Why Multicast Protocols (Don't) Scale: An Analysis of Multipoint Algorithms for Scalable Group Communication

Thesis by

Eve M. Schooler

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

Also published as Caltech Computer Science Technical Report caltechCSTR/2001.003

California Institute of Technology
Pasadena, California

2001
(Defended September 19, 2000)

Acknowledgments

First and foremost, I would like to thank my committee members for their participation and feedback: Jehoshua (Shuki) Bruck, K. Mani Chandy, Deborah Estrin, Jason Hickey, and Alain Martin. I also would like to thank several other individuals who read my thesis and provided valuable comments: Bob Felderman, Joe Kiniry, Rajit Manohar, and Ruth Sivilotti. I extend special thanks to my committee chair and thesis advisor K. Mani Chandy, who has provided generous support and friendship throughout my stay at Caltech. I am also grateful to him for providing the freedom to work on whatever research presented itself as interesting. I consider myself lucky to have been part of his research group, where I came into contact with many gifted researchers: Michel Charpentier, Roman Ginis, Peter Hofstee, Joe Kiniry, K. Rustan Leino, Berna Massingill, Adam Rifkin, Paul Sivilotti, John Thornley, and Dan Zimmerman. I thank each of these individuals not only for contributing to my education, but also for enriching my experience at Caltech. Where else could one resort to the humor in using "the magic 8 ball" to make admissions or coding decisions? I have also been blessed to collaborate with Rajit Manohar, whose advice, guidance and insights have been invaluable. It is a rare friend who is willing to critique half-formed ideas.

There have been many others who keep this department running and whose help has always been greatly appreciated: Cynthia Brady, Jeri Chittum, Betta Dawson, Cindy Ferrini, Louise Foucher, Diane Goodfellow, Cici Koenig, Yvonne Recendez, and Gail Stowers. To the army of people who have maintained the systems on which I rely, I bow down to you: Dave Felt, Roman Ginis, Joe Kiniry, Dave LeBlanc, Chris Lee, Rajit Manohar, Mika Nyström, and Dan Zimmerman. To my friends at Myricom and Cornell, where most of my simulations were run, I could not have finished without the use of your fast machines (may it be a long time until I have to port the network simulator to yet another OS). I am deeply grateful. I am also thankful for the breadth of knowledge of my friends at the Fairchild Engineering Library: Kimberly Douglas and Hema Ramachandran. The resources they have assembled and have made available for on-line researchers have made my life immensely more productive.

There have been many others within the Computer Science Department who have provided both comraderie and intellectual companionship: Cindy Ball, Al Barr, Eric Bax, Cynthia Brady, David Breen, Mathieu Desbrun, Boris Dimitrov, Ilja Friedel, Eitan Grinspun, Rohit Khare, Cici

Koenig, David Laidlaw, Alain Martin, Daniel Maskit, Rajit Manohar, Mark Meyer, Mika Nyström, Marc Reiffel, Steve Taylor, Jerrell Watts, and Zöe Wood. I thank them for making this journey a memorable one. Of course, I am also especially grateful to certain unnamed individuals for never making me feel too "old."

I owe a debt of gratitude to my office mates: Rajit Manohar for sharing his exquisite taste in chocolate and music, Zöe Wood whose artistry and spirit I prize, and Eitan Grinspun for his willingness to launch into song whenever the mood struck us!

I thank Ed Perry and his group at HP Labs Broadband Systems Lab for a rewarding summer internship in 1996. I extend heartfelt thanks to my friends and colleagues at Microsoft's Bay Area Research Center: Jim Gemmell and Jim Gray, as well as a host of other researchers who were there during my internship during the Summer of 1997. It was my honor to work with so accomplished a group of individuals, who were willing to entertain the notion of a telecommuting internship! I have thoroughly enjoyed our collaborations.

There are many individuals from the Internet Engineering Task Force (IETF) community who I thank for providing an extra backdrop against which to do research in Networks: Mark Handley, Ruth Lang, Jöerg Ott, Allison Mankin, Scott Shenker, Abel Weinrib, and many others from the Multiparty Multimedia Session Control (MMusic) working group. In addition, I extend a huge thank you to Steve Coya for making it possible for me to attend IETF meetings on a student budget.

I thank the outstanding community of people who comprise the High Speed Networking Division of USC's Information Science Institute (ISI), where I was employed prior to Caltech and where I continued to work part-time during my first two years in graduate school. Several colleagues and friends from ISI deserve special thanks for having encouraged me over the years and for serving as constant role models: Celeste Anderson, Yigal Arens, Bob Braden, Steve Casner, Danny Cohen, Deborah Estrin, Mike Gorman, Mary Hall, and the late Jon Postel.

Other individuals whose encouragement and friendship I have valued immensely, and whose advice about staying in school I probably should have heeded back when I was completing my Masters at UCLA: Thelma and Jerry Estrin, Len and Stella Kleinrock, and Verra Morgan.

Despite appearances, there are women at Caltech! I have been fortunate to have met many phenomenal women scientists and other Caltech affiliates who have been supportive throughout: Cindy Ball, Andrea Belz, Melanie Bennett Brewer, Zehra Cataltepe, Min Chen, Christina Cohen, Roian Egnor, Carmit Eliyahu, Caroline Fohlin, Jen Linden, Berna Massingill, Helen Parker, Susan Pelletier, Cathy Wong, and Zöe Wood.

Friends I cherish for their wisdom and perspective and without whom I could not have survived: Ruth Ballenger, Suzanne Biegel, Nan Boden, Christina Cohen, Herb Donaldson, Hyewon Hyun, Rajit Manohar, Leanne Lung Nemeth, Ruth Sivilotti, Beverly Stein, and Sara Tucker.

I also wish to thank my friends who have looked after me in times of need: Dr. Lynda Roman and Nurse Charlotte Haravey of the USC Norris Cancer Center, and Nurses Divina Bautista and Alice Sogomonian at the Caltech Student Health Center.

Most importantly, I thank my family: my sisters, who have helped me to balance life and work and to understand what is truly important; my father whose intellect and curiosity know no bounds and whose fascination with invention has always inspired me; my mother who taught me my first algorithms when she taught me to read music (the for-loop of musical repeats, the go-to of skipping to a coda), who has always served as the model for the person I strive to be, and whose ability to create beauty wherever she goes awes me still.

I treasure my son Sean, who has good-naturedly marked the passage of time while I have been at Caltech! He certainly deserves his own doctorate in something: Zoology, for he can navigate the Caltech campus to find and identify most resident wildlife; Patience, for enduring my demanding schedule this past year so I could finish "the book I have been writing"; Frogology, the art of catching frogs from ponds on the Caltech campus? In addition, I could not have completed my degree were it not for Leanne Lung Nemeth, Cathy Saewert, and the wonderful teachers at the Childrens Center at Caltech, who have allowed me to do my research knowing that Sean was in good hands when he was in their care.

Above all, I am indebted to Bob Felderman, the most exceptional husband and father I know. I am grateful to him for his ceaseless encouragement, as well as his sense of humor. I thank him for happily providing advice and feedback (when solicited!), and for having the ingenuity to ask just the right questions when I was trying to debug my simulations, work out a problem, or turn a phrase. To say Bob has been infinitely supportive – not only during the normal course of everyday grad student life, but also during a particularly difficult year of health challenges – would be an understatement. We will always joke that Caltech considers spouses "dependents," for we know who has depended on whom! For the many sacrifices he has made, and for his generosity of spirit, I am eternally grateful. It is to him that I dedicate this thesis.

The research described in this thesis was funded in part by an Earl C. Anthony Graduate Fellowship, a Career Development Grant from the American Association of University Women, a Microsoft Graduate Fellowship, as well as the Air Force Office of Scientific Research and the National Science Foundation. I thank all of them for their generous support.

Abstract

With the exponential growth of the Internet, there is a critical need to design efficient, scalable and robust protocols to support the network infrastructure. A new class of protocols has emerged to address these challenges, and these protocols rely on a few key techniques, or micro-algorithms, to achieve scalability. By scalability, we mean the ability of groups of communicating processes to grow very large in size. We study the behavior of several of these fundamental techniques that appear in many deployed and emerging Internet standards: Suppression, Announce-Listen, and Leader Election.

These algorithms are based on the principle of efficient multipoint communication, often in combination with periodic messaging. We assume a loosely-coupled communication model, where acknowledged messaging among groups of processes is not required. Thus, processes infer information from the periodic receipt or loss of messages from other processes.

We present an analysis, validated by simulation, of the performance tradeoffs of each of these techniques. Toward this end, we derive a series of performance metrics that help us to evaluate these algorithms under lossy conditions: expected response time, network usage, memory overhead, consistency attainable, and convergence time. In addition, we study the impact of both correlated and uncorrelated loss on groups of communicating processes.

As a result, this thesis provides insights into the scalability of multicast protocols that rely upon these techniques. We provide a systematic framework for calibrating as well as predicting protocol behavior over a range of operating conditions. In the process, we establish a general methodology for the analysis of these and other scalability techniques. Finally, we explore a theory of composition; if we understand the behavior of these micro-algorithms, then we can bound analytically the performance of the more complex algorithms that rely upon them.

Contents

List of Figures

List of Tables

List of Programs

Chapter 1

Introduction

1.1 Motivation

The last decade has witnessed the emergence and ubiquity of the Internet. This phenomenon has given rise to an ever-growing level of connectivity. The stress that exponential growth places on the network fabric can be observed in the expectations of users; namely, the ability to connect to the Internet at anytime and from anywhere, the constant availability of machines from which data is needed, the existence of bandwidth to carry data in both a reliable and timely manner, and the resilience of connections to faulty machines and links. Thus, the network is in dire need of solutions to support these demands. This thesis examines issues in the design of efficient, scalable network algorithms that are highly robust.

Much of the networking infrastructure for the Internet was designed under the assumption that machines communicate in a point-to-point fashion, i.e., communication occurs between pairs of machines. A sender addresses a message to a specific receiver and typically awaits explicit feedback to determine whether or not the information sent was properly received. Consequently many of the core Internet protocols were designed with a model that assumes direct and reliable exchanges between a single sender and a single receiver.

There are two aspects about this premise that have changed over time with regards to scalable group communication, by which we mean the ability for groups of communicating processes to grow very large in size. First, point-to-point messaging has given way to multipoint messaging. Various forms of efficient multipoint communication have evolved for use not only within the local area network (LAN), but also within the wide area network (WAN), facilitating multiway exchanges between groups of senders and receivers. Second, acknowledged messaging has been replaced by unacknowledged messaging. Depending on the application, messages sent to large groups of receivers may not require acknowledgment by the recipients. For example, an interactive conferencing application that distributes real-time audio does not retransmit missed speech segments. By the time the data would be retransmitted, the conference is well beyond the point where the audio is useful.

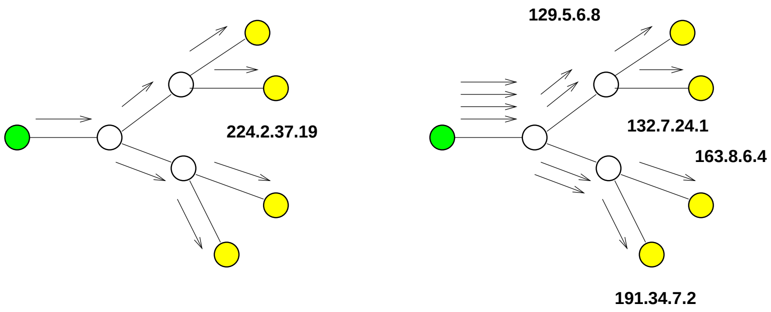

Figure 1.1: **Multicast vs. Unicast: Efficiency and Group Addressing.**

In this thesis, we examine the use of networking substrates that provide an efficient abstraction for multipoint communication, such as IP Multicast [20] as displayed in Figure 1.1. We use the terms multipoint and multicast interchangeably throughout this thesis. However, it is critical to note that our results, while assuming multipoint substrates, are independent of any specific implementation, and in particular independent of the choice of IP Multicast as the underlying implementation.

Multipoint substrates are interesting for at least two reasons. They provide an implementation of efficient broadcast. Copies of a message are made by routers at the branching points in the network, as the message is forwarded to a group of processes, members of which reside along different paths. Therefore, only one copy of the message is ever present on any link in the network. This is in stark contrast to replication at the source; an endpoint keeps track of the individual addresses for group members and disseminates an identical message to every participating process by initiating separate point-to-point interactions on a pair-wise basis. Multipoint substrates also provide the powerful abstraction of a group address. A process interested in communicating with all other group members simply uses the group address. A message sent to the group address is sent to all processes subscribed to the group at that point in time. Therefore multicast provides an efficient and thus scalable means to transmit data to groups of processes. Figure 1.1 shows these benefits by displaying the key differences between multicast and unicast communication: improved link usage and group addressing.

While multicast communication is inherently more scalable for groups of processes than its point-to-point counterpart, it still faces scalability challenges. For example, the traditional mechanism used to support reliable message passing in an unreliable network is for receivers to use acknowledgments (ACKs) to indicate when messages have been successfully received. If this mechanism is used to implement reliable group communication, then a sender must receive an ACK from each receiver participating in the group for each message sent. The number of ACKs grows with the number of receivers, potentially overwhelming the sender and thereby preventing scalability. This leads to the classic problem known as message implosion, not only at the sender but possibly at routers on the path back to sender (Figure 1.2).

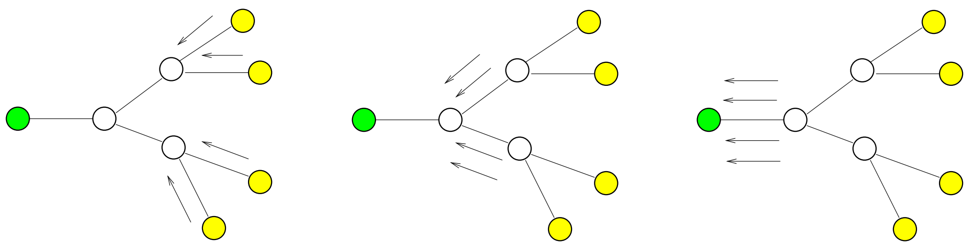

Figure 1.2: **Message Implosion Toward the Sender.**

Alternatively, receivers may send negative acknowledgments (NACKs) to indicate when messages have been lost. A NACK is usually triggered by the detection of a skipped sequence number. Although NACKs reduce the overall numbers of messages sent back to the sender, they do not avoid message implosion, which might occur when a whole branch of the multicast distribution tree loses the same message.

The protocols we examine in this thesis attempt to avoid message implosion by using completely unacknowledged messaging, which we refer to as announcements, for group communication. Messages are announced to a set of processes that passively wait to receive them. Of course, announcements in and of themselves are a form of unreliable communication. However, when an announcement is resent periodically ad infinitum, and the network is not permanently disconnected, the probability that the message reaches a receiver and eventually all receivers approaches 1.

Furthermore, when announcements are made periodically, each message serves to renew or to update local state information from the announcers. Such an approach is highly robust in the face of network faults, late process arrivals and process departures. Because incorrect state will be corrected by subsequent announcements, there is no need to perform any special sequence of events when errors occur. Thus, the system is designed to survive from the outset. Additionally, because all participating processes receive a copy of each message, all group members play an equal role in maintaining the state of the system and there are no single points of failure.

This thesis explores the use of periodic announcements as a replacement for acknowledged messaging. However, due to differences in transmission delays and message losses, the use of announcements may lead to inconsistent data among the different listening processes, as each participant may receive a different set of messages. As a result, this style of messaging commonly leads to the condition that each process is only able to approximate locally the global system state.

