91 | 102 | 42 | 105 | 61 |
| | Loss Rate | 2% | 4% | 5% | — | 0% | 0% | 0% | 0% | 0% |
| CU | Avg. RTT | 266 | 211 | 47 | 73 | | 9 | 88 | 101 | 47 |
| | Loss Rate | 0% | 4% | 5% | 0% | — | 0% | 1% | 0% | 0% |
| NYU | Avg. RTT | 267 | 220 | 39 | 83 | 9 | | 70 | 78 | 56 |
| | Loss Rate | 0% | 4% | 5% | 0% | 0% | — | 0% | 0% | 0% |
| CA1 | Avg. RTT | 168 | 223 | 121 | 32 | 88 | 75 | | 54 | 78 |
| | Loss Rate | 1% | 42% | 5% | 0% | 1% | 0% | — | 0% | 0% |
| UT2 | Avg. RTT | 479 | 266 | 97 | 88 | 100 | 78 | 50 | | 13 |
| | Loss Rate | 1% | 37% | 5% | 0% | 0% | 0% | 0% | — | 3% |
| Emulab | Avg. RTT | 258 | 186 | 76 | 48 | 47 | 57 | 74 | 14 | |
| | Loss Rate | 2% | 4% | 5% | 0% | 0% | 0% | 0% | 3% | — |

Emulab. Emulab is the secondary leader for the rest. We chose secondary leaders that had relatively reliable links to all hosts. We used secondary leaders for all hosts in order to have a meaningful comparison. In practice, secondary leaders would only be used for hosts that have poor links.

Due to occasional loss bursts and TCP's exponential backoff, some running times are very high (several minutes long). Thus, the average running time is not representative. In Table 3.2, we present statistical data about the running times, both overall and local, of the three algorithms. We present the average running time (in milliseconds) taken over runs that complete within 2 seconds. Most runs that experience no more than 2 consecutive losses are included in this average. In Figure 3.1, we present histograms of the distribution of overall running times under 1.3 seconds observed at three of the hosts – MIT which has no lossy links, UT2 which has one lossy link, and TW which has two. The first peak in each histogram represents the overall running time of loss-free runs. The size of the peak illustrates the percentage of the runs of that particular algorithm that were loss-free. The running times over 1 second were sparsely distributed. To illustrate this, we give the percentage of runs that exceed 2, 4, and 6 seconds in Table 3.2.

The overall running time of all-to-all is poor: less than half the runs are under 2 seconds. This is because every instance of all-to-all sends two messages over each lossy link, regardless of the initiator. Thus, most instances experience multiple consecutive losses. Leader has a better overall running time except in TW. This is because each instance of leader initiated at TW traverses each lossy link three times. Instances of leader running from other hosts traverse either one or no lossy links. At the three hosts that have lossy links (TW, UT2,

**Table 3.2** Measured running times, milliseconds, Experiment I.

| Initiator | Algorithm | All-to-all Overall | All-to-all Local | Leader Overall | Leader Local | Secondary Overall | Secondary Local |
|---|---|---|---|---|---|---|---|
| KR | Avg. (runs under 2 sec) | 922 | 550 | 873 | 592 | 695 | 613 |
| | % runs over 2 sec | 55% | 6% | 15% | 8% | 12% | 6% |
| | % runs over 4 sec | 42% | 3% | 9% | 4% | 7% | 3% |
| | % runs over 6 sec | 37% | 3% | 7% | 3% | 5% | 3% |
| TW | Avg. (runs under 2 sec) | 866 | 645 | 1120 | 844 | 679 | 607 |
| | % runs over 2 sec | 54% | 24% | 64% | 43% | 13% | 7% |
| | % runs over 4 sec | 40% | 19% | 43% | 30% | 7% | 4% |
| | % runs over 6 sec | 36% | 18% | 37% | 25% | 6% | 3% |
| MIT | Avg. (runs under 2 sec) | 811 | 295 | 541 | 335 | 585 | 408 |
| | % runs over 2 sec | 55% | 3% | 13% | 6% | 9% | 3% |
| | % runs over 4 sec | 42% | 3% | 8% | 4% | 5% | 2% |
| | % runs over 6 sec | 37% | 3% | 6% | 3% | 4% | 2% |
| UCSD | Avg. (runs under 2 sec) | 860 | 328 | 473 | 332 | 602 | 420 |
| | % runs over 2 sec | 51% | 2% | 6% | 2% | 8% | 3% |
| | % runs over 4 sec | 41% | 2% | 5% | 2% | 5% | 1% |
| | % runs over 6 sec | 35% | 2% | 4% | 2% | 4% | 1% |
| CU | Avg. (runs under 2 sec) | 831 | 320 | 577 | 357 | 578 | 392 |
| | % runs over 2 sec | 53% | 1% | 6% | 1% | 12% | 5% |
| | % runs over 4 sec | 40% | 2% | 4% | 1% | 8% | 4% |
| | % runs over 6 sec | 35% | 2% | 4% | 1% | 6% | 3% |
| NYU | Avg. (runs under 2 sec) | 860 | 319 | 562 | 348 | 598 | 408 |
| | % runs over 2 sec | 54% | 2% | 8% | 3% | 12% | 6% |
| | % runs over 4 sec | 41% | 3% | 6% | 2% | 8% | 3% |
| | % runs over 6 sec | 35% | 2% | 5% | 2% | 6% | 3% |
| CA1 | Avg. (runs under 2 sec) | 850 | 450 | 777 | 553 | 618 | 450 |
| | % runs over 2 sec | 51% | 17% | 30% | 24% | 9% | 3% |
| | % runs over 4 sec | 40% | 13% | 21% | 16% | 6% | 2% |
| | % runs over 6 sec | 35% | 11% | 19% | 15% | 5% | 2% |
| UT2 | Avg. (runs under 2 sec) | 872 | 513 | 1031 | 689 | 636 | 452 |
| | % runs over 2 sec | 52% | 25% | 45% | 36% | 13% | 6% |
| | % runs over 4 sec | 42% | 21% | 34% | 28% | 8% | 4% |
| | % runs over 6 sec | 36% | 17% | 29% | 23% | 6% | 3% |
| Emulab | Avg. (runs under 2 sec) | 844 | 320 | 544 | 356 | 633 | 448 |
| | % runs over 2 sec | 52% | 2% | 8% | 3% | 10% | 5% |
| | % runs over 4 sec | 41% | 2% | 5% | 2% | 6% | 3% |
| | % runs over 6 sec | 37% | 2% | 4% | 2% | 5% | 2% |

and CA1), secondary leader achieves the best overall performance by bypassing the lossy links.

All-to-all has the best local running time at hosts that do not have lossy links. It has a better local running time than leader due to cases in which the triangle inequality does not hold. For example, when UT2 initiates all-to-all, CA1 receives the first message, on average, after 25 ms., and sends a response to all hosts. KR receives this response, on average, after 84 ms., that is, 109 ms. after UT2 sent the first message. This is shorter than the average time it takes UT2's message to get to KR (240 ms.). Therefore, KR engages in all-to-all from UT2 earlier than in leader from UT2. Similarly, when the first stage message to some

host is lost, all-to-all in essence sends it also by a number of alternate paths, one which can prove more effective. This is why the local running time of all-to-all at TW is dramatically better than that of leader.

In the absence of packet loss, the overall running time of leader should be roughly three times the one-way latency on the longest link from the leader, or 1.5 times the RTT. From MIT, the longest link, to KR, has an average RTT of 300 ms. Indeed, the first peak is centered around 400–450 ms. Since all links to MIT other than from TW and KR have significantly shorter latencies (up to 115 ms.), this running time should be experienced whenever there are no losses on the TW and KR links, and at most one or two on each of the other links. Since three messages are sent on each link, and the loss rates of the longest links are 6% and 8%, the probability of no loss occurring on either of the long links is: $.94^3 * .92^3 \approx .65$. Indeed, running times up to 450 ms. occur in 429 out of 659 runs, i.e., 65%.

The longest link from TW is to KR, and its average RTT is 388 ms. Therefore, as expected, the first peak of leader from TW is centered around roughly 1.5 times this RTT, at the 550–600 ms. range. This peak includes only 65 of 643 runs (10%). We now explain why. First, observe that if any of the three messages sent on the link to KR or to UT2 is lost, the running time exceeds the peak. The probability of no loss on the KR link is $.95^3 \approx .86$ and the probability of no loss on the UT2 link is $.63^3 \approx .25$. Next, consider the link to CA1. In the absence of losses, the response from CA1 to TW in the second stage arrives after about 221 ms. (the RTT), and the response from KR to TW arrives after about 388 ms. Once TW sends the final stage message to all hosts, the algorithm terminates at all hosts within half the RTT on the longest link, or roughly 194 ms. If either the first message from TW to CA1 or CA1's response is lost once, then the response arrives roughly after 450 ms., assuming low mean deviation of RTTs. This is sufficiently close to the 388 ms. TW has to wait for KR's message, so it falls in the first peak. However, if the final stage message from TW to CA1 is lost, then CA1 terminates 332 ms. after TW sends the last message, which adds 138 ms. to the overall running time, and pushes it out of the first peak. Two losses on the link to CA1 always push this session away from the peak. The last message to CA1 is not lost with probability 58%. The probability that at most one of the previous messages is lost, and if it is lost, the retransmission is not lost, is: $.58^2 + 2 * .42 * .58^2 \approx .62$. So the probability of the first peak should be $.86 * .25 * .58 * .62 \approx .08$. This is slightly lower than the observed 10%; we hypothesize that this is due to correlated loss, which is significant here due to the high loss rates involved.

The longest link from UT2 is to KR, with an average RTT of 479 ms. Therefore, the peak is around 700–850. We now try to explain why 36% of the runs (230 of 640) are in this range. The probability of having no losses on the KR link is 97%. The link from UT2 to TW is quite erratic. Although the average RTT is 266 ms., the RTT occasionally jumps as

high as 800 ms., and standard deviation of RTTs for the entire experiment period is 139 ms. In periods with low RTT variations, when the mean deviation computed by TCP is low, a run with a single loss to TW in one of the first two stages of the algorithm will fall in the first peak. A loss during a period with a high mean deviation or a loss in the last stage of the algorithm pushes the running time out of the peak. The probability that the last message on this link is not lost is 63%. We hypothesize that the mean deviation is low enough to keep us in the peak approximately half the time. With this assumption, we get that the probability of a loss in one of the first two stages not pushing us out of the peak is 54%, and the probability of the peak should be: $.97 * .63 * .54 \approx .33$, which is close to the observed 36%.

Since TW uses MIT as a secondary leader, we expect secondary leader from TW to behave the same as leader initiated at MIT, with an additional delay of 120 ms. (half the RTT between TW and MIT). Indeed, the first peak is centered around 500–550, and includes roughly the same percentage of the runs as leader at MIT (440/643 = 68%). All-to-all's peak exhibits the lowest overall running time, but the percentage of runs in the first peak is very low, and is the same for all initiators.

Figure 3.2 shows the local running times at the same hosts. The local running time for all-to-all initiated by MIT has a higher peak, as it does not involve any lossy links.

## 3.3 Experiment II: Excluding the Lossiest Host

**Table 3.3** Network characteristics during Experiment II.

| From | To | KR | MIT | Cornell | NYU | CA1 | UT2 | Emulab | UT1 |
|---|---|---|---|---|---|---|---|---|---|
| KR | Avg. RTT | | 294 | 261 | 257 | 165 | 452 | 275 | 500 |
| | Loss Rate | — | 3% | 1% | 3% | 0% | 1% | 3% | 1% |
| MIT | Avg. RTT | 298 | | 43 | 38 | 117 | 117 | 82 | 86 |
| | Loss Rate | 2% | — | 1% | 1% | 1% | 2% | 3% | 2% |
| Cornell | Avg. RTT | 269 | 46 | | 16 | 89 | 101 | 47 | 87 |
| | Loss Rate | 1% | 1% | — | 0% | 1% | 1% | 3% | 1% |
| NYU | Avg. RTT | 257 | 38 | 16 | | 69 | 76 | 60 | 60 |
| | Loss Rate | 3% | 1% | 0% | — | 0% | 0% | 2% | 1% |
| CA1 | Avg. RTT | 165 | 115 | 92 | 75 | | 47 | 79 | 85 |
| | Loss Rate | 0% | 2% | 1% | 0% | — | 1% | 2% | 1% |
| UT2 | Avg. RTT | 454 | 109 | 101 | 77 | 47 | | 14 | 31 |
| | Loss Rate | 1% | 2% | 1% | 0% | 0% | — | 6% | 1% |
| Emulab | Avg. RTT | 275 | 83 | 47 | 60 | 74 | 15 | | 50 |
| | Loss Rate | 4% | 4% | 2% | 2% | 2% | 6% | — | 4% |
| UT1 | Avg. RTT | 503 | 82 | 82 | 60 | 86 | 30 | 52 | |
| | Loss Rate | 1% | 1% | 1% | 1% | 1% | 1% | 5% | — |

We repeated the experiment above without the TW host, which was an end-point on both

lossy links. We also excluded UCSD because it was overloaded at the time of the experiment, and we added UT1. The network characteristics are presented in Table 3.3.

**Table 3.4** Measured overall and local running times, Experiment II.

| Initiator | Algorithm: | All-to-all | | Leader | | Secondary | |
|---|---|---|---|---|---|---|---|
| | | Overall | Local | Overall | Local | Overall | Local |
| KR | Avg. (runs under 2 sec) | 588 | 509 | 758 | 551 | 407 | 388 |
| | % runs over 2 sec | 12% | 7% | 11% | 6% | 9% | 4% |
| MIT | Avg. (runs under 2 sec) | 524 | 278 | 465 | 296 | 442 | 311 |
| | % runs over 2 sec | 11% | 4% | 10% | 5% | 10% | 6% |
| CU | Avg. (runs under 2 sec) | 532 | 277 | 440 | 277 | 471 | 315 |
| | % over 2 sec | 11% | 4% | 9% | 5% | 10% | 5% |
| NYU | Avg. (runs under 2 sec) | 519 | 291 | 449 | 291 | 446 | 296 |
| | % over 2 sec | 12% | 5% | 10% | 5% | 10% | 5% |
| CA1 | Avg. (runs under 2 sec) | 535 | 222 | 378 | 219 | 486 | 367 |
| | % over 2 sec | 11% | 5% | 10% | 5% | 9% | 6% |
| UT2 | Avg. (runs under 2 sec) | 500 | 265 | 866 | 498 | 494 | 383 |
| | % over 2 sec | 10% | 5% | 11% | 6% | 9% | 5% |
| Emulab | Avg. (runs under 2 sec) | 526 | 287 | 506 | 316 | 480 | 338 |
| | % over 2 sec | 12% | 5% | 9% | 6% | 8% | 4% |
| UT1 | Avg. (runs under 2 sec) | 495 | 295 | 982 | 571 | 481 | 367 |
| | % runs over 2 sec | 11% | 4% | 11% | 5% | 10% | 6% |

The running times observed in this experiment are summarized in Table 3.4. In this experiment at least 88% of the runs are under 2 seconds, for all algorithms and all initiators. Even in this setting, all-to-all does not have the best overall running time for any initiator, because even the relatively low loss rates get amplified by the fact that so many messages are sent. Secondary leader works best for most hosts, except for those that are themselves optimal leaders.

When one considers the metric of local running time, we observe that the local running time of all-to-all is always superior to that of leader, regardless of the quality of links. Although they both traverse the same links the same number of times, all-to-all has the advantage that its communication stages may overlap. For example, when the message from the initiator to one of the hosts is delayed due to loss, that host can hear from another host that the algorithm has initiated before receiving the initiator's late message. In the presence of very lossy links, secondary leader outperforms the other two algorithms both locally and globally since it is the only one that avoids the lossy links altogether.

## 3.4 The Impact of Latency

We now present results from Experiment III. In addition to the hosts that participated in Experiment I, the NL host was also included. In this experiment, we evaluated the all-to-all,

**Table 3.5** Network characteristics during Experiment III.

| From | To | KR | TW | MIT | UCSD | CU | NYU | CA1 | UT2 | Emulab | NL |
|---|---|---|---|---|---|---|---|---|---|---|---|
| KR | Avg. RTT | | 643 | 547 | 526 | 587 | 588 | 152 | 446 | 521 | 701 |
| | Loss Rate | — | 9% | 6% | 6% | 4% | 4% | 1% | 3% | 7% | 8% |
| TW | Avg. RTT | 639 | | 235 | 178 | 212 | 222 | 219 | 258 | 187 | 322 |
| | Loss Rate | 10% | — | 4% | 3% | 4% | 3% | 43% | 49% | 4% | 4% |
| MIT | Avg. RTT | 549 | 236 | | 97 | 32 | 28 | 98 | 78 | 71 | 150 |
| | Loss Rate | 8% | 3% | — | 0% | 0% | 0% | 1% | 2% | 1% | 0% |
| UCSD | Avg. RTT | 526 | 179 | 96 | | 73 | 84 | 49 | 91 | 48 | 172 |
| | Loss Rate | 6% | 3% | 0% | — | 0% | 0% | 0% | 2% | 1% | 0% |
| CU | Avg. RTT | 588 | 211 | 32 | 73 | | 9 | 85 | 88 | 47 | 138 |
| | Loss Rate | 4% | 4% | 0% | 0% | — | 0% | 49% | 31% | 1% | 0% |
| NYU | Avg. RTT | 587 | 222 | 28 | 83 | 9 | | 70 | 70 | 57 | 138 |
| | Loss Rate | 4% | 4% | 0% | 0% | 0% | — | 0% | 2% | 1% | 0% |
| CA1 | Avg. RTT | 152 | 219 | 102 | 31 | 94 | 78 | | 54 | 81 | 161 |
| | Loss Rate | 0% | 42% | 1% | 0% | 31% | 0% | — | 2% | 4% | 1% |
| UT2 | Avg. RTT | 446 | 262 | 77 | 91 | 88 | 71 | 50 | | 13 | 154 |
| | Loss Rate | 3% | 48% | 2% | 2% | 31% | 2% | 2% | — | 6% | 2% |
| Emulab | Avg. RTT | 522 | 187 | 70 | 48 | 47 | 57 | 75 | 14 | | 145 |
| | Loss Rate | 8% | 5% | 1% | 1% | 1% | 1% | 4% | 6% | — | 1% |
| NL | Avg. RTT | 697 | 324 | 155 | 175 | 141 | 143 | 165 | 157 | 49 | |
| | Loss Rate | 7% | 3% | 0% | 0% | 1% | 0% | 1% | 2% | 1% | — |

leader, and ring algorithms. All the hosts except UT1 participated in this experiment. Each host ran about 510 sessions of each algorithm. Table 3.5 shows the network characteristics during the experiment. Table 3.6 summarizes the overall and local running times of the three algorithms. Table 3.6 gives the average running time for runs under 3 seconds, and the percentage of runs under 3 seconds. We use a threshold of 3 seconds because link latencies in this experiment are higher than in the previous two. In analyzing the results, we highlight the impact of latency on algorithm performance. In Section 3.5, we discuss the running time of the ring algorithm. In Section 3.6, we show how the highest latency link in the system affects the running time of all-to-all. In Section 3.7, we discuss the impact of a link's latency on the significance of loss on that link. Section 3.8 discusses the fact that the triangle inequality does not hold and the impact this has.

## 3.5 The Running Time of Ring

The message flow in the ring-based algorithm follows the following sequence where each host precedes its neighbor and the first host is the neighbor of the last: NL, Emulab, UT2, CU, NYU, KR, MIT, TW, UCSD, CA1. This above ring was chosen based on latency and loss rate measurements from a previous experiment. The chosen ring is nearly optimal and the loss rates on all the ring links are low.