The tradeoff then in using announcements is that they avoid message implosion but may lead to inconsistency. We therefore are most interested in the class of applications that can tolerate transient inconsistency, so long as the property of eventual consistency is maintained. As evidence of these applications, a new class of protocols has emerged that are based on the principles of multipoint communication, often used in conjunction with announcement-style messaging. These protocols are efficient, scalable, and highly robust as a result.

These protocols reside at a variety of levels in the protocol stack and provide a range of critical Internet services. The protocols range from algorithms for routing (IGMP [13], PIM [18]), to those for quality of service (RSVP [61]), real-time transport (RTP [54]), reliable multicast (SRM [27]), distributed directories (SAP [35], SIP [36], SRVLOC [34]), and more recently, even what we have come to think of as the Internet itself, the Domain Name System (DNS [58] [21]). These protocols represent both deployed and proposed Internet standards. Yet, the principles of multipoint and announcement-style communication have application beyond the Internet, in any context where distributed control arises.

We observed that many of these multipoint protocols rely on a small set of very powerful techniques, or micro-algorithms, in order to accomplish scalability. The main goal of this thesis, through analysis and simulation, is to evaluate several of these fundamental techniques so that designers can gauge their usefulness over a range of operating conditions. An analysis of these techniques provides insight not only into their behavior, but also into the behavior of more complex algorithms that rely upon them.

1.2 Techniques for Scalability

In this thesis, we focus on three of the techniques that are repeatedly used for scalable group communication: Suppression (SUP), Announce-Listen (AL), and Leader Election (LE).

Suppression. Suppression is the technique whereby a process inserts a random delay before message transmission. In the delay interval, if the local process receives a message from another process, then the local process suppresses the transmission of its own message. If not, the announcement is sent as intended.

Suppression is an important tool to employ when processes arrive into a system concurrently, as might happen when processes reboot after a failure, or when all processes respond to the same message received. Suppression takes what would have been a group of simultaneous transmissions and spreads them out over the delay interval, simultaneously making message implosion less likely and reducing the number of messages ultimately sent, since earlier messages will suppress later messages.

Announce-Listen. In Announce-Listen, a sender process disseminates information to a group of processes by issuing periodic multicast announcements, and receivers passively listen for these announcements. A listener process infers information about the global state of a system from the periodic receipt or loss of messages from announcer processes.

AL is used in lieu of acknowledged messaging in situations where there is little or no back-channel bandwidth, i.e., there is limited bandwidth in the direction from receivers to senders. As a result, AL is beneficial in those instances where usage of traditional ACKs or NACKs coming back from receivers would inundate a sender. In addition, AL is useful to employ when information changes happen at a high enough rate to warrant continual updates. Thus, AL messaging is resilient to faults in the network because state is continually replenished. AL is well-suited for scenarios where group membership is constantly in flux, as late-joiners will quickly learn the current state of the system.

Leader Election. Leader Election is an algorithm that identifies a single process from among a group of processes. The criteria for a process becoming leader is agreed upon apriori by all participating processes (e.g., largest process identifier, lowest processing load, highest degree of connectivity, et cetera). By listening to announcements, each group member independently determines who the leader is and independently detects when the leader has departed. Thus, leader election occurs through loose coordination of the processes rather than strict consensus, which might require several rounds of exchanges among all processes in order to converge to a single leader.

LE is similar in its effect to SUP in that it suppresses the number of messages generated by a group of processes. LE, however, more formally identifies a single "leader" process to act on behalf of the other processes. LE keeps the overhead of group messaging low by reducing the numbers of group members that need to issue messages. As a result, LE helps a group of processes to behave as if it were one process.

1.3 Network Model

In the protocols we study, their analysis, and the accompanying simulations, we assume that the network supports a multipoint abstraction. Any message sent to a group address is forwarded to all group members subscribed to the address at that time. There is also the assumption that all messages issued are announcements, in that messages sent do not trigger acknowledgments or negative acknowledgments in response.

Communication groups may change in size over their lifetime [4]. Our analysis primarily concentrates on the scenario where all processes arrive simultaneously into the system, and subsequently when steady state is achieved, i.e., the group membership is steady. We also discuss the impact of perturbations to the membership, where applicable.

We assume message loss can occur. As such, we analyze each algorithm using a probabilistic failure model. Traditional analysis for distributed systems provides hard guarantees for conditions that always hold (safety properties) or conditions that are guaranteed eventually (progress properties). The safety and progress properties that hold in the presence of unacknowledged messaging

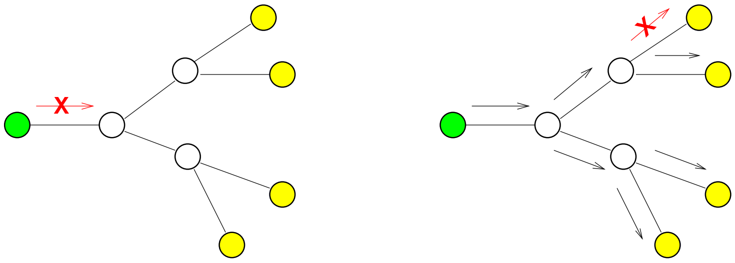

Figure 1.3: **Correlated vs. Uncorrelated Loss: Message Loss Near vs. Far from Sender.**

are very weak because, in the worst case, messages can be lost. Therefore, in this thesis, we devise probabilistic metrics to reason about the performance of each scalability technique.

Messages experience a fixed transmission delay across the network regardless of the location of the sender and receiver. For our calculations, the transmission delay actually encompasses transmission and processing delay, in that it captures the time that it takes for the receiver process to receive the announcement at the application level. We study fixed transmission delays as a prerequisite to understanding variable delays. The simplifying assumption that there are fixed delays makes the analysis tractable. However, the performance metrics that we derive for each scalability technique are still applicable in the variable delay case; provided we know the maximum (or minimum) delay between any pair of processes within the group, we are able to use the performance metrics to provide an upper (or lower) bound on performance for the group.

One challenge in modeling multicast traffic is that multicast distribution trees may lead to loss correlation (Figure 1.3). A message dropped close to the sender leads to correlated loss, as all receivers experience the same loss. A message dropped close to a receiver leads to uncorrelated loss, as it may or may not have been dropped at other receivers. A message dropped somewhere within the multicast distribution tree affects all downstream receivers. We evaluate the impact of both correlated and uncorrelated loss on protocol performance. We study the extreme cases where messages are dropped next to the sender or next to the receiver, in order to understand performance bounds. In the remainder of this thesis, we refer to these extremes simply as correlated or uncorrelated loss, respectively. In addition, we study the impact of multiple losses versus single losses on the behavior of the algorithms.

Another goal of this research is to expose key parameters that impact each algorithm and that have not been examined previously or completely. Consequently, we are able to study regions of the operating space that have not been investigated fully. This is accomplished in part by focusing on a multipoint context (as in the case of Announce-Listen) and in part by supporting more complex loss models (in the case of Suppression). Instead of looking at explicit topologies (as is often the case for Leader Election algorithms), we consider topologies in terms of the delay bounds they produce among processes.

By identifying a small but consistent set of parameters that are used in the mathematical formulae that describe each scalability technique, we enable protocol designers to properly tune their algorithms. For example, understanding the parameters allows us to answer typical questions such as: What is an appropriate timer value? How much delay is incurred when N processes participate? How much messaging overhead is generated when the loss probability is p? How long before there is a consistent copy of the data at all processes? Will an algorithm function properly when placed in a new context where critical network attributes are substantially different (e.g., using a protocol on a wireless network when it was designed originally for the wired realm)?

Bandwidth usage does not appear directly in the model. However, given a time interval T at which announcement messages are sent periodically, a message loss probability p, a network bandwidth capacity c, and a message size of s bytes, then we can define l to be an adjusted loss parameter that indirectly reflects bandwidth limitations. When the amount of traffic remains below c, the simple loss function p is maintained. When the bandwidth capacity is exceeded, only a fraction of the successful traffic gets through, thereby increasing the effective loss rate.

$$l = \begin{cases} p & \text{if } (s/T) \leq c \\ 1 - (1-p)(cT/s) & \text{otherwise} \end{cases}$$

Throughout this thesis, we primarily discuss the impact of p, a simple message loss probability, on the algorithms. However, in Chapter 5, we provide an example of the impact of congestion on p, which results in an effective loss rate that is much higher.

Not only do we aim to model and to analyze scalability techniques for group communication, but we use simulation to validate our results. All of the simulations presented in this thesis used the network simulator `ns` that was developed collaboratively by researchers at UC Berkeley, Lawrence Berkeley National Laboratories, USC Information Sciences Institute, Xerox PARC, and elsewhere [6].

1.4 Related Work

The use of probabilistic models builds on Chandy and Misra's work that explores the idea of process knowledge [15]. They formalize how to reason about what processes learn and assert that knowledge is only gained through message transmission. In later work [14], Chandy and Schooler show that process knowledge is too strong a concept for distributed applications in faulty environments and suggest a weaker but more appropriate concept of estimation. The algorithms we analyze in this thesis are part of the class of distributed problems that are best described with estimation probabilities. However, we analyze these algorithms using a more realistic loss model. We consider both single and multiple message losses, as well as the impact of correlated and uncorrelated losses on the group of communicating processes.

In [10] and [39], reliable multicast is analyzed under probabilistic failure models. Birman et al design a multicast protocol with a bimodal delivery guarantee; when a message is sent to a multicast group, almost all or almost none of the destinations receive the message with high probability. A key departure from the multicast techniques we analyze is that they add synchronization and ordering properties on top of the basic multicast announcements.

The idea of composition, that smaller protocols can be composed into larger protocols, is related to several systems. ISIS [11] is a toolkit of functions that can be combined to create group communication protocols. ISIS provides a rich set of semantic building blocks that focus on tightly-coupled communication, where senders and receivers have acknowledged message exchanges. However, ISIS only offers a fixed set of semantic properties that can be combined. In response to this limitation, software libraries such as Horus [57] and the research of Bhatti et al [9] have decomposed network algorithms into finer grain micro-protocols. Both allow flexibility in the kinds of properties that can be composed into larger protocols. The Ensemble system [38] takes these design principles one step further by using formal methods to reason about the correctness of the resulting protocols. Nonetheless, these efforts have a slightly different emphasis than our work, which is aimed at providing a probabilistic analysis of the behavior of the component algorithms.

In a similar vein, the Reliable Multicast Working Group of the Internet Research Task Force has begun to examine a building block approach to designing reliable multicast transport protocols [40]. Because there is sufficient commonality between the many existing reliable multicast protocols, it should be possible to define a small number of components out of which the others can be constructed. The kinds of building blocks that they propose are at a higher level than the techniques we study in this thesis. Additionally, our focus is on the analysis rather than functional attributes of the components. However, we expect that many of the building blocks that emerge for reliable multicast transport will rely on the techniques for scalability that we study here.

In addition to identifying the key components to build larger protocols, we must pose the question of equivalence; can we evaluate an algorithm by evaluating the components out of which it is built? Hayden touches on this in his work on formal methods that allow him to reason about the correct operation of constructed algorithms [38], given the properties of the component algorithms. Here, we ask the related question of whether or not the performance analysis of the components has any bearing on the performance analysis of the combined algorithms.

In the area of complexity theory there is the related strategy of using a series of transformations to translate one algorithm into another. This technique is used to decide if an algorithm of unknown complexity can be transformed into an algorithm already known to be NP-complete [32]. In our research, we ask if the performance bounds of the algorithmic building blocks are indicative of the bounds for the complex algorithms that rely upon them.

For the specific scalability techniques that we study, we review related work in each of the chapters in which they are discussed.

1.5 Overview

In Chapters 2 - 5 we derive a series of performance metrics for each scalability technique. These metrics provide probabilistic bounds for the behavior of the algorithms under various network conditions. The metrics focus on the expected response time, network usage, memory overhead, consistency attainable, and convergence time. Given the metrics, a designer is better equipped to parameterize the algorithms and to make informed tradeoffs. Moreover, the performance of an algorithm can be predicted, given that specific operating conditions exist (e.g., a certain level of loss or delay among group members). Broadly speaking, these chapters identify the key parameters that affect this class of loosely-coupled multipoint algorithms, and establish a general methodology for the analysis of these and other scalability techniques.

In Chapter 2, we evaluate the Suppression algorithm initially with no loss in the network. We deliberately scrutinize its performance separate from any particular context, e.g., reliable multicast. As a stand-alone entity, the algorithm is not subject to assumptions about how it may or may not be used in subsequent iterations. We show clearly that Suppression's response metric is in delicate balance with its overhead metric, and examine a new timer distribution function that raises the issue of "designer" distribution functions in general. Given a target range for parameter values, and a metric of choice, we establish that a distribution function can be created to optimize the performance of the algorithm.

In Chapter 3, we revisit Suppression under lossy conditions. We re-examine the metrics established in Chapter 2, but updated to reflect message loss. We present and analyze several new metrics that are important for Suppression, both as a stand-alone algorithm and as an algorithm combined with other techniques: the completion or halting time of the algorithm, as well as the number of extra versus required messages generated. Most importantly, we model loss more realistically than previous work on Suppression [27] [50] [45], allowing multiple losses and studying the effects of correlated and uncorrelated loss.

Chapter 4 presents the Announce-Listen algorithm, focusing on an analysis of its consistency model. We propose an alternate model for analysis than existing models [53] [49]. The new model has several advantages. It exposes essential parameters for evaluation: timers not only for announcement and update periodicity, but also for cache expiration; transmission delay, which can be compared with timer intervals; and group size, which allows a more accurate assessment of messaging overhead. Finally, unlike previous models, our model is directly implementable in simulation, allowing us to validate the analysis. With validation, the analysis has stronger predictive value.

In Chapter 5, we examine the Leader Election algorithm, exploring the idea that metrics for simple components can be used to bound the performance of the more complex algorithms that rely upon them. Specifically, we investigate the idea that Leader Election is composed from Suppression and Announce-Listen. In the optimal case, there is a direct correlation between the performance of LE and the performance of SUP and AL, but under non-optimal conditions it diverges considerably. Nonetheless, the Leader Election technique offers insight into the importance of equivalence in compositional algorithms. In addition, our examination of Leader Election is a departure from the norm, in that we do not tie it to a particular topology. Moreover, we analyze its judicious use of Suppression to scale beyond conventional LE algorithms and explore the use of an alternate leader election criteria to improve convergence. As with other techniques, we study the algorithm in the context of both correlated and uncorrelated loss models, to more accurately reflect the range of loss conditions that groups of processes may encounter.

In Chapter 6, we summarize our results, contributions and future directions.

Chapter 2

Suppression

In this chapter, we present an analysis of the basic Suppression (SUP) algorithm. We describe the behavior of the algorithm using pseudocode. We discuss why it is considered a scalable technique for groups of communicating processes, and under what conditions it is most useful to employ.

We introduce two key performance metrics to describe the Suppression algorithm: Time Elapsed, the delay incurred by a process that employs the Suppression technique; and Extra Messages, the corresponding messaging overhead. We present the intuition behind the derivation of these metrics, then explore the impact of different Suppression timer distribution functions on the metrics. We examine a uniform distribution, as well as a decaying exponential distribution. We also investigate the creation of designer distribution functions that optimize a particular metric, in those instances where a subset of the network parameters remain fixed.

By studying deployed protocols and applications, we are able to use realistic values for various network parameters that affect the metrics. Consequently, we estimate operating ranges for the metrics and use these ranges as input to simulations. We present simulation results that validate the analysis. In addition, we provide an overview of related work, a summary of our findings, and proposals for future avenues of research.