Ring has the highest average running time in the absence of message loss. However, ring

**Table 3.6** Measured running times, milliseconds, Experiment III.

| Initiator | Algorithm | All-to-all Overall | All-to-all Local | Leader Overall | Leader Local | Ring Overall | Ring Local |
|---|---|---|---|---|---|---|---|
| KR | Avg. (runs under 3 sec) | 1197 | 692 | 1340 | 954 | 1853 | 1158 |
| | % runs over 3 sec | 66% | 9% | 25% | 13% | 18% | 5% |
| TW | Avg. (runs under 3 sec) | 1139 | 809 | 1644 | 1227 | 2014 | 1137 |
| | % runs over 3 sec | 64% | 28% | 84% | 69% | 22% | 4% |
| MIT | Avg. (runs under 3 sec) | 1168 | 515 | 896 | 589 | 1912 | 1117 |
| | % runs over 3 sec | 67% | 3% | 13% | 6% | 18% | 6% |
| UCSD | Avg. (runs under 3 sec) | 1172 | 497 | 833 | 558 | 2040 | 1115 |
| | % runs over 3 sec | 61% | 2% | 14% | 7% | 24% | 6% |
| CU | Avg. (runs under 3 sec) | 1133 | 494 | 1179 | 703 | 2076 | 1120 |
| | % over 3 sec | 58% | 3% | 9% | 2% | 21% | 4% |
| NYU | Avg. (runs under 3 sec) | 1156 | 516 | 1183 | 715 | 2092 | 1134 |
| | % over 3 sec | 62% | 3% | 8% | 3% | 27% | 5% |
| CA1 | Avg. (runs under 3 sec) | 1127 | 563 | 992 | 670 | 2073 | 1141 |
| | % over 3 sec | 66% | 33% | 44% | 37% | 27% | 5% |
| UT2 | Avg. (runs under 3 sec) | 1120 | 558 | 1190 | 637 | 2121 | 1165 |
| | % over 3 sec | 64% | 51% | 60% | 53% | 30% | 8% |
| Emulab | Avg. (runs under 3 sec) | 1108 | 474 | 884 | 594 | 2066 | 1133 |
| | % over 3 sec | 67% | 5% | 15% | 8% | 24% | 5% |
| NL | Avg. (runs under 3 sec) | 1161 | 585 | 1146 | 772 | 2035 | 1143 |
| | % over 3 sec | 65% | 3% | 16% | 7% | 25% | 5% |

has some nice properties: First, the ring algorithm is least affected by message loss. From the network characteristics depicted in Table 3.5, we observe that in the absence of message loss, the total time it takes a message to circulate around the ring twice is about 1900 ms. Unlike leader and all-to-all, the average overall running time for ring appraoches this expectation. The reason for this is that ring sends the fewest messages and uses the most reliable links. Second, the choice of initiator does not have a big impact on the performance of ring, since messages travel over the same links. The only difference between initiating ring from different hosts is that the initiator only receives a message once. This explains why ring sessions initiated at KR have a slightly better overall running time since KR has the longest link. Finally, notice that ring's overall running time is not exactly twice the local running time since the second round is shorter than the first.

## 3.6 Latency Changes over Time

The longest links in the system were between KR and TW and KR and the NL. The latency of these two links varied dramatically in the course of the experiment. We now divide the data gathered in this experiment into two periods. In the first period, the link from KR to the NL had an average RTT of 754 ms., and the link from KR to TW had an average RTT of 683 ms. In the second period, the average RTTs from KR to the NL and to TW dropped to 355 ms. and 385 ms., respectively. So the average one-way message latency on

the longest link dropped by 185 ms. This was the only notable difference between the two periods.

In Figure 3.3, we show histograms of the measured overall running times of all-to-all from all initiators during each of the two periods. The histograms show runs up to 2 seconds; this includes 23% of the runs during the longer latency period, and 60% of the runs during the shorter latency period. We observe that in the period with high latencies, the best running times are around 500 ms. In the period of low latencies, the first peak occurs at 300 ms., or roughly 200 ms. earlier, which is close to the decrease in the one-way latency on the longest link. As we see, the all-to-all algorithm from all initiators is affected by the increase in latency. In contrast, the only instances of the leader algorithm that were affected by this latency change were those initiated at TW, KR, or the NL. Other instances of the leader algorithm were unaffected. For example, the first peak of the leader algorithm initiated at Emulab occurs at 300–350 ms. for both periods.

## 3.7 Latency and Loss

The loss rates from TW to CA1 and UT2 are 43% and 49% respectively. This causes the running times of leader from these hosts to be very high (at least 44% of the runs exceed 3 seconds). The loss rates from CU to CA1 and UT2 are also fairly high (49% and 31% respectively). In spite of this, only 8% of the runs of leader from CU last over 3 seconds. We see that the lossy links from CU do not impact the overall running time as do the lossy links from TW. This is because the latencies of the lossy links from CU are only about one sixth the longest link latency. Therefore, even two consecutive losses on these links do not impact the overall running time.

## 3.8 The Triangle Inequality

The average RTT from UCSD to KR is 526 ms. and the average RTT from UCSD to CA1 is 49 ms., while the average RTT from CA1 to KR is 152 ms. Although UCSD and CA1 are geographically close, the average RTT from UCSD to KR is more than 3 times the average RTT from CA1 to KR. The latency from UCSD to KR can be reduced to less than a half by routing messages indirectly through CA1.