In Chapter 3, we revisit the analysis of Suppression under lossy network conditions. In Chapter 5, we explore the use of Suppression as a building block for other algorithms; we examine its role in the composition of the Leader Election algorithm.

2.1 Core Algorithm

The general behavior of the algorithm is described in this section using pseudocode (Program 2.1). There are N processes[1] numbered $0, \ldots, N-1$. When a process i arrives in the system, it chooses a random time t to delay sending its message by calling `random`(d, T). Process i selects time t

[1] In general, we assume the processes reside on N distinct processors across the network. However, Suppression is still useful even when the processes are co-resident on a single processor. If processes awaken simultaneously, the processor still may wish to spread out and to aggregate responses in order to handle the load.

with random probability as described by the density d. If the distribution is uniform, t is randomly selected from the uniform interval $[0, T]$, whereas if it is exponential, t is selected from a decaying exponential distribution with decay parameter $\alpha = T$ (Section 2.4.2). The process then calls the `sleep`(t) routine, causing the process to wait for t units of time to elapse. On awakening, the process checks if it has received a message from any other processes, $j \neq i$, during the delay interval. If there were `no_messages_received`(), process i sends its message. Otherwise, the message is suppressed.

```
SUPPRESSION(d, T)
  t = random(d, T)
  sleep(t)
  if no_messages_received() then
    send_message()
```

Program 2.1: **Suppression Algorithm.**

2.2 Scalability

The Suppression (SUP) algorithm is considered scalable because it decreases the number of messages generated by a group of communicating processes. SUP takes what would have been simultaneous message transmissions and spreads them out over the suppression interval, making message implosion less likely. At the same time, early messages suppress the messages that would have been sent by later awakening processes. Therefore, SUP reduces the overall number of messages sent, ideally replacing multiple message transmissions with a single message. The end result is that Suppression helps avoid network congestion, processor overrun, and packet loss.

Suppression is an important tool to employ when all processes generate similar messages, which otherwise would lead to redundancy. Messages generated by different processes can be replaced by a single message because the content of the messages is identical.

Suppression may be needed when, for example, process actions are synchronized in time: when processes arrive for a scheduled event, such as a teleconference; when all processes respond to the same multicast message received, such as a query sent to a cluster of servers; and when all processes bootstrap themselves at initialization or after a failure.

2.3 Metrics

We can analyze the effect of Suppression in the worst case when all N processes *arrive* into the system simultaneously. Simultaneous arrivals are considered a worst case scenario because that is when there is the greatest potential for concurrent message transmission. When the number of processes grows large, there are potentially even greater bottlenecks due to synchronized competition for resources, such as network bandwidth, CPU cycles (to handle message interrupts), or memory for the storage of process state.

The scalability afforded by Suppression is not without a cost. What is gained by reducing messaging overhead is at the expense of timeliness. Therefore, the effectiveness of the Suppression algorithm is measured using two metrics: the time before the first message is sent and the number of extra messages generated. The former gauges the latency of the algorithm, whereas the latter gauges the message overhead. The challenge is to minimize the delay but also to minimize the number of messages transmitted.[2]

We explore these metrics in the sections below. Our approach is to define the metrics analytically using probability theory, then to examine the effects of allowing the random delay variable to be selected using different distributions. We use simulation to validate our results since approximations are used to make the analysis tractable.

For simplicity, the metrics are initially studied in the context of a network without message loss, where the transmission delay between processes, Δ, is fixed. In Chapter 3, the model is extended to consider loss.

2.3.1 Time Elapsed

The Time Elapsed metric measures the minimum time t_i selected by any process. The minimum time selected also can be construed as the minimum delay before all other processes have the potential to become suppressed. In some sense, this can be viewed as the minimum potential *stopping time* of the algorithm, or its earliest opportunity to complete. Completion is important, as Suppression is often used as an initial mechanism to reduce the pool of participating processes (i.e., processes issuing messages), after which another algorithm is performed among the remaining processes (IGMP [24], PIM [23], SRM [27]).

Expected Value. The expected value of a random variable x is $E[x] = \sum xp(x)$, where $p(x)$ is the probability density function. The expected value of t_{min}, the minimum elapsed time before one of the processes wakes up and sends a message, is derived as follows.

Process i picks time t, using a probability density function $p_i(t)$, where $0 \leq t \leq T$ and T is the maximum amount of time a process waits before issuing a message. $P_i(t)$ is the cumulative distribution function, i.e., $P_i(t) = \int_0^t p_i(x)\, dx$.

The time chosen by process i is the minimum if every other process picks a time greater than t. Thus, in the second step of the derivation below, the sum contains N terms, and the product

[2] The time until the last message is sent is also an interesting metric, but more so when loss is introduced. We discuss it in detail in Chapter 3. In the lossless case, the upper bound on the time until the last message is simply the delay until the first message is sent plus the maximum transmission delay between any pair of participating process.

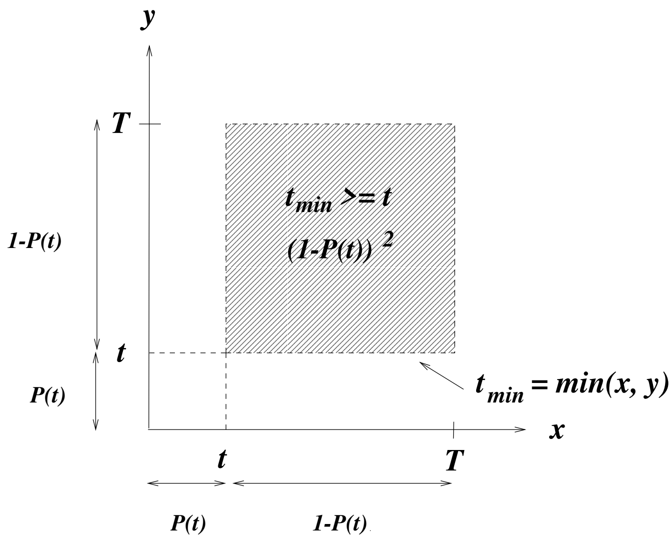

Figure 2.1: **Intuition Behind $E[t_{min}]$ for 2 Processes.**

contains $N-1$ terms. We add up the probabilities for each i, which gives us the likelihood of the minimum being equal to t. We assume that all density functions are the same, which allows us to drop the subscripts i and j between the second and third steps of the derivation.

$$\begin{aligned} E[t_{min}] &= E\Big[\min_{0\le i<N} t_i\Big] \\ &= \int_0^T t \sum_{0\le i<N} p_i(t) \prod_{j\ne i} \int_t^T p_j(x)\,dx\,dt \\ &= \int_0^T (1-P(t))^N\,dt \end{aligned}$$

Intuition. The intuition behind this formula is displayed in two-dimensions in Figure 2.1. Consider two processes, shown as two independent random variables x and y, where the cumulative distribution function is $P(t) = t/T$. Each process selects a time to delay. The minimum value is $t_{min} = min(x, y)$. The probability that $x \le t$ is $P(t)$. Therefore, the probability that $x \ge t$ is $1 - P(t)$. Since x and y are independent variables, the probability that both values x and y are greater than or equal to t is the same as the probability that $t_{min} \ge t$. The result is equal to $(1 - P(t))^2$, which is shown by the shaded region.

This construction generalizes to N dimensions. When there are N processes participating in the algorithm, the result is the area defined by $(1-P(t))^N$, which is exactly the integral proposed earlier. Although Figure 2.1 depicts a uniform distribution, the results generalize to other distributions as well.

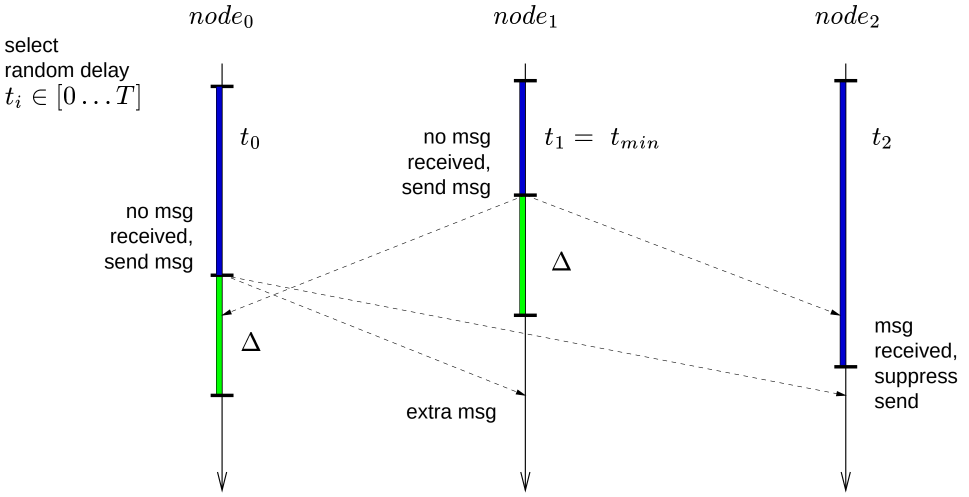

Figure 2.2: **Suppression Algorithm.**

2.3.2 Extra Messages

In a lossless network without transmission delay, only one message is generated by the Suppression algorithm. The earliest message generated suppresses all other messages.

In a lossless network with delay, any message generated beyond the earliest message is considered an extra message. Extra messages are generated because messages take a non-zero time to reach listeners. The earliest message is sent at time t_{min}, the minimum t_i, $0 \leq i < N$. A process receives this message at time $t_{min} + \Delta$, where Δ is the message transmission delay. Any message sent at time t_k such that $t_{min} < t_k < t_{min} + \Delta$ corresponds to an extra message that was not suppressed.

In Figure 2.2, three processes arrive into the system at the same time. Each selects a time t_i to delay. Transmission delay between all pairs of nodes equals Δ. The minimum delay t_{min} is chosen by $node_1$. Therefore, when $node_1$ does not receive a message during the interval $[0, t_1]$, it awakens and issues a message. When $node_2$ awakens after t_2 units of time, the process discovers it received a message from $node_1$, therefore suppressing its own message. However, on awakening at time t_0, $node_0$ has not yet received the message sent by $node_1$ because $t_{min} < t_0 < t_{min} + \Delta$. Therefore, $node_0$ sends its message, which is considered an extra message. Note that the number of extra messages is a count of the extra messages sent (not received).

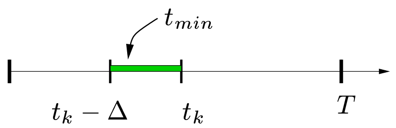

Figure 2.3: **Condition for Extra Message Transmission.**

Expected Value. The expected number of extra messages sent is calculated as follows. The expected number of extra messages is simply the sum of the likelihood of each process sending an extra message, or the number of processes that have selected a delay time that is within Δ of the minimum time selected. In other words, a process for which $t_k - \Delta < t_{min} < t_k$ holds true is a process that generates an extra message (Figure 2.3). The likelihood that the minimum lies between $t_k - \Delta$ and t_k is calculated by determining the likelihood that each process $i \neq k$ picks a time greater than $t_k - \Delta$, and subtracting from it the likelihood that every process $i \neq k$ picks a time greater than t_k.

$$\begin{aligned} E[\#\ extra] &= \sum_{0 \leq k < N} Pr\left[t_k - \Delta < t_{min} < t_k\right] \\ &= N \cdot P(\Delta) - 1 + N \int_{\Delta}^{T} p(t)(1 - P(t-\Delta))^{N-1}\, dt \end{aligned}$$

Intuition. The intuition behind this calculation is shown in Figure 2.4. Consider the problem with 2 processes, represented as two independent random variables x and y, where the cumulative distribution function is $P(t) = t/T$. The probability that process x chooses a suppression value $x \geq t$ is $1 - P(t)$, and the probability that process x chooses a suppression value $x \geq (t - \Delta)$ is $1 - P(t - \Delta)$. Therefore, the probability that x lies between $t - \Delta$ and t is equivalent to $(1 - P(t-\Delta)) - (1 - P(t)) = P(t) - P(t - \Delta)$.

The probability that $t_{min} = min(x, y)$ lies between t and $t - \Delta$ requires we perform this calculation in both dimensions, shown by the shaded area in the diagram. This result generalizes to N dimensions.

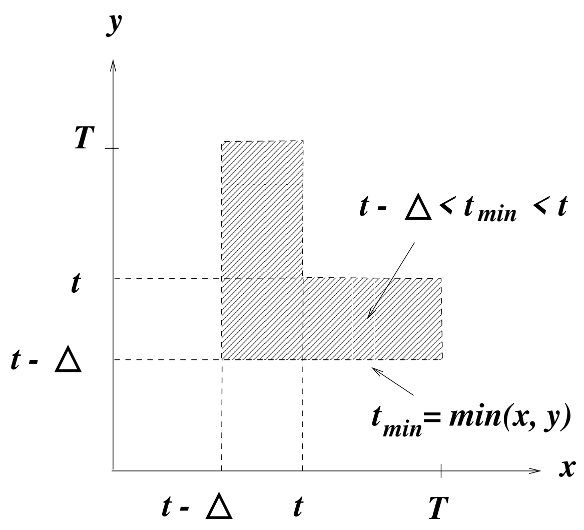

Figure 2.4: **Intuition Behind $E[\#extra]$ for 2 Processes.**

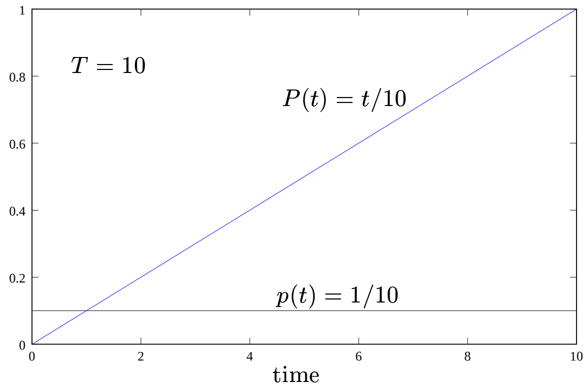

Figure 2.5: **Uniform Distribution.**

2.4 Distributions

In this section, we apply the analysis derived above to the uniform distribution and the decaying exponential distribution. The uniform distribution is attractive from the perspective that it is used in most deployed Suppression algorithms. It is also simple to implement and more efficient than implementations of exponential distributions. The exponential distribution is interesting from the standpoint that it has been used as an integral part of collision-avoidance algorithms for shared media, like Ethernet and packet radio. In addition, it is used as the basis for collision back-off schemes for shared media networks, among others.

2.4.1 Uniform Distribution

The most common probabilistic model in existing suppression algorithms is the uniform distribution. Each process randomly selects a value t from a uniform distribution in the range $[0 - T]$. Note that there is equal likelihood that any process will pick t_{min}; all processes are treated equally.

In Figure 2.5, we show a uniform distribution, with $p(t) = 1/T$, and $P(t) = t/T$. When the upper bound of the suppression interval is set to $T = 10$, the probability density of a process selecting a specific wake-up time t is $p(t) = 1/10$ and the cumulative probability for selecting t is $P(t) = t/10$.

$$
\begin{aligned}
E[t_{min}] &= \int_0^T (1 - t/T)^N \, dt \\
&= \left\{ \frac{-T(1 - t/T)^{N+1}}{N+1} \right|_0^T \\
&= \frac{T}{N+1}
\end{aligned}
$$

Consider one process. Intuitively, $E[t_{min}] = \frac{T}{2}$ makes sense. Averaging over many runs, the minimal suppression timer converges to the midway point, $T/(N + 1)$, of the suppression interval, T. For a

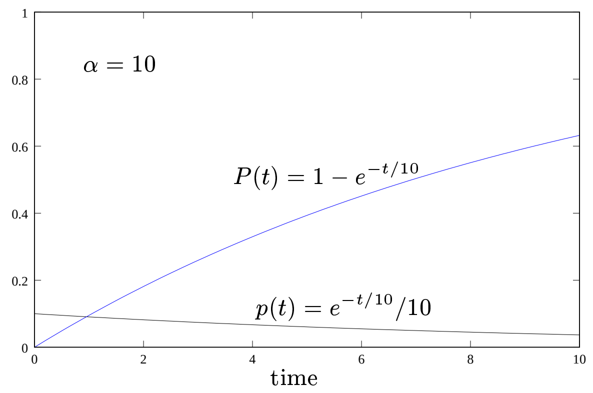

Figure 2.6: **Exponential Distribution.**

given N, it is more likely that one of the timer values selected will be less than the minimum timer value for the $N-1$ case, which is why the minimum timer value, on average, dissects the timer interval by $(N+1)$ since there exists one more random variable, which can be greater or less than t_{min}. If the new value is greater than t_{min}, the minimal suppression value is unchanged. However, if the new value is less than t_{min}, then t_{min} would decrease. Because there is a non-zero probability of the new value being less, t_{min} decreases as N increases.

$$\begin{aligned} E[\#\ extra] &= N\frac{\Delta}{T} - 1 + N\int_{\Delta}^{T} 1/T(1 - t/T + \Delta/T)^{N-1}\,dt \\ &= N\frac{\Delta}{T} - \left(\frac{\Delta}{T}\right)^{N} \end{aligned}$$

Consider what we can deduce about the behavior of the metric for extra messages. When $N = 2$, the algorithm generates Δ/T extra messages on average. The intuition behind this result is that when there are two processes, one wakes up earliest at t_{min} and the other wakes up within Δ of t_{min} Δ/T of the time.

2.4.2 Decaying Exponential Distribution

For the case of the decaying exponential distribution, $p(t) = e^{-t/\alpha}/\alpha$, $P(t) = 1 - e^{-t/\alpha}$. In Figure 2.6, we set $\alpha = 10$, resulting in $p(t) = e^{-t/10}/10$, $P(t) = 1 - e^{-t/10}$.

$$\begin{aligned} E[t_{min}] &= \int_{0}^{\infty} (e^{-t/\alpha})^{N}\,dt \\ &= \frac{\alpha}{N} \end{aligned}$$

$$\begin{aligned} E[\#\ extra] &= N(1-e^{-\Delta/\alpha}) - 1 + N\int_{\Delta}^{\infty} \frac{e^{-t/\alpha}}{\alpha}(e^{-(t-\Delta)/\alpha})^{N-1}\,dt \\ &= (N-1)(1-e^{-\Delta/\alpha}) \end{aligned}$$

2.5 Analysis

In this section, we explore realistic values for T, N, and Δ, as well as the ratios T/N and Δ/T, which appear prominently in the derivation of the metrics. The parameter ranges are gleaned from the observation of deployed protocols and applications. Setting realistic parameter bounds serves two purposes; we can examine the operating range of the metrics and use these ranges as input to our simulations. We also study the impact of a distribution choice on the metrics. We analyze the uniform and exponential distributions separately, then compare the operational crossover points for each metric. Finally, we present our simulation results, which fully validate the analysis.

2.5.1 Realistic Parameters

For existing algorithms, the suppression interval, T, takes on a value of $O(1)$ second for local updates and quickly changing data (SRM [27]); $O(5)$ seconds for real-time control status updates (RTCP [54]); $O(10)$ seconds for multicast group management (IGMP [24]); $O(.5)$ - $O(1)$ minutes for router and QoS protocols (PIM [22] [17] and RSVP [61] [60] [8]); $O(5)$ minutes for slowly changing data updates and updates using adaptive periodicity to maintain fixed bandwidth. Some of these values represent announcement periodicities rather than suppression intervals. They are included here because they measure the relative frequency of process interactions in system-wide multicast algorithms. We display the timer ranges in Table 2.1.

N, the number of processors participating in the algorithm, ranges from $O(1)$ for DNS server load balancing, distributed file systems, and personal conferencing, to $O(10)$ for ISP mail servers and Web gateways, to $O(100)$ for parallel computations, to $O(1000)$ for Internet telepresentations [3], and $O(10K)$ for the number of processors that might simultaneously request a Web page from a popular site (Table 2.2).

An estimate of Δ, the transmission delay, depends on the domain of communication. The range is quite large when we compare Ethernet performance to that of the wide-area Internet. The variance is also pronounced when comparing the local Ethernet delay with that of satellite or modem communication, both considered slow links. Our estimated values for delay are .1-.5 msec for the local subnet (Ethernet), $O(1)$ msec local but different subnets (e.g., 2-3 msec across a bridge), $O(10)$ msec to transit a router or series of routers (note that a delay-free transmission takes approximately 30 msec to transit the continental US, based on the speed of light), $O(100)$ msec for slow connections within the US and communications between continents and beyond, $O(500)$ msec for roundtrip Satellite communication, and finally $O(10)$ seconds and upwards for Interplanetary Internet transmissions (Table 2.3). The times reported include the time for a message to transit the network, as well as to get through the operating system network protocol stack.

$O(T)$	*Protocol*
1 sec	SRM
5 sec	RTP
10 sec	IGMP
30 sec	RSVP
60 sec	PIM
5 min	DNS, MZAP

Table 2.1: **Timer Range (T).**

$O(N)$	*System*
10^0	Distributed file systems, Personal conferencing
10^1	ISP mail servers, Web server load balancing
10^2	Parallel computations
10^3	Internet telepresentations
10^5	Simultaneous Web page requests

Table 2.2: **Number of Processes (N).**

$O(\Delta)$	*Network*
.1 msec	Same Subnet
1 msec	Local but Different Subnet
10 msec	Across a Router
30 msec	Speed of Light across US
100 msec	Moderate US connection
500 msec	Satellite transmission
10 sec	Interplanetary Internet

Table 2.3: **Transmission Delay (Δ).**

We assume $T \geq 1$, $N \geq 1$, and $\Delta \leq T$. Therefore $\Delta/T \leq 1$ and has a positive range of $[3e\text{-}06, .5]$, whereas T/N has a positive range $[1e\text{-}05, 300]$.

2.5.2 Uniform Distribution

When using a uniform distribution for the Suppression timer, the expected delay before which a message is sent, $E[t_{min}]$, is simply a function of T and N. Using a log scale on the y-axis, we plot $E[t_{min}]$ vs. T in Figure 2.7, and using a log-log plot, we show $E[t_{min}]$ vs. N in Figure 2.8. Simply

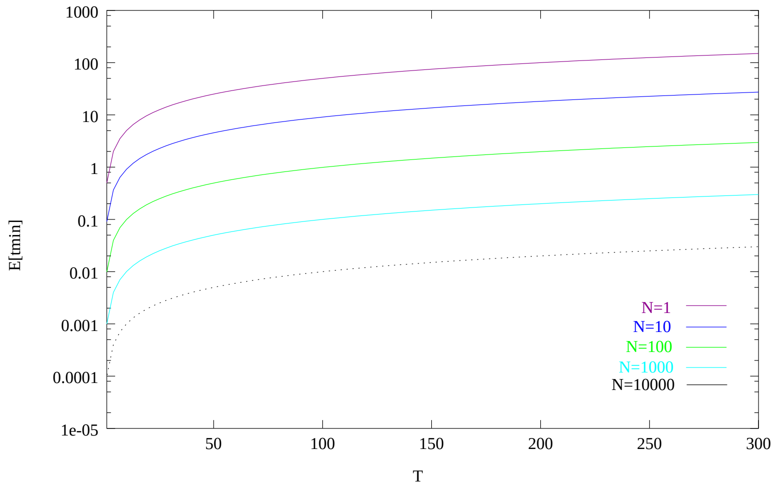

Figure 2.7: **Uniform: Time Elapsed vs. T.**

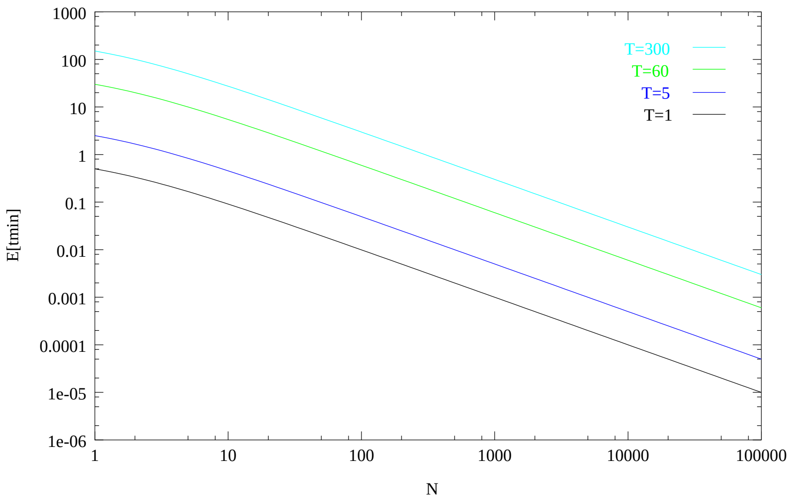

Figure 2.8: **Uniform: Time Elapsed vs. N.**

put, $E[t_{min}]$ grows with T, the upper bound on the suppression range, and shrinks with N, the number of processes participating in the algorithm.

The relationship between the elapsed time $E[t_{min}]$ and the variables T and N is even more clearly evident as N grows very large,

$$\begin{aligned} E[t_{min}] &= \frac{T}{N+1} \\ &\approx \frac{T}{N} \end{aligned}$$

As N grows very large, $E[t_{min}]$ approaches 0.

The ratio Δ/T plays a critical role in the expected number of extra messages sent, $E[\#\ extra]$.

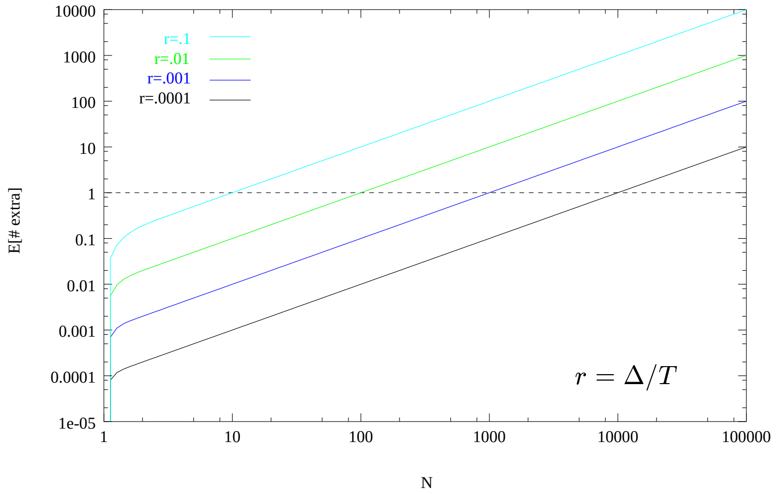

Figure 2.9: **Uniform: Extra Messages vs. N.**

Using a log-log scale, Figure 2.9 plots $E[\#\ extra]$ vs. N and displays the effects of the ratio $r = \Delta/T$ over several orders of magnitude. Figures 2.10 and 2.11 display $E[\#\ extra]$ vs. Δ/T with large and small N respectively.

As N grows very large,

$$\begin{aligned} E[\#\ extra] &= N\left(\frac{\Delta}{T}\right) - \left(\frac{\Delta}{T}\right)^N \\ &\approx N\left(\frac{\Delta}{T}\right) \end{aligned}$$

Statistically it will take $N \geq T/\Delta$ processes to generate extra messages. Therefore, when N is very large, Δ/T must be extremely small to avoid extra messages. When N is small, the ratio between Δ and T can afford to be larger. Thus the relationship between N and Δ/T is important for proper system parameterization.

This observation suggests that suppression may not be effective at keeping the number of extra messages low for extremely large groups, especially in contexts like the wide-area Internet where it may be hard or even impossible to keep the Δ/T ratio predictably low, and hence to keep large numbers of extra messages from being generated. Extra messages are also a concern for bandwidth-limited networks (e.g., modem channels, wireless sensor nets), which cannot afford the overhead.

Note that when the metrics $E[t_{min}]$ and $E[\#\ extra]$ are examined with N set to a large value, their inverse relationship becomes clear. When the parameters that influence the expected number of extra messages are re-arranged,

$$E[\#\ extra] \approx N(\Delta/T) = \Delta(N/T) \approx \Delta/E[t_{min}]$$

There is a direct tradeoff between reducing the number of extra messages generated and increasing the minimum delay before which the first message is sent. The metrics are inversely proportional by a factor of Δ.

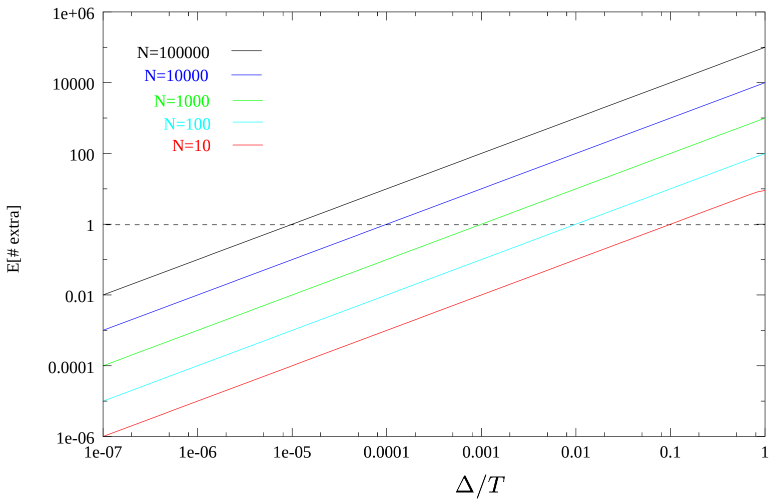

Figure 2.10: **Uniform: Extra Messages vs. Δ/T (Large N).**

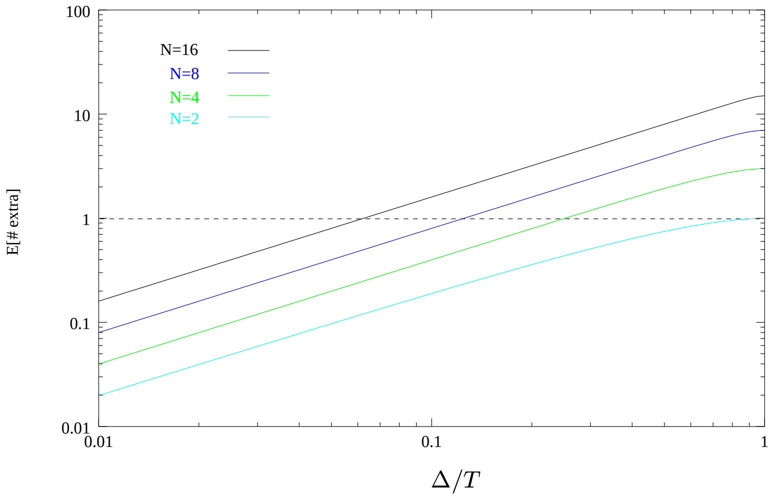

Figure 2.11: **Uniform: Extra Messages vs. Δ/T (Small N).**

While a given system parameterization might improve one metric, it is **always** at the expense of the other. Therefore, the optimal balance will be a function of the operating environment. For example, if the context is a group of processes communicating over a network with limited capacity or a group of sensors with limited power for communication, the goal might be to keep $E[\#\ extra]$

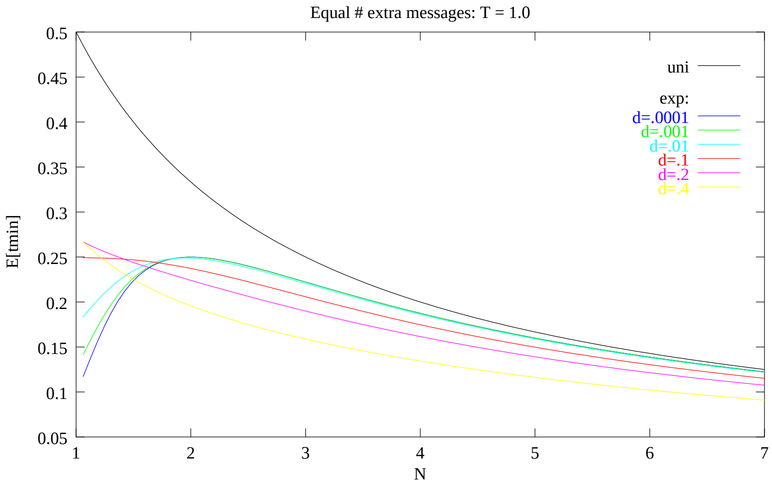

Figure 2.12: **Comparison: Time Elapsed vs.** N **(Varying** $d = \Delta$**).**

below a given threshold. If N or Δ fluctuates, then T will need to be adaptive to compensate. We elaborate on this point in Section 2.7.