**Figure 3.1** Histograms of overall running times, Experiment I, runs up to 1.3 seconds.

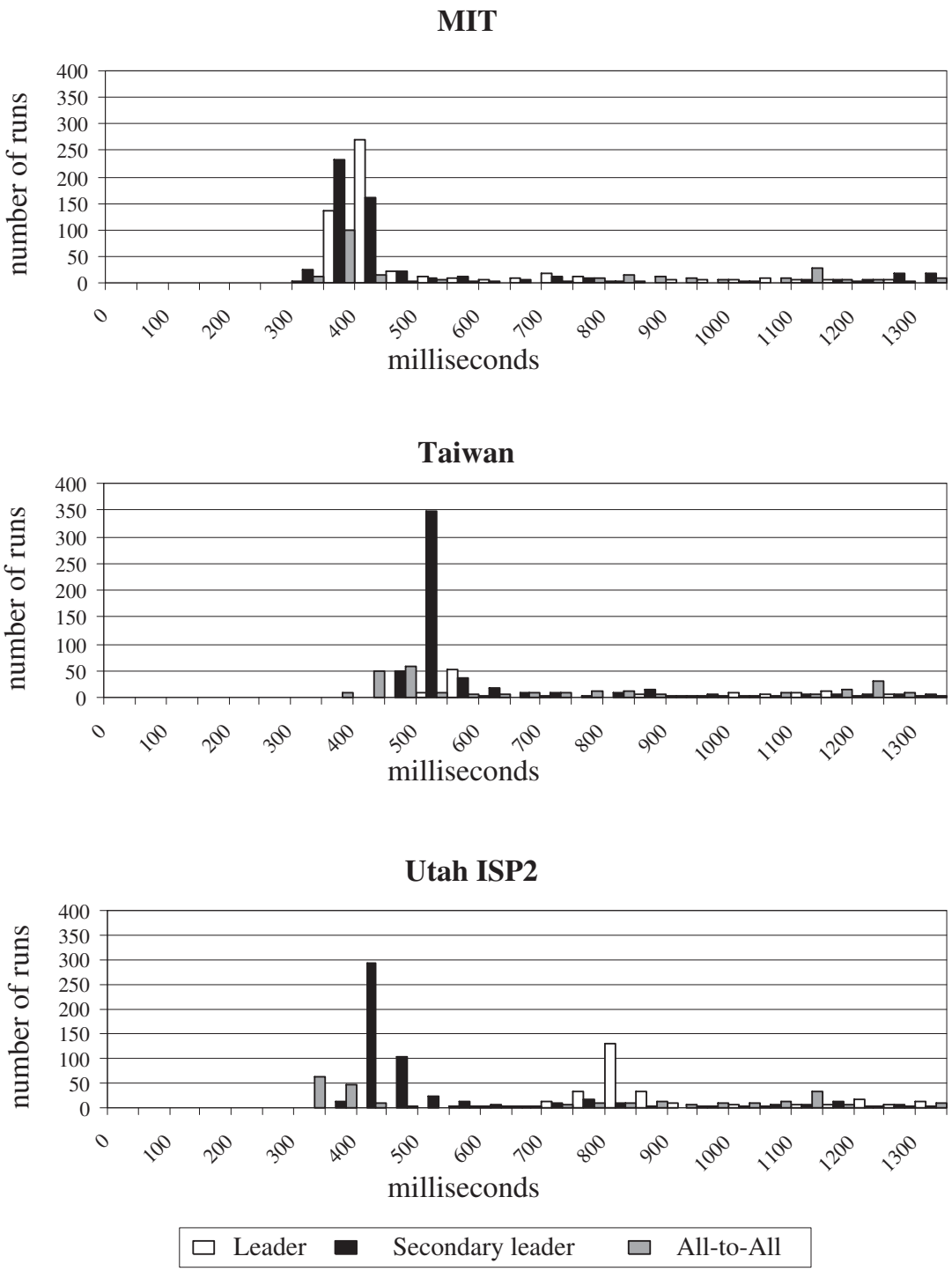


**Figure 3.2** Histograms of local running times, Experiment I, runs up to 1.3 seconds.

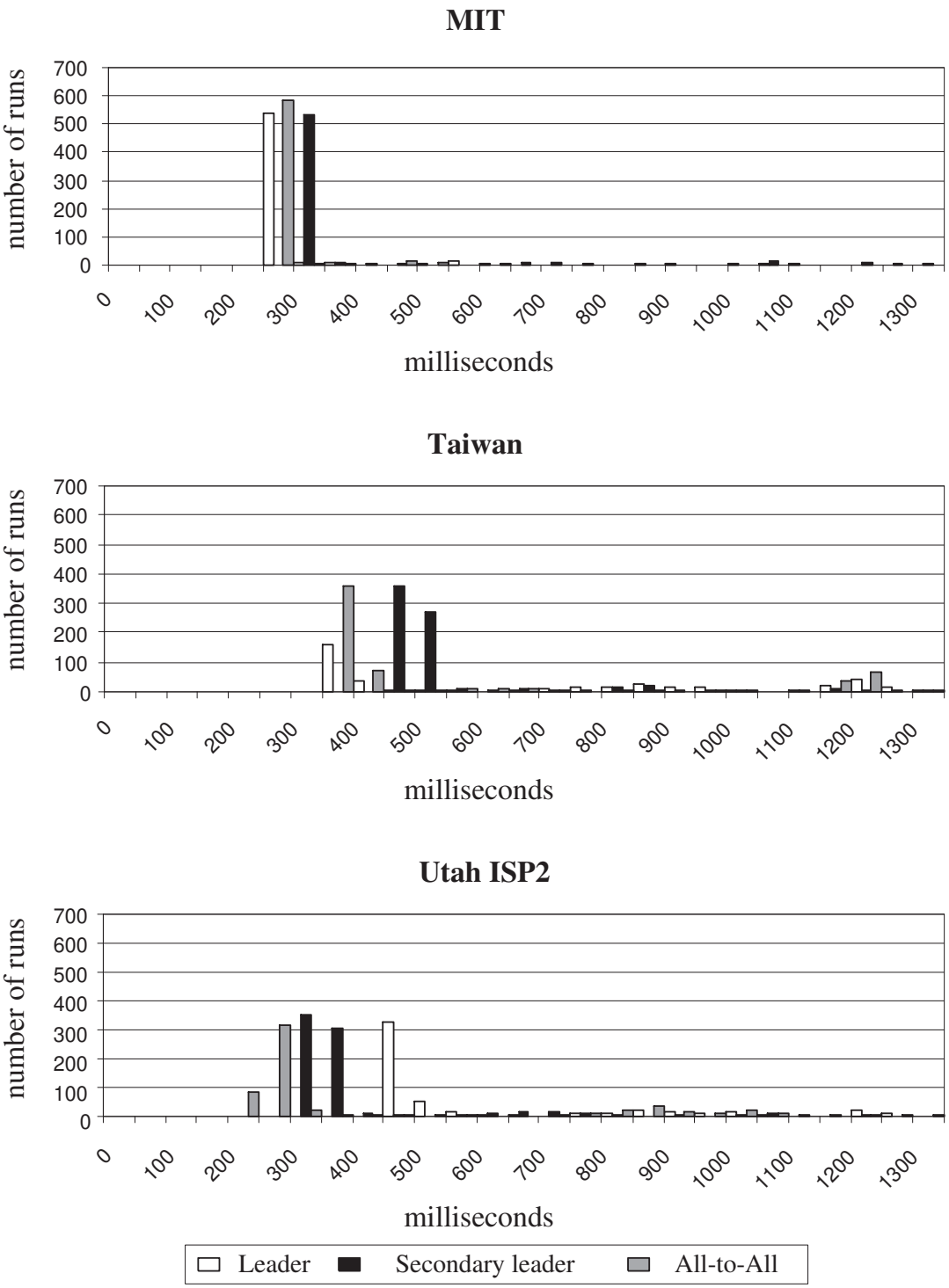
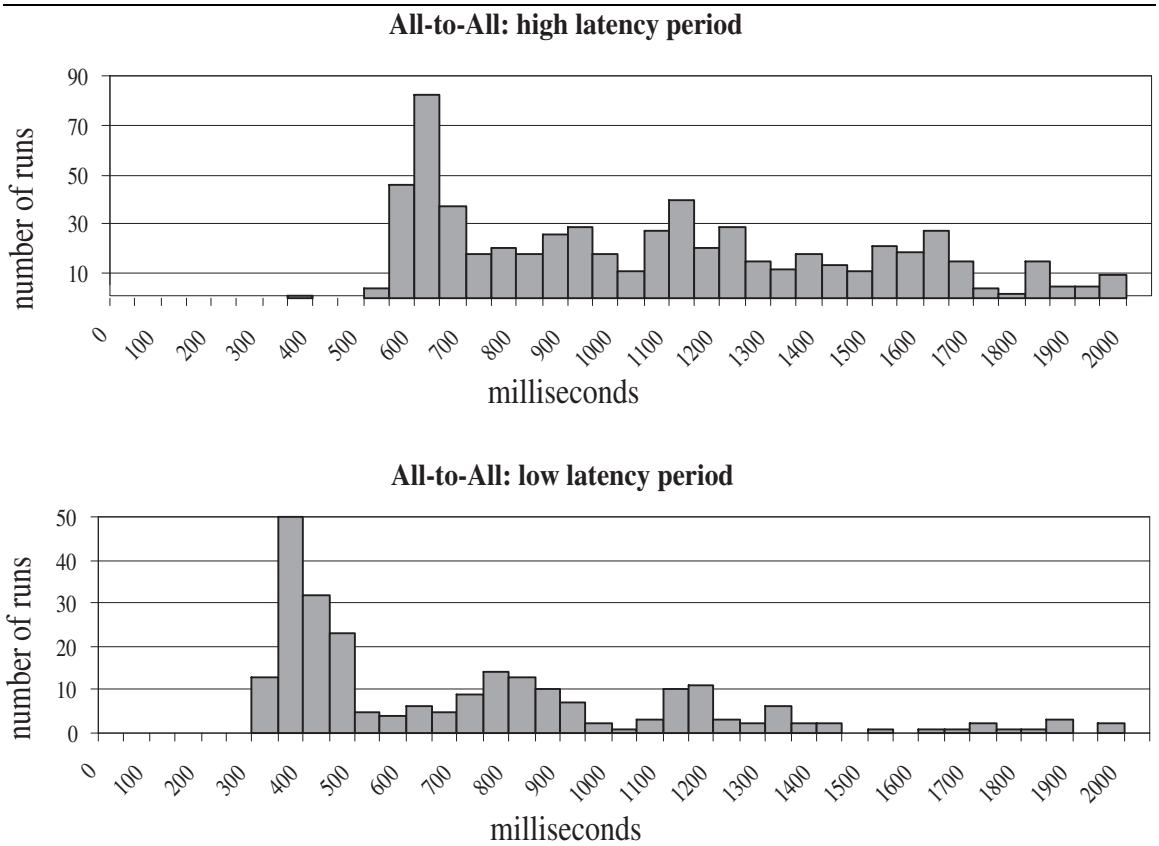


**Figure 3.3** Histograms of overall running times, runs up to 2 seconds, Experiment III.



# Chapter 4

# The Gather-Quorum Primitive

This chapter presents results gathered from two experiments. Experiment IV lasted approximately nine and a half days and included all hosts except CA4 (a total of 27 hosts). Of the hosts that participated only MIT, UT2, CA1, NL, ISR1, AUS, KR and TW were initiators. Each initiator ran leader once every two minutes on average, and in total, roughly 6700 times. Experiment V lasted approximately five and a half days and included only hosts located in North America (a total of 18 hosts), out of which only MIT, CA1, Emulab, CU were initiators. Each initiator ran all-to-all once every two minutes on average, and in total, we accumulated roughly 3700 samples per initiator. The all-to-all algorithm we ran in the latter experiment is a slightly modified version of one that appears in the pseudo code section and used in the previous experiment. Unlike in the previous experiments, hosts do not start sending messages to other hosts in a particular session until they have received a message from the initiator of that session. In both experiments, hosts sent ping probes to each other once every two minutes.

Even though we did not explicitly run algorithms that implement the gather-quorum primitive, we extrapolated the running of these algorithms from the data we accumulated by only looking at the response times for quorums for different probe sets and disregarding irrelevant data. In our analysis analysis of experiment IV, since we only ran the leader algorithm, it was enough to consider the local running times to get a fair comparison. In both experiments, host crashes and network partitions occurred. Table 4.1 shows the table quorum system we used in our evaluation of the results in Experiment IV. We used Table 4.2 as a quorum system in our evaluation of the results in Experiment V.

## 4.1 Comparing the Two Primitives

Since the performance of gather-quorum algorithms depend on the probe set, any comparison with gather all algorithms depends on this parameter. In this section, we consider the

**Table 4.1** Table quorum system in experiment IV.

| MIT | CMU | NYU | Emulab | UCSD | |
|---|---|---|---|---|---|
| NL | SWD | GR | ISR1 | ISR2 | |
| MA1 | MA3 | NY | UT1 | CND | |
| MA2 | AUS | CU | CA1 | UT2 | NC |
| NZ | TW | KR | Swiss | CA2 | CA3 |

**Table 4.2** Table quorum system in experiment V.

| MIT | CU | NYU | CMU | |
|---|---|---|---|---|
| Emulab | UCSD | UT1 | UT2 | |
| MA1 | MA2 | MA3 | NY | NC |
| CND | CA1 | CA2 | CA3 | CA4 |

two extreme cases: minimal probe sets, including exactly one quorum, and complete probe sets, including all hosts. However, independent of the size of the probe set, we can make the following general observations. Gather-quorum algorithms have the advantage that hosts only need to hear from a quorum, (which is usually much smaller than the entire universe of hosts). Therefore, in cases where availability is not an issue, gather-quorum algorithms strictly dominate gather-all algorithms in running time. However, Gather-all algorithms do not fail by definition, since hosts only need to hear from hosts that are currently alive regardless of how many there are. Therefore, in cases of high failure rates (where no quorum exists), gather-all algorithms succeed while gather-quorum algorithms fail.

### 4.1.1 Complete Probe Sets

Even though we had several host failures and network partitions during both experiments, they were not frequent enough to bring down the entire quorum system being used; Tables 4.3 and 4.4 show the percentage of runs that failed for different probe sets. Therefore, for the duration of both experiments, the probability of a host not finding a live quorum was negligible regardless of the quorum system being used (table or majority). Thus, in a gather-quorum algorithm, each host must wait to hear from a set of hosts that is a strict subset of the set of hosts it has to wait to hear from in a gather-all algorithm. This explains the results shown in Figures 4.1 and 4.2, which show a significant gap in the running time for both leader and all-to-all. Figure 4.1 shows the results at six different initiators from Experiment IV. Figure 4.2 shows the results of samples initiated at MIT during Experiment V. The figures show the cumulative distributions of the running time of each primitive.

However, if we try to determine which quorum system is better, the answer is not as clear. If we look at it from a theoretical view point, we find that each system has its advantages

**Figure 4.1** Comparing the gather-all and gather-quorum primitives using the leader based algorithm and complete probe sets (results from experiment IV).