2.5.3 Decaying Exponential Distribution

When using an exponential distribution for the Suppression timer, the expected delay before which a message is sent, $E[t_{min}]$, is now a function of α and N. As N grows very large, $E[t_{min}]$ behaves like the Uniform distribution, with $\alpha = T$.

$$E[t_{min}] \quad \approx \quad \frac{\alpha}{N}$$

Similarly, the ratio Δ/α plays a critical role in $E[\#\ extra]$. As N grows large,

$$\begin{aligned} E[\#\ extra] \quad &= \quad (N-1)(1-e^{-\Delta/\alpha}) \\ &\approx \quad N\left(1-e^{-\Delta/\alpha}\right) \end{aligned}$$

2.5.4 Comparisons

Below, the performance of the uniform and decaying exponential distributions are compared. The cross-over points for $E[t_{min}]$ and $E[\#\ extra]$ are determined by setting the metrics for the two distributions equal under certain conditions.

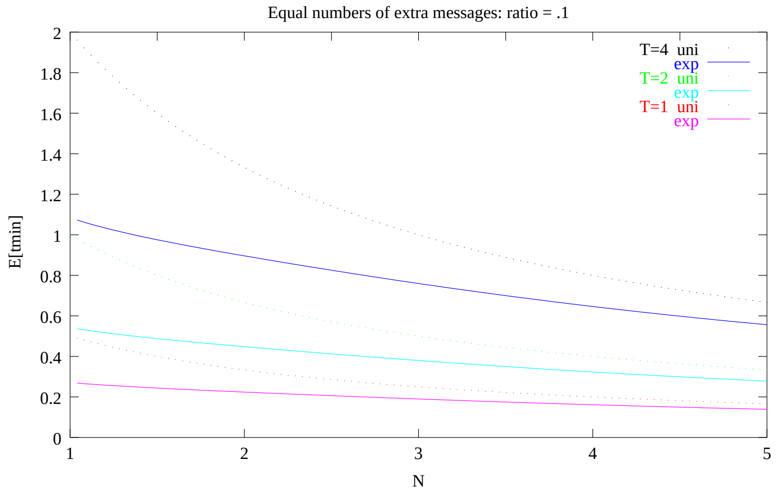

Figure 2.13: **Comparison: Time Elapsed vs. N (Varying T).**

Time Elapsed. The decaying exponential distribution generates the same average number of extra messages as the uniform distribution if we set $E[\#\ extra]$ equal for the two distributions, solve for α, then plug α back into the original equation to rederive $E[t_{min}]$ for the decaying exponential distribution,

$$\begin{aligned}
N\left(\frac{\Delta}{T}\right) - \left(\frac{\Delta}{T}\right)^N &= (N-1)(1-e^{-\Delta/\alpha}) \\
\alpha &= -\Delta/\ln\left(1 - \frac{N\,(\Delta/T) - (\Delta/T)^N}{N-1}\right) \\
E[t_{min}] &= \frac{\alpha}{N} \\
&= -\Delta/N\ln\left(1 - \frac{N\,(\Delta/T) - (\Delta/T)^N}{N-1}\right)
\end{aligned}$$

When $E[t_{min}]$ vs. N is plotted while keeping the number of extra messages equal, the exponential distribution is marginally more responsive than the uniform distribution. This phenomenon is more pronounced for small N and larger ratios, $.1 < \Delta/T \leq 1$ (Figure 2.12), as well as for larger T (Figure 2.13).

Extra Messages. The decaying exponential distribution generates the same minimum delay as the uniform distribution if we set $E[t_{min}]$ equal for the two distributions. To rederive $E[\#\ extra]$ for the decaying exponential distribution, we solve for α, then plug it back into the original equation. The last step is reached by using the Taylor expansion $e^x = 1 + x + x^2/2! + x^3/3! + \ldots \geq 1 + x$, and noting that $1 - e^x \leq -x$.

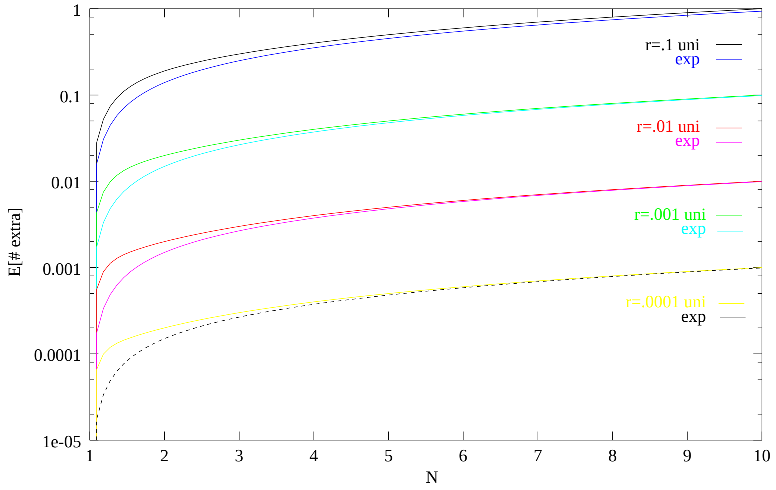

Figure 2.14: **Comparison: Extra Messages vs.** N **(Small** $r = \Delta/T$**).**

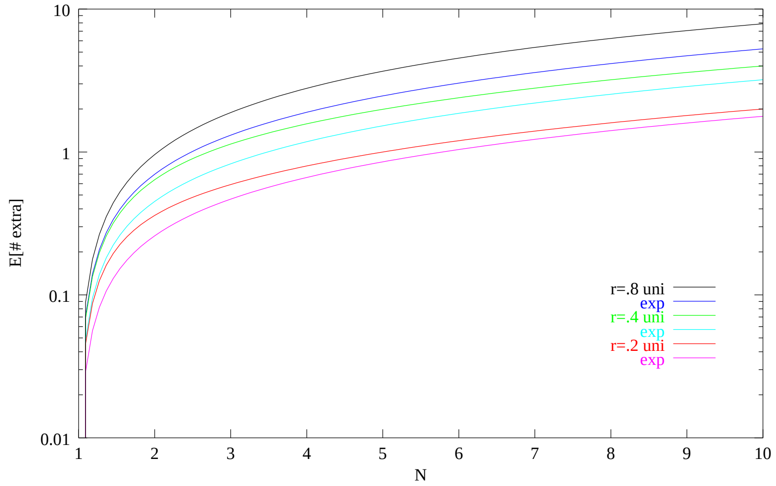

Figure 2.15: **Comparison: Extra Messages vs.** N **(Large** $r = \Delta/T$**).**

$$
\begin{aligned}
\frac{T}{N+1} &= \frac{\alpha}{N} \\
\alpha &= TN/(N+1) \\
E[\#\ extra] &= (N-1)(1-e^{-\Delta(N+1)/TN}) \\
&\leq \frac{(N-1)\Delta(N+1)}{TN} = N\left(\frac{\Delta}{T}\right) - \frac{1}{N}\left(\frac{\Delta}{T}\right)
\end{aligned}
$$

Under these conditions, the exponential distribution always outperforms the uniform distribution in terms of E[# messages]. However, with small Δ/T, the impact is fewer than one extra message, and the exponential and uniform curves quickly converge beyond small N. With larger ratios, the effects are significant, especially as N increases. These results are evident in Figures 2.14 and

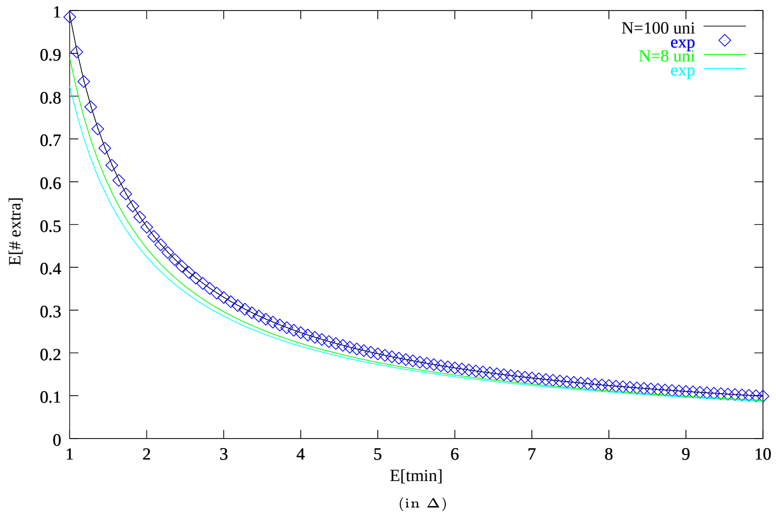

Figure 2.16: **Comparison: Extra Messages vs. Time Elapsed (Large N).**

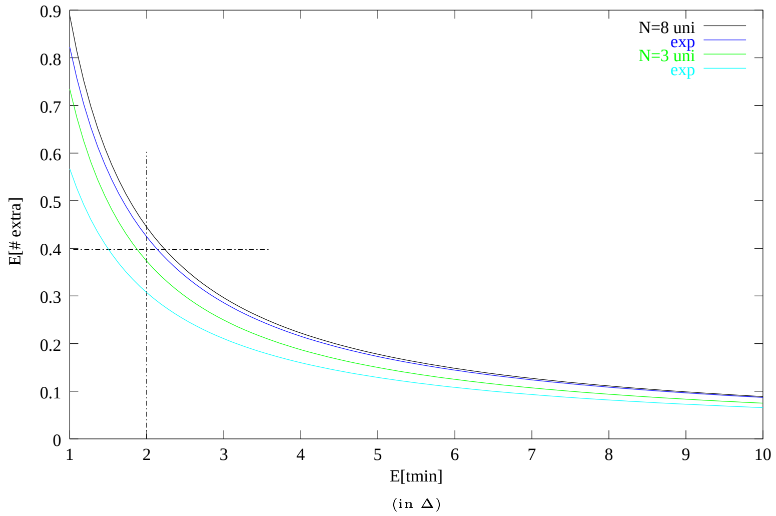

Figure 2.17: **Comparison: Extra Messages vs. Time Elapsed (Small N).**

Figure 2.15, which display $E[\#\ extra]$ vs. N for uniform and exponential distributions with small and large ratios of Δ/T respectively. The results raise the issue that it may make sense to design adaptive systems that use one distribution in one part of the operating range and another when it crosses into another operating range. For example, when $\Delta/T \leq .1$ use a uniform distribution, whereas if $\Delta/T > .1$ use a decaying exponential distribution. Current multicast protocols employing the Suppression technique strictly rely on uniform distributions, meaning that if an application has a requirement to minimize messaging overhead (as compared to responsiveness), then it is not having its needs met adequately. In addition, this raises the larger issue of whether or not to explore other distributions.

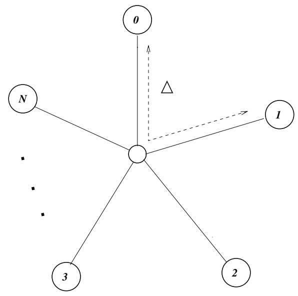

Figure 2.18: **Star Topology with fixed Δ.**

Extra Messages vs. Time Elapsed. In Figure 2.16 and Figure 2.17, $E[\#\ extra]$ vs. $E[t_{min}]$ is plotted for uniform and exponential distributions, with large and small N, respectively. Units of Δ are derived for the uniform distribution by observing that $E[t_{min}] = T/N + 1$. Therefore, $T = (N+1)E[t_{min}]$. Substituting for T in $E[\#\ extra]$ and setting $x = E[t_{min}]/\Delta$, we derive:

$$\begin{aligned} E[\#\ extra] &= N\left(\frac{\Delta}{(N+1)E[t_{min}]}\right) - \left(\frac{\Delta}{(N+1)E[t_{min}]}\right)^N \\ &= N\left(\frac{1}{(N+1)x}\right) - \left(\frac{1}{(N+1)x}\right)^N \end{aligned}$$

The graphs display the tradeoff between $E[\#\ extra]$ and $E[t_{min}]$ in units of Δ for different N. The probability of producing extra messages diminishes as delay increases. Regardless of Δ, one observes which curves have better responsiveness, $E[t_{min}]$, by drawing a horizontal line at a constant $E[\#\ extra]$. Similarly, to obtain which curves have less overhead, $E[\#\ extra]$, one draws a vertical line at a constant $E[t_{min}]$. Note that both graphs reveal that for smaller N, the decaying exponential distribution outperforms the uniform distribution. By $N = 8$, the differences are small, and by $N = 100$, imperceptible.

2.6 Simulation

In our initial simulations, N, T and Δ were fixed. The network topology was configured so that the transmission delay between all pairs of nodes was identical. This was accomplished by creating a star topology (Figure 2.18), with the processes at the edges of the star, and with Δ calculated as specified below.

$$\Delta = \frac{s}{b} + d$$

s = packet size in bits

b = link speed in bits per second

d = link delay in seconds

Each simulation was run 1000 times using the ns simulator [6]. The range for N was examined from 1 to 10 in increments of 1, and from 10 to 100 in increments of 10. For a few specific data sets, we also examined N between 100 and 1000 in increments of 100.

Each node chose a delay time, t_i, based on the Unix `srandom()` function that had been seeded with the simulation start time. All simulation results successfully overlay the analysis results. As can be seen, the simulation for $E[t_{min}]$ vs. T matches (Figure 2.19) as does the simulation comparing $E[t_{min}]$ vs. N (Figure 2.20). Similarly, the simulations of $E[\#extra]$ vs. N and $E[\#extra]$ vs. Δ/T validate the analysis (Figures 2.21, 2.22, 2.23).

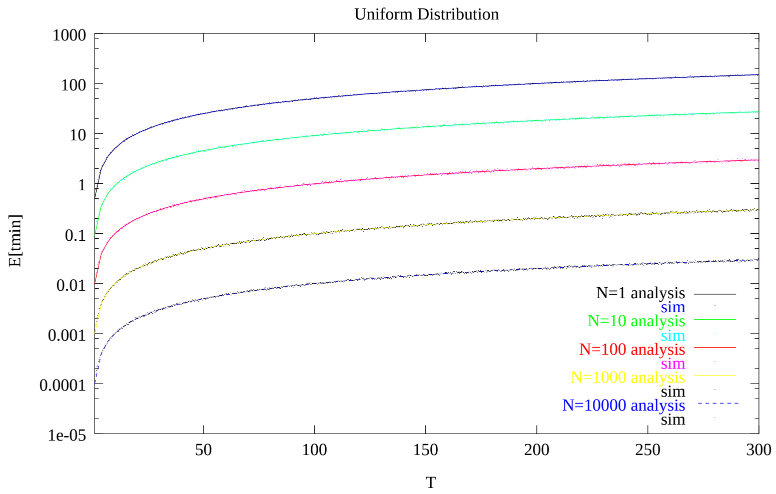

Figure 2.19: **Simulation vs. Analysis: Time Elapsed vs. T (Uniform).**

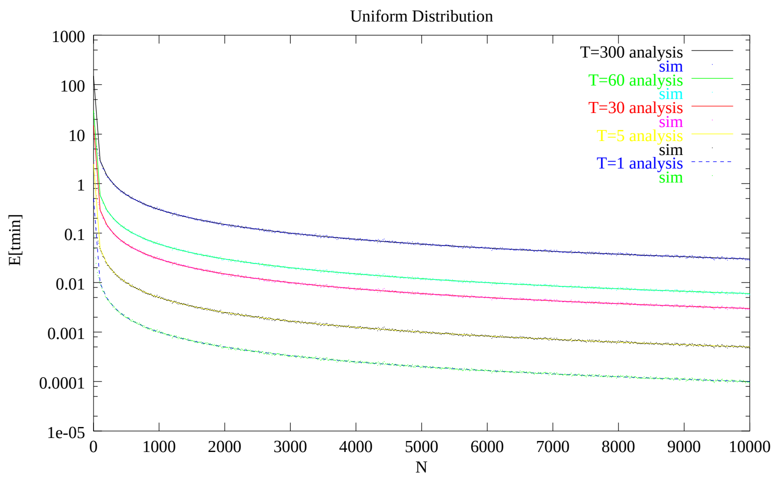

Figure 2.20: **Simulation vs. Analysis: Time Elapsed vs. N (Uniform).**

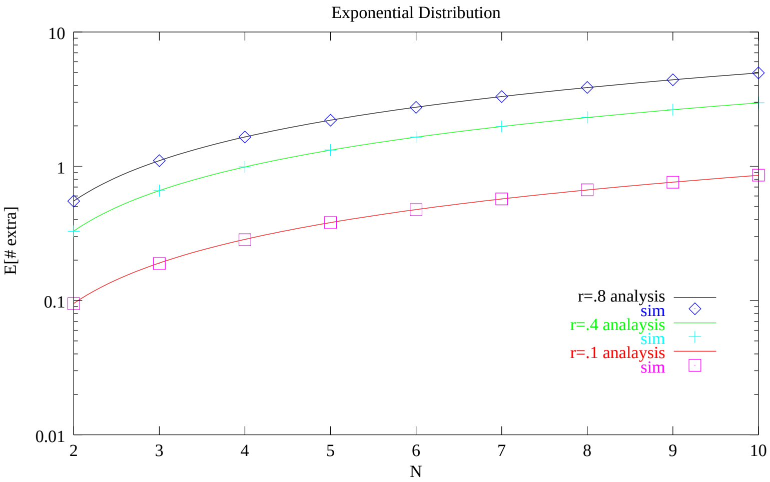

Figure 2.21: **Simulation vs. Analysis: Extra Messages vs.** N **(Large** $r = \Delta/T$**).**

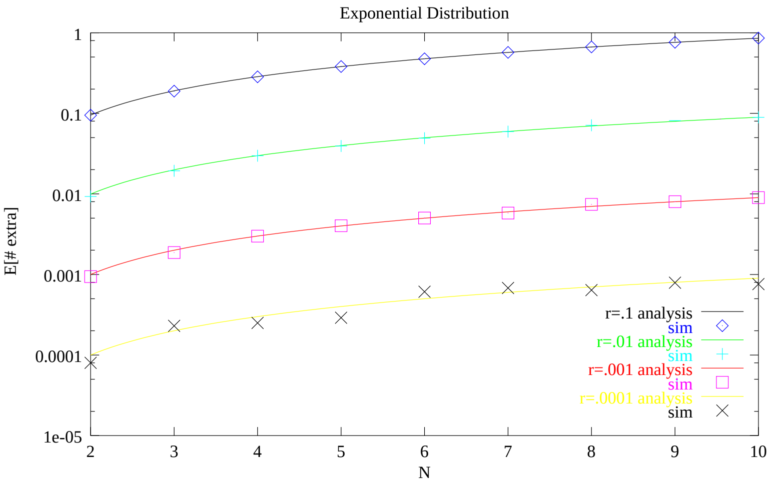

Figure 2.22: **Simulation vs. Analysis: Extra Messages vs.** N **(Small** $r = \Delta/T$**).**

2.7 Related Work

Suppression as a technique to delay messaging has its origin in the packet radio ALOHA protocol [1] and the Ethernet protocol [44]. The protocols that were designed and subsequently optimized for these media share an important characteristic with multicast communication. Namely, all messages sent to the group address are sent to all receivers subscribed to that address. Packet radio and Ethernet, however, are contention networks. Because the physical media are shared among all processes, there is the potential for messages to collide during transmission. Consequently, techniques like random delay are used to help manage bandwidth for collision avoidance and after collision detection (e.g., exponential backoff).

A key difference between Suppression as used with ALOHA versus with multicast is that it is employed after a collision is detected rather than at the beginning of a transmission. Ethernet,

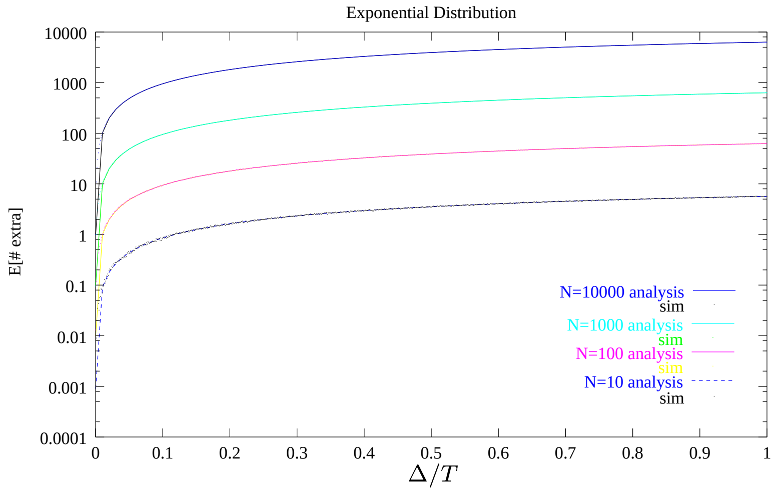

Figure 2.23: **Simulation vs. Analysis: Extra Messages vs. Δ/T (Exponential).**

on the other hand, does rely on processes to listen before sending (e.g., carrier-sense) and thus act accordingly (not send if another process is in the middle of transmitting and reschedule to re-sense the network in a random period of time). A notable attribute of Ethernet is that it can rely on the fact that transmission delay between processes is uniformly low. In fact, for Ethernet the propagation delay is assumed to be small compared to the transmission time to send the message. Propagation delay is also considered identical for all processes. As a result of the physical properties of the networks, the kinds of metrics of interest to Ethernet and ALOHA are the number of collisions generated, the backlog of processes waiting to retransmit, and the channel utilization.

Multicast as a communication technique is merely an abstraction on top of a variety of physical media. Therefore, it is not as controlled an environment and processes do not have the luxury of being able to sense that other processes are in the middle of transmissions. Therefore, its use of Suppression is as a first line of defense, rather than as an action taken after detecting a potential collision. Conversely, because a Multicast process cannot sense the current state of the other processes of the network, until such time as it receives a message, processes do not employ multiple rounds of Suppression for back-off purposes like in Ethernet. However, a strategy similar to back-off has been used by pairs of request-response Suppression algorithms; in contrast to Ethernet, where back-off is used to avoid collisions, it is used with Suppression to ensure successful retransmission of a repair request [27].

There is precedent in studying adaptive algorithms for Suppression. Floyd et al. in their study of SRM [27] propose an adaptive Suppression interval, T. As their work is aimed at Suppression for reliable multicast, there is both a request and a response phase, each of which relies on a separate but related Suppression algorithm with its own pair of timer values. The two phases are tightly coupled in their analysis and simulations. A major difficulty with their approach is that it relies on each process to estimate the distance (the roundtrip delay) between every other process.

Kermode's work on SharqFEC [41] [42] attempts to make the SRM algorithm more scalable by partitioning the group of processes into administratively-scoped multicast zones. This produces more localized regions of control in which delays are estimated between zones, rather than between every pair of processes. Suppression plays a key role in these estimates and once again is employed to reduce message implosion.

The research by Nonnenmacher et al. is most similar to ours, in that they study a series of distributions from an analytic standpoint [45] [46] [47]. Considering their work as a starting point, we attempted to clarify some of their seminal analysis work. We were especially interested in their discovery of the positive, truncated exponential distribution, which has very promising performance characteristics under certain conditions. However, this distribution (and the focus of their work in general) is very firmly set in the context of reliable multicast data transfer. Thus the distribution is optimized for a specific operating point; up to 10^6 group members, minimizing the messaging overhead, and using Suppression in back-to-back mode (as mentioned above) for requesting data retransmission, as well as data replenishment. In contrast, we try to speak more generally about parameter interaction in order that the appropriate operating point can be derived for different applications and different operating environments.

In this work as well as that presented by Nonnenmacher, the uniform distribution is used as a baseline for comparison (since it is most commonly deployed). Yet each presents a different exponential algorithm as an alternative, highlighting the scenarios under which it outperforms the more traditional scheme. Our exponential function is of the form $Ae^{-\alpha t}$, whereas their exponential function is of the form $Ae^{\alpha t}$. The negative form of the function, which is not truncated, spreads the messages over a potentially larger time interval, attempting to avoid unnecessary messaging. The positive form of the function produces a small number of quick responders, with many more slow responders. This behavior is useful for applications that aim to avoid message implosion. As such, it has led us to begin examination of a more extreme example of this type of distribution, which we refer to as a "bin" approach and which we discuss in Section 2.9. The general idea is that the function has two or more distinct regions of operation. For example, with the function proposed by Nonnenmacher et al., in one region the function keeps $E[\#\ extra]$ constant, whereas in the other it grows linearly. The linear part of the function does not dominate until N is large, allowing for $E[\#\ extra]$ to remain small. Hypothetically, it should be possible to design a timer distribution function that optimizes the performance of a particular Suppression metric, given that we know the range of various operating parameters.

Minimizing $E[\#\ extra]$ is a compelling goal for reliable multicast. SRM [27] accomplishes this by using a uniform distribution and adjusting T during each round of Suppression. The work by Nonnemacher et al. [45] aims to eliminate the reliance on an accurate estimate of N, the number of

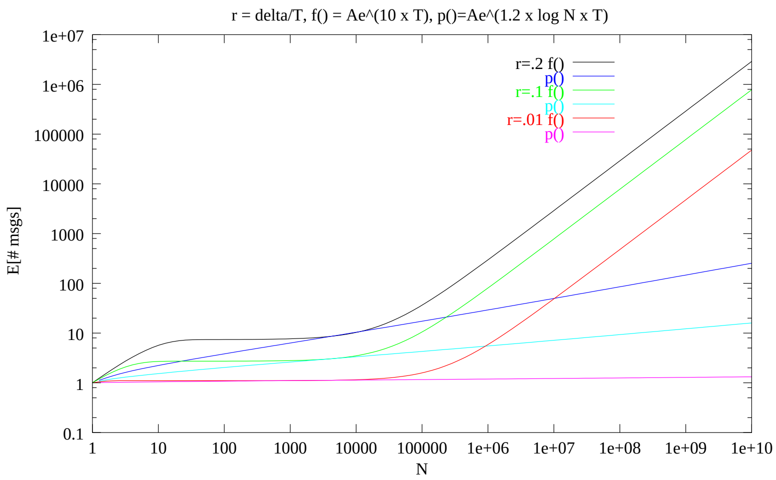

Figure 2.24: **Improvement on Positive, Truncated Exponential.**

participants, and the amount of state required at each receiver. Although the resulting solution of a positive, truncated exponential timer function is a useful alternative, it falls short of providing a general approach. First, the positive form of the function is optimized for a particular range of the operating space, and then only roughly. On closer examination of the α parameter in $Ae^{\alpha t}$, we found that their use of $\alpha = 10$ is only a rough estimate of an optimal value for the range $(10^0 - 10^6)$ over which their function is designed to work. In fact, $\alpha = 10$ is only optimal for 10^4 participants. There are better choices for α for different subranges within the larger range. Were there an application with a more specific operating range for N, the truncated, positive exponential distribution could be further improved by picking a different α. Second, most of the analysis of their function examines Δ/T as ranging from $[.1, 1]$. We have shown that there are important applications where the ratio falls within $[0, .1]$ and needs more complete investigation. Our work attempts to study the full range, or at very least, to study the boundary between large and small ratios. Finally, although their scheme is less reliant on group size N than other schemes, it is criticized in [50] because the timer distribution is actually tuned for an assumed range of N operation, and "once one has this information it might be better used in some local recovery approach rather than using it merely to tune the timer parameters."

This in turn leads to the question of designer functions in general. On cursory examination, we found that by setting $\alpha = 1.2 \times log\ N$, we could create a function that outperformed the original function with regards to $E[\#\ extra]$ (Figure 2.24). Due to the inherent property that there is a tradeoff between minimizing the metric for overhead versus the metric for latency, our designer function not surprisingly performed slightly worse (5%) for the metric $E[t_{min}]$.

2.8 Summary of Results

It is clear from the analysis that there exists an inverse relationship between the metric for messaging overhead, $E[\#\ extra]$, and for messaging delay, $E[t_{min}]$. Unfortunately, gains in one are offset by losses in the other. Hence, the optimal balance will be a function of the operating environment and the application employing the Suppression algorithm.

Our analysis showed and our simulations confirmed that Suppression may not be as effective for very large groups as it is for small or moderate-sized groups because it is difficult to suppress the numbers of extra messages. This is especially true when it becomes difficult to keep Δ/T ratio predictably low (i.e., in the wide-area). We observed that when N is very large, Δ/T must be extremely small to avoid extra messages. Conversely, when N is small, the ratio can afford to be larger.

If the goal is to keep $E[\#\ extra]$ below a given threshold, but N or Δ fluctuate, then T will need to adapt to compensate. Floyd et al [27] propose an adaptive scheme for reliable multicast that employs two back-to-back Suppression algorithms. This scenario has the benefit of a request-reply interaction to estimate the delay between pairs of processes and uses multiple polling rounds for accurate estimation. We corroborate their findings and differentiate our results in Section 2.7.

To our surprise, setting the metrics equal showed that the decaying exponential distribution outperforms the uniform distribution under a number of conditions. When the $E[\#\ extra]$ are equal and N is small, the exponential distribution has smaller elapsed delay $E[t_{min}]$, when the ratio of Δ/T is greater than .1, and with larger T. When $E[t_{min}]$ are equal, fewer extra messages $E[\#\ extra]$ are generated with the exponential distribution when the Δ/T ratio is large and more significantly as N increases.