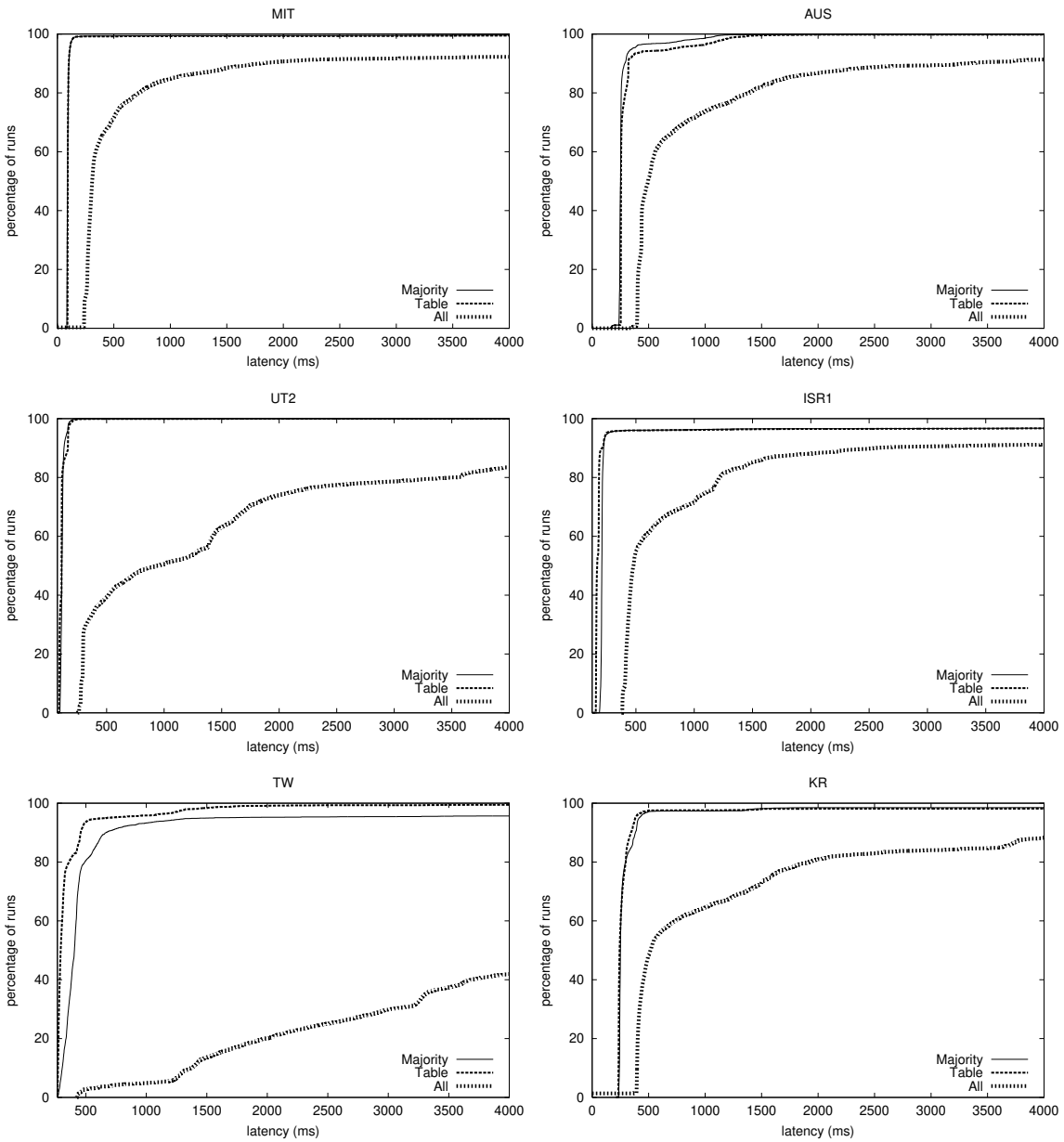


**Figure 4.2** Comparing the gather-all and gather-quorum primitives using the all-to-all algorithm and complete probe sets (results from experiment V).

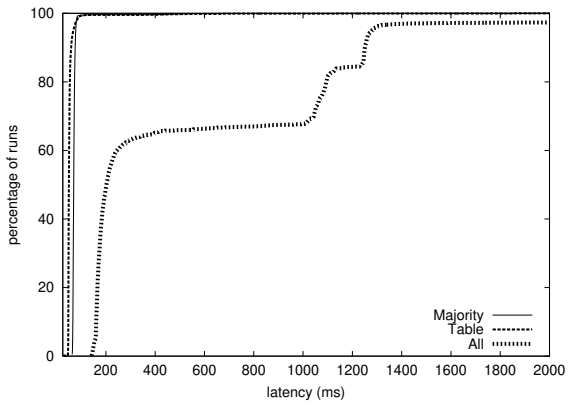



**Table 4.3** Failure percentages for quorum-leader algorithms using the majority quorum system at different initiators for all possible probe set sizes (size is measured in terms the number of hosts). Results compounded during Experiment IV.

| Initiator / Size of Probe Set | MIT | AUS | UT2 | ISR1 | TW | KR |
|---|---|---|---|---|---|---|
| 14 | 8% | 32% | 24% | 15% | 13% | 18% |
| 15 | 2% | 3% | 0% | 4% | 7% | 4% |
| 16 | 0% | 1% | 0% | 3% | 7% | 2% |
| 17 | 0% | 0% | 0% | 3% | 4% | 2% |
| 18 | 0% | 0% | 0% | 3% | 4% | 2% |
| 19 | 0% | 0% | 0% | 3% | 4% | 2% |
| 20 | 0% | 0% | 0% | 3% | 4% | 2% |
| 21 | 0% | 0% | 0% | 3% | 4% | 2% |
| 22 | 0% | 0% | 0% | 3% | 4% | 2% |
| 23 | 0% | 0% | 0% | 3% | 4% | 1% |
| 24 | 0% | 0% | 0% | 3% | 4% | 1% |
| 25 | 0% | 0% | 0% | 3% | 4% | 1% |
| 26 | 0% | 0% | 0% | 3% | 4% | 1% |
| 27 | 0% | 0% | 0% | 3% | 4% | 1% |

and disadvantages. For example, the quorum size in the table-based quorum system (in Experiment IV it ranges from 5-10 hosts) is usually smaller than the majority (14 hosts in this experiment). However, the number of subsets that are quorums is greater in the majority-based quorum system than the in table-based system. With 27 hosts and Table 4.1, we have the following:

$$\text{number of majorities} = \binom{27}{14} = 6,104,700$$

**Table 4.4** Failure percentages for quorum-leader algorithms using the table quorum system at different initiators for all possible probe set sizes (size is measured in terms table rows). Results were compounded during Experiment IV.

| Initiator / Size of Probe Set | MIT | AUS | UT2 | ISR1 | TW | KR |
|---|---|---|---|---|---|---|
| 1 | 1% | 1% | 1% | 4% | 1% | 4% |
| 2 | 0% | 0% | 0% | 3% | 0% | 2% |
| 3 | 0% | 0% | 0% | 3% | 0% | 2% |
| 4 | 0% | 0% | 0% | 3% | 0% | 2% |
| 5 | 0% | 0% | 0% | 3% | 0% | 2% |

$$\text{number of table-quorums} = 1 + 5 + 5^2 + 5^3 + (6)(5)^3 = 906$$

Even though we picked a table that improves the performance, with 27 hosts, there is a significant chance that no table exists which is optimal for every host in the system. Our results from this experiment indicate that for most initiators, table-based quorums outperform majority. For each of these initiators, there exists a subset of the optimal 14-host majority for that initiator that forms a quorum based on Table 4.1. However, for AUS, no such subset exists, which explains why majority provides superior performance in this case.

### 4.1.2 Minimal Probe Sets

With minimal probe sets we have a different story. In this case, every host deals with a particular quorum instead of any quorum. In the majority quorum system, the probe set is composed of the closest majority to the initiator; in the table quorum system, the probe set is composed of the first row of the table (choosing probe sets and quorum systems is discussed in more detail in Section 4.2). Even though this particular quorum is usually chosen because it usually has the best availability, its failure probability is higher than that of the entire quorum system. A quorum fails if any of its hosts fail. Since every host has a nonzero probability of failure, the probability of a quorum failing grows exponentially with the size of that quorum. This means that the probability that a particular quorum fails in the majority quorum system is higher than in the table quorum system. If we look at the graphs in Figures 4.3 and 4.4, it is clear that the running time of algorithms that use the table-based quorum system is by far superior to algorithms using majority and gather-all algorithms. This is the case for two reasons. First, since the minimal probe set in the table quorum system is the first row of the table, the number of hosts involved in the algorithm is very small relative to majority and gather-all algorithms. Second, in a well chosen table, hosts in the first row usually have the highest availability.

If we look at the curves for majority and gather-all algorithms, we find that majority

**Figure 4.3** Comparing the gather-all and gather-quorum primitives using the leader based algorithm and minimal probe sets (results from experiment IV).

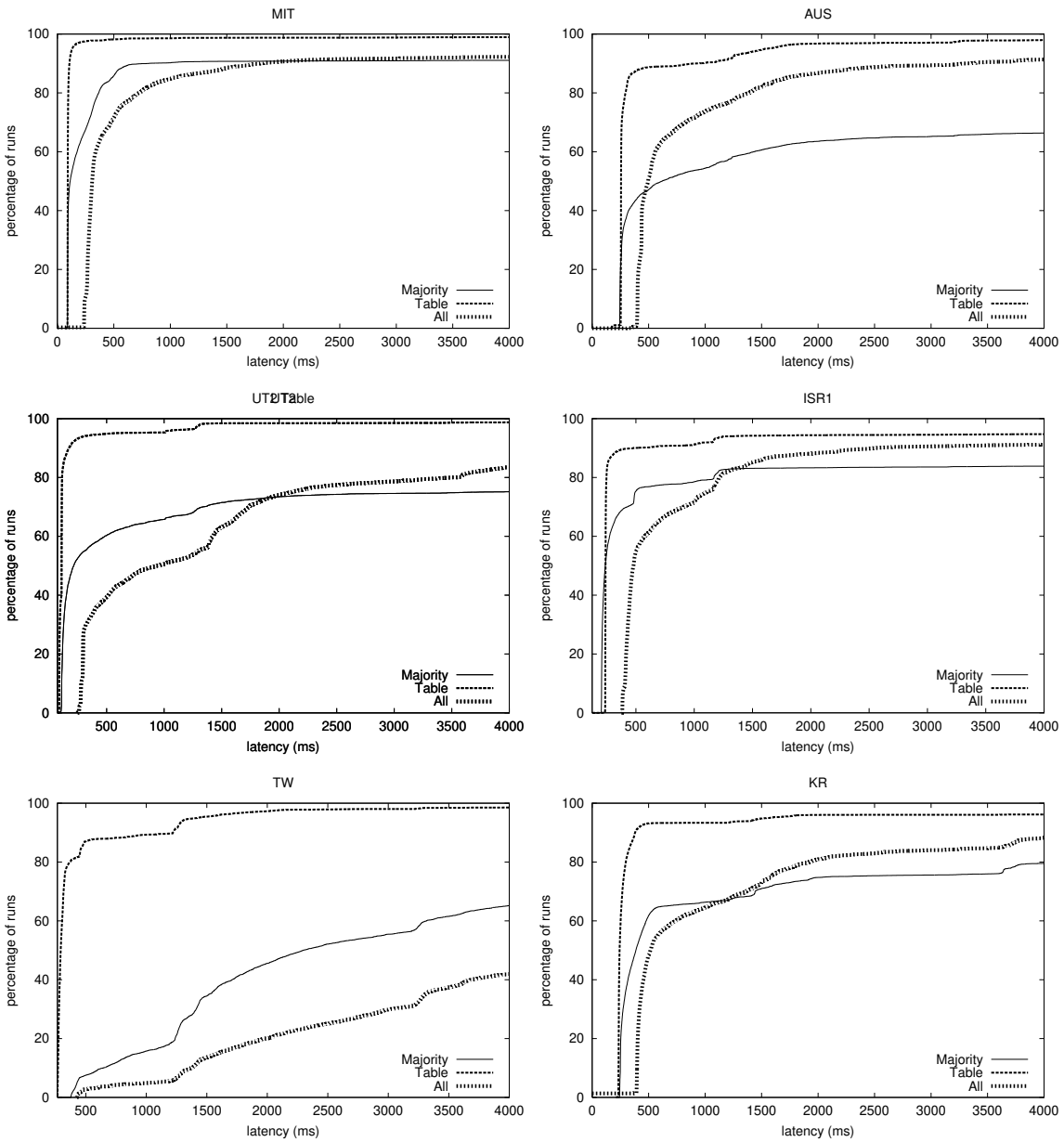
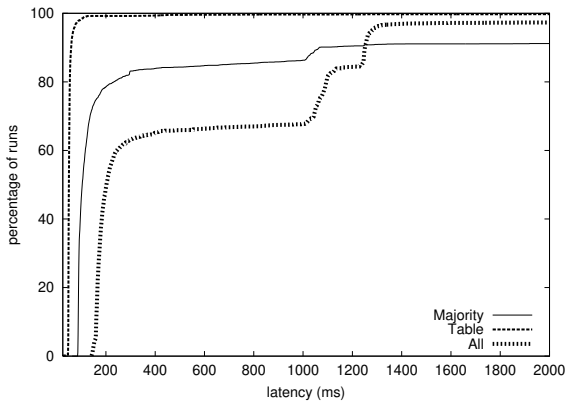