Directly comparing the results of the uniform and decaying exponential probability density functions suggest that it may make sense to design systems that use one distribution in one part of the operating range, and adapt to use another distribution when in another part of the operating range. This is particularly interesting given that most deployed multicast algorithms using Suppression select random delay timer values from uniform distributions.

In summary, the work presented in this chapter is novel on several counts. Our analysis is aimed at understanding the core Suppression algorithm; a single iteration of it, completely decoupled from any previous or subsequent execution of the algorithm. Our work distinctly does not assume that it is used in conjunction with a second Suppression algorithm, nor that it is used iteratively. We acknowledge that any effective adaptation scheme will rely on multiple iterations in order to accrue history, but that past state information may be obtained through a number of means, some or all of which may be the repetition of the Suppression algorithm.

The study of Suppression as a stand-alone element allows us to understand the effect of composition, when Suppression is combined with other algorithms, including with the Suppression algorithm itself. Our focus has been to identify the parameters that play a role in the outcome of the metrics. Our work examines alternative distribution strategies than ones proposed and/or deployed, as well as presents the underlying theoretical basis for the performance of the algorithm as observed in operation or through simulation. Our study aims to understand the theoretical underpinnings, not of a single parameter, but of each network parameter in isolation, with a goal of improving the parameterization of deployed network protocols as well as those under design.

2.9 Future Work

There are numerous directions in which to take this research. We present a sampling of the possibilities in the sections below: we discuss several ways in which the model itself could be extended to be more sophisticated, we describe a rough approach for the derivation of unknown parameters from observed system behavior, and we propose additional Suppression distributions for investigation.

Refining the Model. Because we make no assumptions about the iterative use of the protocol, we assert that the delay between pairs of processes are fixed over the duration of the algorithm. However, if we relax this constraint, then we can more effectively study the iterative use of the algorithm and its combination with other algorithms (in order to collect past history), in order to support adaptive Suppression strategies.

We have assumed that the delay between pairs of processes is fixed. This assumption is only true under certain conditions, for example, in a local area network, where delay between pairs of processes is characterized as being a fixed Δ within a small amount of variance. The larger the region encompassed topologically by a set of processes, the less valid a fixed-Δ model is likely to be. However, it may still be acceptable to describe a set of processes as having an *effective* Δ, when the set behaves as if Δ were a particular value on average. In particular, we would like to explore three questions regarding transmission delay variance of Δ:

- What are the effects of an *outlier* process, one that has transmission delay considerably larger or smaller than the rest of the group members?
- By how much can the delay vary between pairs of processes before our fixed-Δ analysis is no longer applicable?
- What are the effects of multiple classes of Δ on the behavior of the algorithm, i.e., when there exist subsets of processes within discrete ranges of delay from each other?

In this thesis, we also assume that delays between pairs of processes do not vary over time, as might happen under light versus heavy loads. Our reasoning is that the Suppression algorithm, when used in isolation (versus iteratively) is typically short-lived, meaning that T, the upper bound on the delay interval, is small. In this case, we believe it is still useful to employ the algorithm and to parameterize it to optimize the average expected behavior.

Because Suppression is often employed in combination with other algorithms, many of which perform process synchronization before relying on Suppression, we have deferred modeling asynchronous process arrivals in this chapter. Our rationalization has been that Suppression is often targeted to solve implosion when processes arrive at approximately the same moment. Synchronized process arrivals might occur after a system reboot, after the explicit receipt of a message or at the beginning of a scheduled teleconference. On the other hand, asynchronous process arrivals might occur as group membership fluctuates. The questions then become, when processes arrive randomly into the system, what is the impact on the Suppression metrics proposed, i.e., is Suppression still an effective technique for scalability? While Floyd et al. touch upon this issue when they study the impact of topology on back-to-back Suppression algorithms [27], we would like to reframe the question in terms of delay variance, as applied to the basic Suppression algorithm.

Derivation of N and Other Parameters. It can be difficult to ascertain the group size of a set of communicating processes [51]. If the group size is large, an active method like querying all group members is untenable because of the potential for message implosion. A passive method, such as simply listening and counting the number of processes from which messages are received, is only useful when all processes issue messages (as with Announce-Listen). With the Suppression algorithm, the expectation is that only a very small number of processes will actively issue messages.

Yet the estimation of group size is important for scaling multi-process communication. It is often the case that N is derived through a combination of listening and approximation techniques [30] [29]. Once derived, N is used to bound the amount of bandwidth dedicated to control messaging. The difficulty is ascertaining an accurate estimation quickly, for example immediately at startup or after a large membership fluctuation occurs [52].

In [46], Nonnenmacher suggests using Suppression variables to derive a first approximation for group size estimation in a single round. If the participating processes agree on a set timer distribution, then we can derive N from the latency and overhead characteristics. For example, if the first message is received at $t_{recv} = t_{min} + \Delta$, and all processes use a uniform pdf with a fixed T and Δ, then $N = T/(t_{recv} - \Delta) - 1$. Nonnenmacher actually uses the positive, truncated exponential distribution and assumes processes synchronization through the receipt of a prior message. In [30] [29], Friedman et al. refine the technique, but note that the original distribution and Suppression algorithm lead to bias; above N for low values of N, and below N for higher values. This may be in part due to the usage of an α value that has been optimized for 10^4 participants. It would be interesting to compare the behavior of other distributions for group size estimation and combine them with the improvements cited in [30] [29] (notably refinements to make them robust even when there are heterogeneous delays between processes).

Once N is derived, a process can determine which part of the operating space it is in and could switch between appropriate pdfs, for example, using decaying exponential distribution for small N, and the uniform distribution for large N. Along similar lines, if we obtain N, and know T, then we could derive an estimate of average transmission delay, Δ, by obtaining the number of processes which issue Suppression messages. This might be particularly useful when there are heterogeneous delays between participating group members, but the delay variance is within some bound. The idea would be to use this information to create administratively-scoped multicast groups for groups of processes with similar delay characteristics.

"Two-Component" Uniform Distribution. The realization that uniform and exponential distributions outperform each other in different parts of the parameter space raises the larger question of whether there are other probability distribution functions in need of exploration. We have suggested that systems could benefit by adapting to use one pdf in one part of the parameter space and another outside that range. Another strategy would be to use different pdfs for different classes of processes. For instance, such a strategy might be advantageous when processes can be easily categorized into server vs. client, router vs. end system, high-speed vs. low-speed (link, cpu, memory), leader vs. non-leader. Below, we introduce the two-component uniform distribution as an example of the kind of distribution that can be designed to differentiate between processor classes.

The inspiration behind this distribution is Nonnenmacher's exponential timer distribution function [45], which leads to two operating regions for number of extra messages generated. In one region the function behaves linearly and in the other it remains constant. A generalization of this approach would be to create a small number of regions of the function so different classes of processors would behave differently to the wake-up timer.

The two-component uniform distribution is like a uniform distribution except the pdf is zero for $a \leq t < a + w$. The two-component uniform distribution is the base case for the N-component uniform distribution, the separation of processes into N "bins" of equal width and spacing.

The motivation behind exploring a *bins* theory is that there may be some benefit to separating the group of processes into 2 or more classes. Certain processes will respond quickly and others more slowly, i.e., processes nearby vs. further away, or processes with more CPU power vs. those without. Therefore, the bins serve to distinguish between the different processor classes. The width of the bins and the space between them is critical to spread N processes into the proper bins and ultimately to help reduce the number of processes issuing messages. Multiple bins could be implemented by rounding up or down, so that the value chosen from the uniform distribution falls within the range of one of the bins.

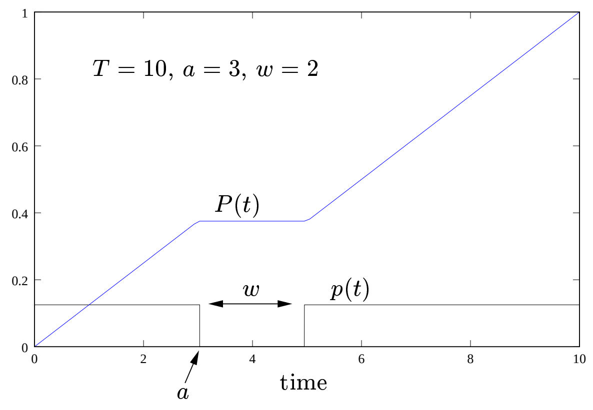

Figure 2.25: **Two-Component Uniform Distribution.**

Due to the complexity of an N-bin distribution, we outline an approach for studying the simplest form of the algorithm when $N = 2$. The two-component uniform distribution has three disjoint ranges. We derive $E[t_{min}]$ by splitting the integral into these three pieces and substituting $P(t)$ for each range.

$$p(t) = \begin{cases} 1/(T-w) & \text{for } 0 \le t < a \\ 0 & \text{for } a \le t < a+w \\ 1/(T-w) & \text{for } a+w \le t < T \end{cases}$$

and

$$P(t) = \begin{cases} t/(T-w) & \text{for } 0 \le t < a \\ a/(T-w) & \text{for } a \le t < a+w \\ (t-w)/(T-w) & \text{for } a+w \le t < T \end{cases}$$

$$\begin{aligned} E[t_{min}] &= \int_0^T (1-P(t))^N \, dt \\ &= \int_0^a (1-t/(T-w))^N \, dt + \int_a^{a+w} (1-a/(T-w))^N \, dt \\ &\quad + \int_{a+w}^T (1-(t-w)/(T-w))^N \, dt \\ &= \frac{T-w}{N+1} + w(1-\frac{a}{T-w})^N \end{aligned}$$

For the rest of the calculation, we assume that $w \ge \Delta$, $\Delta < a$, and $\Delta < T-a-w$. The first assumption corresponds to the case when the width w of the "hole" in the pdf is greater than or equal to Δ. The second and third assumptions mean that the width of the two ranges where the pdf is non-zero are both larger than Δ. We make these assumptions to simplify the integrals.

$$\begin{aligned}
E[\#\ extra] &= N \cdot P(\Delta) - 1 + N \int_{\Delta}^{T} p(t)(1 - P(t-\Delta))^{N-1}\, dt \\
&= N \cdot P(\Delta) - 1 + N \int_{\Delta}^{a} p(t)(1 - P(t-\Delta))^{N-1}\, dt + \\
&\quad N \int_{a}^{a+w} p(t)(1 - P(t-\Delta))^{N-1}\, dt + \\
&\quad N \int_{a+w}^{a+w+\Delta} p(t)(1 - P(t-\Delta))^{N-1}\, dt + \\
&\quad N \int_{a+w+\Delta}^{T} p(t)(1 - P(t-\Delta))^{N-1}\, dt \\
&= \frac{N\Delta}{T-w} - \left(1 - \frac{a-\Delta}{T-w}\right)^{N} + \frac{N\Delta}{T-w}\left(1 - \frac{a}{T-w}\right)^{N-1} + \left(1 - \frac{a+\Delta}{T-w}\right)^{N}
\end{aligned}$$

To get a feel for this expression, let $\alpha = 1 - \frac{a}{T-w}$. We have already assumed $\Delta < T - w - a$. If we further assume that $\Delta \ll T - w - a$,

$$\begin{aligned}
E[\#\ extra] &= \frac{N\Delta}{T-w} + \alpha^N \left(-\left(1 + \frac{\Delta}{T-w-a}\right)^{N} + \frac{N\Delta}{T-w-a} + \left(1 - \frac{\Delta}{T-w-a}\right)^{N} \right) \\
&\leq \frac{N\Delta}{T-w} - \alpha^N \frac{N\Delta}{T-w-a}
\end{aligned}$$

An interesting case is when the gap in the pdf is symmetric, i.e., $a = (T-w)/2$. In this case, $\alpha = 1/2$. Note that for $E[t_{min}]$, the value in parentheses is always non-negative.

$$\begin{aligned}
E[t_{min}] &= \frac{T-w}{N+1} + \frac{w}{2^N} = \frac{T}{N+1} - w\left(\frac{1}{N+1} - \frac{1}{2^N}\right) \\
E[\#\ extra] &\leq \frac{N\Delta}{T-w} - \frac{1}{2^N}\frac{2N\Delta}{T-w} = \frac{N\Delta}{T-w}\left(1 - \frac{1}{2^{N-1}}\right)
\end{aligned}$$

Chapter 3

Suppression with Loss

In this chapter we examine the impact of loss on the model for Suppression that we introduced in Chapter 2. We revisit the metrics defined earlier but re-evaluate them under lossy conditions. In addition, we present several new metrics to predict performance when loss occurs. These include metrics for the effective completion time for the algorithm (Maximum Time Elapsed); the total number of messages generated during the normal course of the algorithm (Number of Messages Generated); and an estimate of how many of the messages that are sent are really necessary (Messages Required) and how many are overhead (Extra Messages with Loss). We discuss several other metrics that impact CPU performance, which are related to the probability of message receipt.

In addition to studying appropriate metrics, this chapter studies the effects of both fully correlated and fully uncorrelated loss. Fully correlated loss occurs when a message is lost closest to the sender, so its loss is experienced by all receivers. With fully uncorrelated loss, a message is lost closest to the receivers, thus the losses experienced by each of the receivers may be different. Most multicast groups will experience loss somewhere between uncorrelated and correlated behavior, so we calculate both as a way to bound the best and worst behavior. Throughout this chapter and the remainder of the thesis, we use the terms correlated and uncorrelated to mean fully correlated and fully uncorrelated, respectively.

Before concluding, we provide an overview of related work, summarize our results, and propose future directions for our research.

3.1 $E[t_{min}]$ Re-visited: Time Elapsed with Loss

Because $E[t_{min}]$ was defined as the earliest time that a message is sent, packet loss does not actually have any impact on the earliest possible time that a node *selects* a suppression wake-up time. However, the message sent by the process that selects the earliest time might be lost by the network. Therefore, different processes participating in the Suppression (SUP) algorithm might be suppressed by different messages. For Suppression with loss, we therefore define $E[t_{min_e}]$, an “effective” $E[t_{min}]$, that we use for comparisons. $E[t_{min_e}]$ is the expected time of the earliest message sent but *not completely dropped* in the network. Therefore, $E[t_{min_e}]$ represents the earliest hope for suppression by a remote process.

Given a particular vector of times $\vec{t} = (t_0, t_1, \ldots, t_{N-1})$, where each t_i is the wake-up time selected by the individual processes, let $\vec{T}_{min} = (t_{min_0}, t_{min_1}, t_{min_2}, \ldots, t_{min_k}, \ldots, t_{min_{N-1}})$ be a permutation of the vector $\vec{t}$ such that $t_{min_i} \leq t_{min_{i+1}}$ for $0 \leq i < N-1$. Thus, $\vec{T}_{min}$ is a function of the t_i sorted in non-decreasing order. Note that the value t_{min_0} is the same as t_{min} (defined in Chapter 2, Section 2.3.1) and $t_{min_{N-1}}$ is the largest time selected by any of the processes.

In Figure 3.1, we display a Suppression interval of length T. Each of N processes selects a time to awaken, t_i, and these are shown ordered from t_{min_0} to $t_{min_{N-1}}$. Under lossy conditions, some subset of the processes will awaken to find that they have not received a message from any other process; these processes, indicated with an `x`, generate a message. The remaining processes, indicated with an `o`, are suppressed.

If messages can be lost, then processes might be suppressed by a process other than the one awakening at t_{min_0}. Furthermore, each process may be suppressed by a different t_{min_i}, since a message received by one process may be lost while in transit to another.