**Figure 4.4** Comparing the gather-all and gather-quorum primitives using the all-to-all algorithm and minimal probe sets (results from experiment V).




dominates in the low latency region and performs worse in the high latency region*. This happens for two reasons. First, since the number of hosts in the minimal probe set in the majority quorum system is approximately half of the total number of hosts, the running time will be lower than with the gather-all algorithm when every host in the probe set is available. Second, since the quorum fails whenever any of its elements fail, a majority becomes unavailable for a significant amount of time during which gather-all algorithms continue to succeed (at higher latencies of course).

## 4.2 The Size of the Probe Set

We now analyze the relationship between the size of the probe set and the running of the quorum-leader and quorum-to-quorum algorithms (using both majority and table quorum systems). In particular, we look at how this relationship is influenced by network dynamics (lost messages, latency variation and failures) and the type of algorithm. The results in presented Section 4.2.1 are from Experiment IV. Section 4.2.2 presents results from Experiment V.

### 4.2.1 Quorum-Leader

#### Majority

Since we have a total of 27 hosts, a majority consists of at least 14 hosts (including the initiator). In this section we look at improvements in the running time as the size of the

*TW is an exception. We will discuss this host in more detail in Section 4.2

**Figure 4.5** The cumulative distribution of local running times of quorum-leader algorithms (using the majority quorum system) initiated at different hosts during Experiment IV, runs up to 4 seconds ($n$ denotes the number of hosts in the probe set).

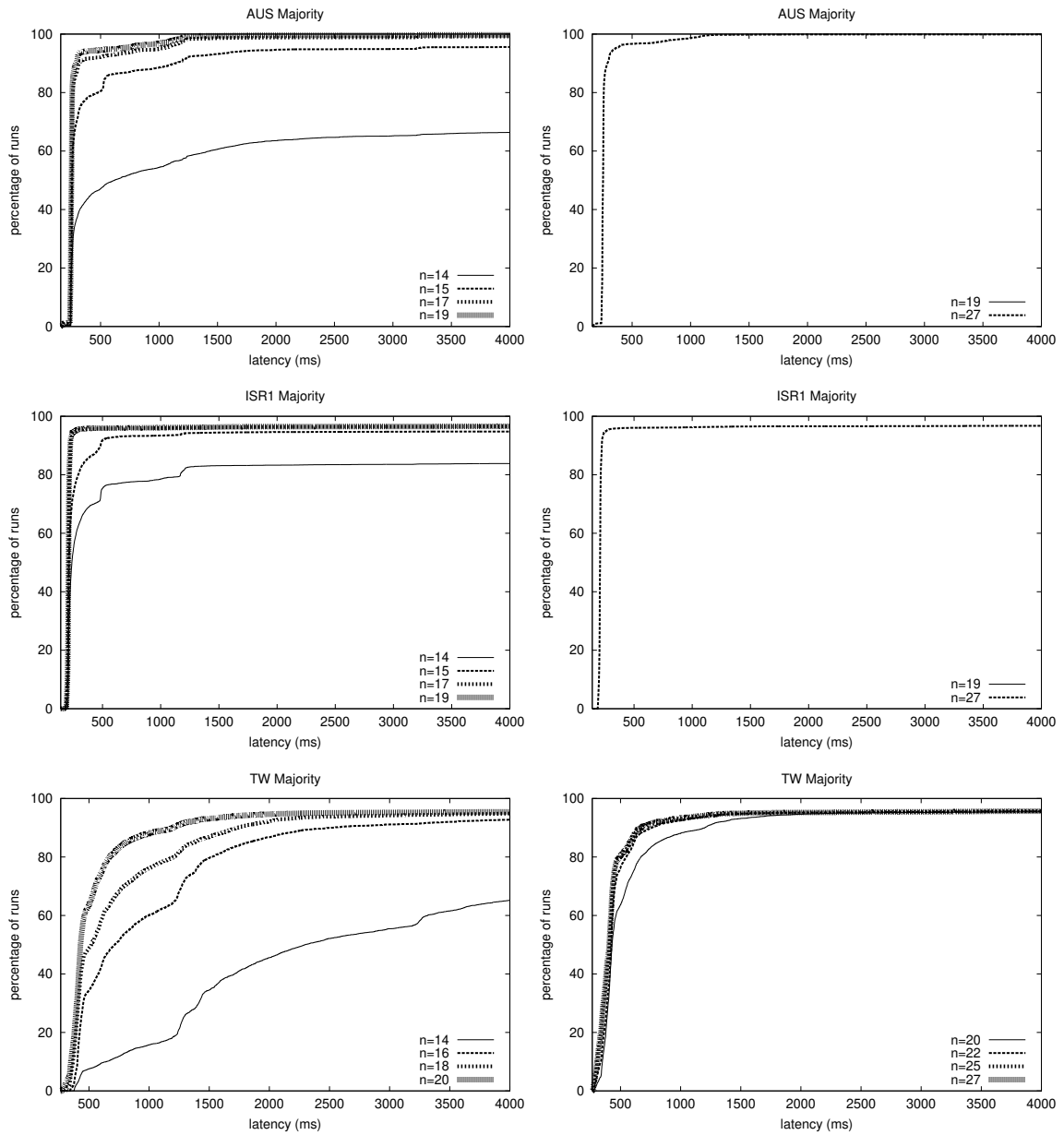


**Table 4.5** Link characteristics from TW to other hosts during experiment IV.

| Host | Loss Rate | Avg. RTT | STD | Min. RTT | TCP Connectivity | % under 1 sec |
|---|---|---|---|---|---|---|
| TW | 0% | 0 | 0 | 0 | 100% | 100% |
| UCSD | 3% | 232 | 25 | *198* | 100% | 97% |
| Emulab | 3% | 238 | 26 | 216 | 100% | 97% |
| NYU | 3% | 273 | 22 | 251 | 100% | 97% |
| MIT | 4% | 303 | 398 | 256 | 100% | 96% |
| CMU | 4% | 289 | 41 | 254 | 100% | 96% |
| CU | 3% | 339 | 127 | 247 | 100% | 97% |
| AUS | — | — | — | — | 99% | 96% |
| NL | 3% | 361 | 23 | 339 | 100% | 96% |
| CA1 | 31% | 482 | 626 | *174* | 95% | 58% |
| NY | 32% | 445 | 853 | 234 | 96% | 63% |
| SWD | 3% | 399 | 59 | 371 | 100% | 96% |
| UT2 | 30% | 743 | 1523 | *171* | 96% | 58% |
| MA2 | 28% | 742 | 1517 | 230 | 94% | 59% |
| NC | 32% | 465 | 616 | 255 | 90% | 63% |
| ISR2 | 3% | 424 | 70 | 400 | 100% | 97% |
| UT1 | 27% | 979 | 1847 | *189* | 96% | 55% |
| MA1 | 29% | 645 | 712 | 238 | 96% | 57% |
| ISR1 | 4% | 551 | 2682 | 400 | 100% | 94% |
| CA2 | 30% | 606 | 1094 | *179* | 69% | 45% |
| GR | 3% | 447 | 29 | 419 | 96% | 93% |
| CND | 35% | 686 | 834 | 212 | 93% | 51% |
| MA3 | 32% | 774 | 1473 | 241 | 96% | 54% |
| NZ | 35% | 636 | 830 | 271 | 91% | 43% |
| KR | 11% | 357 | 163 | 200 | 42% | 40% |
| CA3 | 32% | 1047 | 2091 | *178* | 15% | 10% |
| Swiss | 3% | 384 | 22 | 362 | 15% | 15% |

probe set increases from 14 to 27. For a given initiator, For each instant of leader it initiated, we sort the hosts in ascending order based on the response time for that instant, and assign each host a rank that corresponds to its position in the sorted list.Then average those ranks over all instances. based on those average ranks we sort the hosts in ascending order. Based on that order we rank hosts from 2 to 27 (of course the initiator being 1). This rank represents the order in which we add hosts to the probe set of each initiator. In general there are several arguments to be made for probe sets that are larger than the minimum. First, because of dynamic nature of the Internet (changing routes, lost messages), the 14 hosts closest to the initiator do not stay the same for the whole duration of the experiment. Message loss, in particular, plays a significant role: any lost message from any of the 14 hosts in the minimal probe set, with high probability, increases the running time of the algorithm beyond the RTT of the 15th host. And no matter how reliable links between the initiator and its closest 14 hosts, they still have nonzero probabilities of dropping messages. Second, some hosts also fail during the experiment. However, since failures during the experiment were not very frequent and network partitions were very short, the first factor plays a bigger role in our analysis especially since most failures and network partitions effect all initiators

equally while variations in TCP latencies are different for different hosts.

In general every initiator except TW had low varying and highly reliable links (loss rates of 10% or less) to most hosts. TW on the other hand, had many links with highly variable latencies and loss rates of 25% or more (most of the hosts with bad connections to TW were ISPs in North America). For initiators other than TW, our results indicate that optimal performance is achieved with a probe set that contains 19 hosts. The improvements in performance gained by increasing the size of the probe set beyond 19 hosts are negligible. The highest improvements occur when the size of the probe set is increased to 15 and then 16. The marginal rate of return continues to decrease with the number of hosts and diminishes when this number is increased beyond 19. Figure 4.5 illustrates this observation by showing the cumulative distribution of the running time of runs initiated at AUS and ISR1 for probe sets with different sizes. However, this is not the case with TW. The performance continues to improve significantly as we increase the number of hosts probed by TW to 27. The TW graphs shown in the same figure, show the cumulative distribution of runs initiated at TW for different numbers of hosts. In order to get a better understanding to what is going on with TW, we refer to Table 4.5[†] which shows the link characteristics as measured by "ping" from TW to other hosts in the system (the column labeled "TCP connectivity" refers to the percentage of time the TCP connection was up). We can see that hosts that have loss rates of 25% or more to TW also have the highest average latencies. At first glance, it would appear that the problems of high message loss are compounded by the high latency. However, we see that these hosts have the smallest minimum RTTs (highlighted in the table), which means that the best case involves these hosts. We also notice that the standard deviation is highest for those links, which means that the low latency runs are more probable. The probability of getting good running times increases as we send to more of these hosts.

Now the question remains how well can we estimate the optimal size of the probe set given our knowledge of the network characteristics. For a given to probe set, how accurately can we predict the percentage of runs below a certain threshold based on what we know about the TCP latency distributions? As an example, we will use TW and see how well we can approximate the percentage of runs below 1 second for probe sets the contain 14, 15, 16, and 17 hosts. The last column in Table 4.5 shows the percentage of TCP round trips under 1 second for each link. For simplicity, we will assume that different messages travel through the network independently (not entirely true). We will also restrict our attention to links in which the percentage of TCP RTTs under 1 second is less than 90%.

The probe set of size 14, which contains the first 14 hosts listed in Table 4.5, includes CA1, NY, UT2, MA2. the percentages of TCP RTTs under 1 second from TW to these four are 58%, 63%, 58%, and 59% respectively. Based on the assumptions we have made, we can

[†]AUS is inside a firewall that filters ICMP traffic.

estimate the probability that TW hears from a majority ($Pr_{14}$) as follows:

$$\begin{aligned} Pr_{14} &= .63 * .58 * .58 * .59 \\ &\approx .13 \end{aligned}$$

Indeed the value we measured was .16, which is close to the estimated value. Now how much improvement in the running time can we expect from adding the 15th host (NC) to the probe set? This is the same as the probability of exactly one out of the four lossy hosts failing to make the 1 second threshold and NC succeeding.

$$\begin{aligned} Pr_{15} - Pr_{14} &= .63(.37 * .58 * .58 * .59 \\ &\quad +2 * .63 * .42 * .58 * .59 \\ &\quad +.63 * .58 * .58 * .41) \\ &\approx .21 \end{aligned}$$

The value that we measured was .20. Similarly the improvement we can expect from increasing the size of the probe set from 15 to 16 is the probability of exactly two out of the five lossy hosts failing to make the 1 second threshold and ISR2 succeeding.

$$\begin{aligned} Pr_{16} - Pr_{15} &= .97(2 * .41 * .37 * .63 * .58 * .58 \\ &\quad +2 * .41 * .63 * .63 * .58 * .42 \\ &\quad +.59 * .37 * .37 * .58 * .58 \\ &\quad +.59 * .63 * .63 * .42 * .42 \\ &\quad +4 * .59 * .37 * .63 * .42 * .58) \\ &\approx .33 \end{aligned}$$

The measured value was .26. The improvement we expect we expect from increasing the size of the probe set from 16 to 17 is the probability of exactly three out of the five lossy hosts failing (or exactly two succeeding) to make the 1 second threshold and UT1 succeeding.

$$\begin{aligned} Pr_{17} - Pr_{16} &= .55(2 * .59 * .37 * .63 * .42 * .42 \\ &\quad +2 * .59 * .37 * .37 * .58 * .42 \\ &\quad +.41 * .37 * .37 * .58 * .58 \\ &\quad +.41 * .63 * .63 * .42 * .42 \\ &\quad +.41 * .37 * .63 * .42 * .58) \\ &\approx .12 \end{aligned}$$

The value we measures was .9. The results above suggest that we can predict with a certain

degree of accuracy the probability distribution of the running time for a given probe set.

**Table**

In this section, we analyze the performance based on Table 4.1 which contains 5 rows, with each row containing 5-6 hosts. In this section we look at improvements in the running time of table as the number of table-rows in the probe set increases from 1 to 5. In order to improve performance, we picked the table rows as follows:

- In the first row, we put hosts located at North American universities which were up for the duration of the experiment.
- Our ping traces indicate that hosts located in Europe and Israel are connected to each other by low latency and low loss rate links. Therefore, in order to improve the performance for these hosts, we placed them in the second row (except Swiss which was under firewall restriction for a portion of the experiment).
- In the third row, we put five other hosts in North America that did not crash during the experiment.
- We filled out the last two rows with the remaining hosts.

Every quorum in this setting must include at least one host in the first row. Therefore, while sending to more rows may improve availability in the case of some first row hosts failing, the performance is eventually constrained by the first row. The graphs in Figure 4.6 shows the performance for different initiators. Depending on where the initiators are located, they see different gains at different row numbers. Note especially that difference in performance between probing one table row and probing all rows is smaller than the difference in performance between probing 14 hosts and probing all hosts (in the majority system). This is usually the case since the first row is filled with hosts that were up for most of the experiment and had reliable connections to other hosts.

### 4.2.2 Quorum-to-Quorum

In this section, in order to get meaningful results, we need to look at the overall running times because of the asymmetry of the two phases of the quorum-to-quorum algorithm. For a given probe set, messages travel on the same links regardless of the initiator. Therefore, we only present results from samples initiated by MIT without loss of generality.

**Figure 4.6** The cumulative distribution of local running times of quorum-leader algorithms (using the table quorum system) initiated at different hosts during Experiment IV, runs up to 4 seconds ($n$ denotes the number of table rows in the probe set).

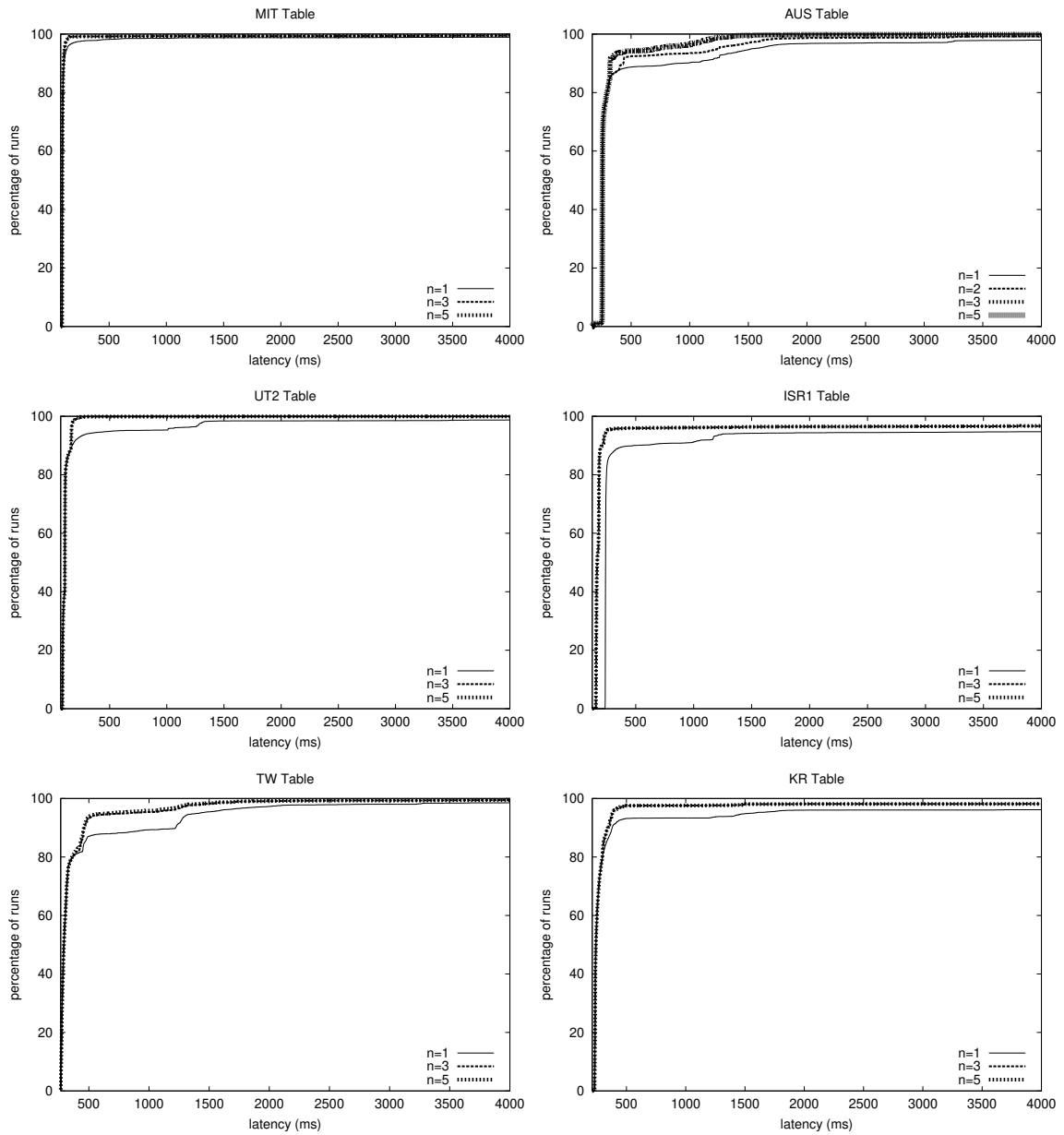


**Figure 4.7** The cumulative distribution of overall running times of majority for all-to-all initiated by MIT during Experiment V, runs up to 2 seconds ($n$ denotes the number of hosts the probe set).

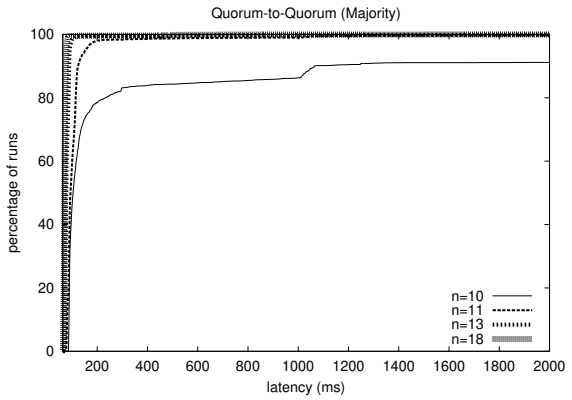
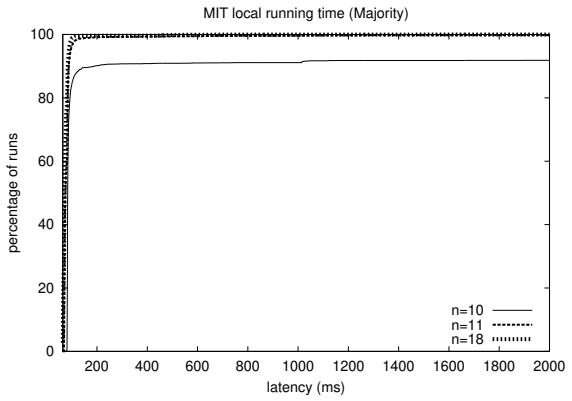



**Figure 4.8** The cumulative distribution of local running times of majority for all-to-all initiated by MIT during Experiment V, runs up to 2 seconds ($n$ denotes the number of hosts in the probe set).




### Majority

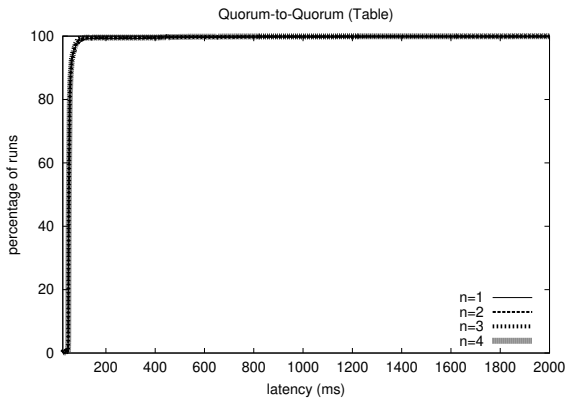
Since we have a total of 18 hosts, a majority consists of at least 10 hosts (including the initiator). In this section we perform the same analysis as Section 4.2.1. We only look at the samples that were initiated by MIT, and rank the hosts (as in Section 4.2.1) with respect to MIT. All the hosts in this experiment are close geographically, and communicate with each other over low-latency and low loss-rate links. In addition, throughout the experiment, only two hosts failed (CA1, UT1) and neither were ranked in the top 10. So we would expect minimal improvements in the overall running time gained by increasing the size of the probe set. However, there is another factor to consider in the quorum-to-quorum algorithm. Since

the algorithm only terminates when a majority of hosts hear from a majority of hosts, the 10-host majority that is optimal for MIT might not be optimal for other hosts, and probing more than 10 hosts increases the probability of other hosts finding their optimal majority. So how much of a role does this play? In order to find out we compare effects of increasing the number of hosts on the overall and local running times. Figure 4.7 shows the cumulative distributions of the overall running time for different majorities. Figure 4.8 shows the cumulative distributions of the local running time for different majorities. From the figures, we observe the following: first, the improvement in performance gained by increasing the size of the probe set from 10 to 11 is "somewhat" greater in the overall running time. Second, sending to 11 hosts is near optimal in the local running time (this is not the case in the overall running time). Since the local running time in this experiment is same as the local running time of the quorum-leader algorithm, the results suggest that optimal running time can be reached with a smaller probe set in the case of quorum-leader than quorum-to-quorum.

**Table**

In this section, we analyzed the performance based on Table 4.2 which contains 4 rows, with each row containing 4-5 hosts. We placed hosts located at universities in the east cost in the first row of the table. In the second row, we put two west cost university hosts and two west coast hosts located at ISPs. We filled the third with the rest of the east cost hosts and put the remaining hosts in the last row. In this particular case, since the hosts in the first row are geographically close to each other and were up for the entire duration of the experiment, the first row was the optimal quorum for every host in the first row. As a result we did not see a significant performance improvement gained by sending to more table rows. Figure 4.9 illustrates these results.

**Figure 4.9** The cumulative distribution of overall running times of table for all-to-all initiated by MIT during Experiment V, runs up to 2 seconds ($n$ denotes the number of table rows the in the probe set).



# Chapter 5

# Conclusions

We measured and analyzed the performance of two primitives and four common information propagation algorithms over the Internet. We explained the distribution of the algorithms' running times in terms of underlying link latencies and loss rates.

One important lesson one can learn from our observations is that loss rates over the Internet are not negligible. Consequently, algorithms that send many messages often have a high running time, even if the messages are sent in parallel in one communication step. More generally, we learn that some communication steps are more costly than others. E.g., it is evident that propagating information from only *one* host to all other hosts is faster than propagating information from *every* host to each of the other hosts.

We suggest to refine the communication step metric as to encompass different kinds of steps. One cost parameter, $\Delta_1$, can be associated with the overall running time of a step that propagates information from all hosts to all hosts*. This step can be implemented using any of the algorithms analyzed in Chapter 3. A different (assumed smaller) cost parameter, $\Delta_2$, can be associated with a step that propagates information from one host to all other hosts. Another cost parameter, $\Delta_3$ can be associated with propagating information from a quorum of the hosts to all the hosts, as measure in Chapter 4.

This more refined metric can then be used to revisit known lower and upper bound results. For example, [14] presents a tight lower bound of two communication steps for failure-free executions of consensus in practical models. Under the more refined metric, the lower bound is $2\Delta_1$, whereas known algorithms (e.g., [6, 16]) achieve running times of $\Delta_2 + \Delta_3$.

*Local running times cannot be composed in this manner.

# References


[1] D. A. Agarwal, L. E. Moser, P. M. Melliar-Smith, and R. K. Budhia. The Totem multiple-ring ordering and topology maintenance protocol. *ACM Transactions on Computer Systems*, 16(2):93–132, May 1998.

[2] Y. Amir and A. Wool. Evaluating quorum systems over the Internet. In *IEEE Fault-Tolerant Computing Symposium (FTCS)*, pages 26–35, June 1996.

[3] D. G. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *SOSP*, pages 131–145. ACM, October 2001.

[4] O. Bakr and I. Keidar. Evaluating the running time of a communication round over the internet. In *ACM Symposium on Principles of Distributed Computing (PODC)*, July 2002.

[5] B. Chandra, M. Dahlin, L. Gao, and A. Nayate. End-to-end WAN service availability. In *Third Usenix Symposium on Internet Technologies and Systems (USITS01)*, March 2001.

[6] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, May 1993.

[8] S. Floyd and V. Paxson. Difficulties in simulating the Internet. *IEEE/ACM Transactions on Networking*, 9(4):392–403, August 2001. An earlier version appeared in Proceedings of the 1997 Winter Simulation Conference, December 1997.

[9] J. N. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*, volume 60, pages 393–481. Springer Verlag, Berlin, 1978.

[10] R. Guerraoui and A. Schiper. The decentralized non-blocking atomic commitment protocol. In *IEEE International Symposium on Parallel and Distributed Processing (SPDP)*, October 1995.

[11] Katherine Guo, Werner Vogels, and Robbert van Renesse. Structured virtual synchrony: Exploring the bounds of virtual synchronous group communication. In *7th ACM SIGOPS European Workshop*, pages 213–217, September 1996.

[12] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 68–76, May 1996.

[13] I. Keidar and D. Dolev. Increasing the resilience of distributed and replicated database systems. *Journal of Computer and System Sciences special issue with selected papers from ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS) 1995*, 57(3):309–324, December 1998.

[14] I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults – a tutorial. Technical Report MIT-LCS-TR-821, MIT Laboratory for Computer Science, May 2001. Preliminary version in SIGACT News 32(2), pages 45–63, June 2001 (published May 15th 2001).

[15] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. Moshe: A group membership service for WANs. *ACM Transactions on Computer Systems*, 20(3):1–48, August 2002. Previous version in the 20th International Conference on Distributed Computing Systems (ICDCS) pp. 356–365, April 2002.

[16] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 78.

[18] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[19] D. Malkhi and M. K. Reiter. An architecture for survivable coordination in large-scale systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):87–202, March/April 2000.

[20] J-P. Martin, L. Alvisi, and M. Dahlin. Minimal byzantine storage. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC 2002)*, October 2002.

[21] M. Naor and A. Wool. The load, capacity, and availability of quorum systems. *Information Processing Letters*, 27(2):423–447, 1998.

[22] V. Paxson. End-to-end Internet packet dynamics. In *ACM SIGCOMM*, September 1997.

[23] D. Peleg and A. Wool. Availability of quorum systems. *Inform. Comput.*, 123(2):210–223, 1995.

[24] S. Rajsbaum and M. Sidi. On the performance of synchronized programs in distributed networks with random processing times and transmission delays. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):939–950, 1994.

[25] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zahorjan. Detour: a case for informed Internet routing and transport. *IEEE Micro*, 19(1):50–59, January 1999.

[26] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The end-to-end effects of Internet path selection. In *ACM SIGCOMM*, pages 289–299, September 1999.

[27] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[28] N. Sergent. Evaluating latency of distributed algorithms using Petri nets. In *5th Euromicro Workshop on Parallel and Distributed Processing*, pages 437–442, London, UK, January 1997.

[29] D. Skeen. Nonblocking commit protocols. In *ACM SIGMOD International Symposium on Management of Data*, pages 133–142, 1981.

[30] Richard Stevens. *TCP/IP Illustrated*, volume 1. Addison-Wesley, 1994.

[31] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 224–237, 1997.

[32] P. Urbán, X. Défago, and A. Schiper. Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms. In *9th IEEE International Conference on Computer Communications and Networks (IC3N 2000)*, October 2000.

[33] Y. Zhang, N. Duffield, V. Paxson, and S. Shenker. On the constancy of Internet path properties. In *ACM SIGCOMM Internet Measurement Workshop*, November 2001.