3.1.1 Loss Analysis

We examined both correlated and uncorrelated loss. Simulations showed that correlated loss produces $E[t_{min_e}]$ that is higher than the uncorrelated case. Therefore, we present a detailed analysis of the correlated case as it bounds the performance of the uncorrelated case and the Suppression algorithm in general.

Correlated Loss. Let the probability of message loss be given by l. If the message sent at the earliest selected time t_{min_0} arrives successfully, it represents the earliest message to arrive at process i that might suppress it. Successful arrival happens with probability $(1-l)$. If instead the earliest message is lost, we examine the message sent by the process with the next earliest time t_{min_1}. If that message is lost, we examine the next time selected t_{min_2}, and so forth. We assume that not all messages are lost. Thus, the expected time of the earliest message sent to process i taking loss into account is given below. The normalization factor $1/(1-l^N)$ comes from the assumption that not all messages are lost.

$$
\begin{aligned}
E[t_{min_e}] &= E\left[\frac{((1-l)t_{min_0} + (1-l)lt_{min_1} + \cdots + (1-l)l^k t_{min_k} + \cdots + (1-l)l^{N-1}t_{min_{N-1}})}{(1-l^N)}\right] \\
&= \frac{(1-l)}{(1-l^N)} E\left[\sum_{0 \le k < N} l^k \cdot t_{min_k}\right]
\end{aligned}
$$

We need to calculate $E[t_{min_k}]$, the expected value of the kth smallest time given a randomly chosen vector of N times. For a particular time t_i to be the kth smallest, there must be exactly k times smaller than it, and $N-1-k$ times larger than it (note that indices begin at 0). The probability of this event is $\binom{N-1}{k} P(t_i)^k (1-P(t_i))^{N-1-k}$. Therefore, the expected value is given by:

$$
E[t_{min_k}] = \int_0^T Ntp(t)\binom{N-1}{k} P(t)^k (1-P(t))^{N-k-1}\, dt
$$

Substituting $E[t_{min_k}]$ into the equation for $E[t_{min_e}]$, we derive the formula below. We use the binomial expansion to simplify the equation and let $Q(t) = (1-l)P(t), q(t) = (1-l)p(t)$, where $q(t) = dQ/dt$. Note that the resulting formula is similar in form to the expression for Suppression without loss, and when $l = 0$ the formula is identical to the expression derived in Section 2.3.1.

$$
\begin{aligned}
E[t_{min_e}] &= \frac{(1-l)}{(1-l^N)} \sum_{0 \le k < N} l^k \int_0^T Ntp(t)\binom{N-1}{k} P(t)^k (1-P(t))^{N-k-1}\, dt \\
&= \frac{(1-l)}{(1-l^N)} \int_0^T Ntp(t)(lP(t) + (1-P(t)))^{N-1}\, dt \\
&= \frac{-Tl^N}{(1-l^N)} + \frac{1}{(1-l^N)} \int_0^T (1-Q(t))^N\, dt \\
&= \frac{-Tl^N + \int_0^T (1-Q(t))^N\, dt}{(1-l^N)}
\end{aligned}
$$

3.2 Maximum Time Elapsed

It is common for the Suppression algorithm to be characterized in terms of $E[t_{min}]$, i.e., when the algorithm sends its first message. However, it is equally useful, particularly with message loss, to ask how long it takes for the algorithm to complete. Thus we define $E[t_{max}]$, when the algorithm sends its last message. Completion time is important when Suppression is followed by another algorithm, which is often the case.

The value $E[t_{max}]$ is defined as the expected time selected by the last process that actually generates a message (Figure 3.1). This is appreciably different than the maximum t_i selected by all processes. In other words, $t_{max} \neq t_{min_{N-1}}$, and when $l \neq 1$ we would expect that $t_{max} < t_{min_{N-1}}$.

In the lossless case, $E[t_{min}] \le E[t_{max}] \le E[t_{min}] + \Delta$, meaning the last message is sent within

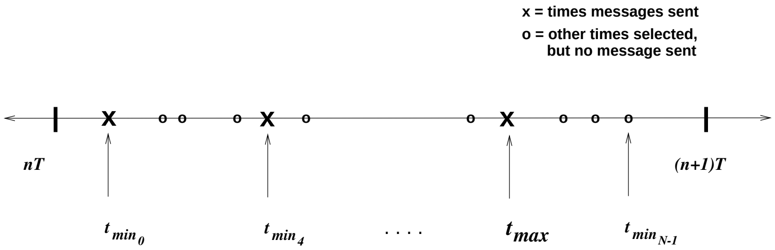

Figure 3.1: t_{max}**: The Maximum** t_{min_i} **Sent.**

Δ of the earliest message. However, with loss, multiple messages may be necessary to suppress a group of processes. Thus, the time of the last message sent, $E[t_{max}]$, is no longer defined in terms of the time of the earliest message sent, $E[t_{min}]$. In either case, $E[t_{max}] + \Delta$ can be thought of as delineating a point in time beyond which all processes are considered suppressed, i.e., the expected time after which all nodes are in agreement to halt the algorithm.

3.2.1 General Form

Consider $Pr[t_{max} = t_{min_k}]$, the probability that the maximum time elapsed equals the k^{th} value in the vector $\vec{T}_{min}$. The likelihood that t_{min_k} is the last suppression message actually sent is the probability that the k^{th} process does not receive a message from any earlier processes and that all later processes receive at least one message from one of the earlier processes or the k^{th} process. A process k may not receive a message from another process for one of several reasons:

- the other process never sent a message because it was suppressed,
- the message was sent but was lost, or
- because the message arrived late (it was generated after $t_{min_k} - \Delta$).

In general, we can write

$$E[t_{max}] \quad = \quad \sum_{0 \leq k < N} t_{min_k} \times Pr[t_{max} = t_{min_k}]$$

3.2.2 Zero Delay

The calculation of $E[t_{max}]$ is subtle since the event $(t_{max} = t_{min_k})$ consists of two parts that are interdependent: process k must lose all messages sent before it, and every process after k must not lose all the previous messages from processes $\{0, \ldots, k\}$. These parts are dependent because they both rely on particular messages actually being sent (or not) by processes $\{0, \ldots, k-1\}$.

Furthermore, the calculation of $E[t_{max}]$ is complicated by the combined effects of both message loss and delay. We attempt to differentiate between their effects on $E[t_{max}]$ by understanding how

Event	*Probability*
$t_{max} = t_0$	$(1-l)^2$
$t_{max} = t_1$	$l(1-l^2)$
$t_{max} = t_2$	$l((1-l)+l^2)$

Table 3.1: **Zero-Delay Event Probabilities.**

the system behaves when each is set to 0. In Chapter 2, we studied Suppression with zero loss. Here we isolate the effect of loss by setting the transmission delay to as close to zero as permitted by the network simulator.[1] The result of setting Δ as close to zero as possible (and also very small relative to T) is that communication with other processes happens virtually instantaneously. This makes it statistically improbable that a message is generated within Δ of other messages and thus we can observe the impact of loss on the algorithm.

Consider a three node example that illustrates the difficulties of modeling $Pr[t_{max} = t_{min_k}]$. For simplicity let us assume that $\vec{T}_{min} = \vec{t}$, that the processes $0, 1, 2$ select wake-up times t_i in ascending order. Let us also assume that there is zero delay in the network.

$\mathbf{t_{max} = t_0}$**.** Process 0 generates the last message sent if message 0 was received by processes 1 and 2. Each of these events occurs with probability $(1-l)$. Therefore, $Pr[t_{max} = t_0] = (1-l)^2$.

$\mathbf{t_{max} = t_1}$**.** Process 1 generates the last message sent if message 0 was not received by process 1 (which happens with probability l), and either message 0 was received by process 2 or message 0 was lost by process 2 and message 1 was received by process 2 (which is equivalent to saying process 2 did not lose both messages sent, $(1-l^2)$). Therefore, $Pr[t_{max} = t_1] = l(1-l^2)$.

$\mathbf{t_{max} = t_2}$**.** Process 2 generates the last message if process 2 lost message 0 (occurring with probability l), and process 1 never sent a message (because it received the message from process 0, with probability $(1-l)$) or process 1 sent a message (because it lost the message from process 0, with probability l) and it was lost by process 2 (which occurs with probability l). Therefore, $Pr[t_{max} = t_2] = l((1-l)+l^2))$.

The probabilities for these events are summarized in Table 3.1. The example highlights that we must take into account that some messages are *never sent due to having been suppressed previously!* In contrast, the process that selects the minimum time *always* sends its message. We must be careful not to assume when process k loses all previous messages that all k messages were actually generated; some of them may have been suppressed by other messages. In fact, there will be $1 < i \leq k$ messages generated. Moreover, when we consider that all processes beyond process k do not lose

[1] It is impossible in `ns` to have zero delay, so we characterize it as approximately zero. Although we set link delay to 0, there is a small amount of time consumed for switching delay through nodes. We offset this by making the T interval sufficiently large to mask the effects.

all the messages sent, we must be careful not to double count the loss by assuming that all $k+1$ messages were generated.

Simulation results showed that uncorrelated loss produces $E[t_{max}]$ that is higher than the correlated case. Therefore, we present a detailed analysis of the uncorrelated case as it serves as an upper bound on the performance of the algorithm.

Uncorrelated Loss. Let us derive $Pr[t_{max} = t_{min_k}]$ more precisely in the case where there is zero transmission delay in the network. t_{max} is equal to t_{min_k} when process k loses all messages that were sent from $0 \leq j < k$, and processes $k+1 \leq j < N-1$ receive at least one message from processes $0 \leq j \leq k$ that generate messages (Figure 3.2). This can be decomposed into a set of disjoint events, each event being the case when processes in the range $0 \leq j < k$ generate i messages, where i ranges from 1 to k.

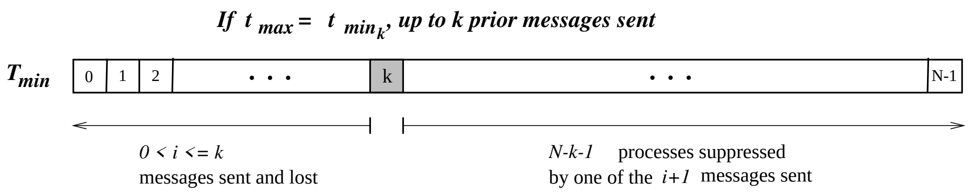

Figure 3.2: $Pr[t_{max} = t_{min_k},\ given\ i\ messages\ sent]$.

Given that exactly i messages were sent, process k must have lost i messages, which occurs with probability l^i. Additionally, all $N-k-1$ processes with later wake-up times than process k each must have been suppressed by an earlier process, meaning that each did not lose all of the $i+1$ previous messages sent. Also, the probability that $t_{max} = t_{min_0} = t_{min}$ is equal to the likelihood that all subsequent $N-1$ processes receive the first message sent. These observations are summarized by the following equations:

$$
\begin{aligned}
Pr[t_{max} = t_{min_k}]_{\Delta=0} &= \sum_{1 \leq i \leq k} Pr[exactly\ i\ messages\ sent] \times l^i (1 - l^{i+1})^{N-k-1} \\
Pr[t_{max} = t_{min_0}]_{\Delta=0} &= (1-l)^{N-1}
\end{aligned}
$$

We now determine the probability that exactly i messages are sent by processes 0 to $k-1$. Let $P(i,n)$ be the probability that exactly i messages are sent by n processes. Note that the probability $Pr[exactly\ i\ messages\ sent]$ is simply $P(i,k)$.

$P(i,n)$ can be broken down into two disjoint parts: (a) Process $n-1$ sends a message, and processes $0 \leq j < n-1$ send $(i-1)$ messages; (b) Process $n-1$ does not send a message, and processes $0 \leq j < n-1$ send i messages. In case (a), process $n-1$ must lose the $(i-1)$ messages sent, which happens with probability l^{i-1}. In case (b), process $n-1$ must receive at least one of the i messages sent, which happens with probability $1-l^i$. Therefore, we can write:

$$\begin{aligned} P(i,n) &= Pr[exactly\ i\ messages\ sent\ by\ n\ processes] \\ &= P(i-1,n-1) \times l^{i-1} + P(i,n-1) \times (1-l^i) \end{aligned}$$

Since the process with the earliest selected time always sends a message, $P(0,n) = 0$ for $n > 0$. We can also compute $P(i,i)$, because this is the likelihood that each process sends a message, i.e., each process loses all messages sent by any earlier process. Therefore,

$$\begin{aligned} P(0,n) &= 0 \\ P(i,i) &= l^1 \cdot l^2 \cdot l^3 \cdots l^{i-1} \\ &= l^{(i-1)i/2} \end{aligned}$$

Thus,

$$\begin{aligned} E[t_{max}]_{\Delta=0} &= \sum_{0 \le k < N} t_{min_k} \times Pr[t_{max} = t_{min_k}]_{\Delta=0} \\ &= \sum_{0 \le k < N} t_{min_k} \times \sum_{0 \le i \le k} Pr[exactly\ i\ messages\ sent] \times l^i(1-l^{i+1})^{N-k-1} \\ &= \sum_{0 \le k < N} \int_0^T Ntp(t)\binom{N-1}{k} P(t)^k (1-P(t))^{N-k-1}\, dt \times \\ &\quad \sum_{0 \le i \le k} P(i,k) \times l^i(1-l^{i+1})^{N-k-1} \end{aligned}$$

This formula matches our intuition. When there is no message loss in the network ($l = 0$), $E[t_{max}]_{\Delta=0} = t_{min_0}$, and when all messages are dropped ($l = 1$), $E[t_{max}]_{\Delta=0} = t_{min_{N-1}}$.

3.3 Number of Messages Generated

We define the metric $E[\#\ messages]$ as the total number of messages that are generated by the algorithm. In the lossless case, this metric is simply $E[\#\ messages] = E[\#\ extra] + 1$, as the algorithm generates only one useful message and the rest extra messages. Given the derivation for $E[\#\ extra]$ from Chapter 2, $E[\#\ messages]$ in the lossless case is:

$$\begin{aligned} E[\#\ messages]_{l=0} &= 1 + E[\#\ extra]_{l=0} \\ &= N \cdot P(\Delta) + N \int_\Delta^T p(t)(1-P(t-\Delta))^{N-1}\, dt \end{aligned}$$

In the lossy case, multiple messages may perform suppression by affecting different subgroups of processes, thus we need to revise how $E[\#\ messages]$ is calculated. If we can determine the probability that i messages are sent, then:

$$E[\#\ messages] \quad = \quad \sum_{1 \le i \le N} i \times Pr[exactly\ i\ messages\ sent]$$

Uncorrelated Loss. Simulations conclusively show that uncorrelated loss leads to higher message generation. Thus we present an analysis of the expected number of messages generated in the uncorrelated case. From Section 3.2.2, we know the probability that i messages are sent in the zero-delay case, and therefore $E[\#\ messages]_{\Delta=0}$ is:

$$\begin{aligned} E[\#\ messages]_{\Delta=0} &= \sum_{1 \le i \le N} i \times Pr[exactly\ i\ messages\ sent] \\ &= \sum_{1 \le i \le N} i \times P(i, N) \end{aligned}$$

Although we do not present an analytic solution for $E[\#\ messages]$, we offer $E[\#\ messages]_{\Delta=0}$ as an approximation.

3.4 Number of Messages Required

We also define the average number of useful messages, $E[\#\ required]$. We can think of $E[\#\ required]$ as being the number of messages required to fully suppress a group of size N. This is equivalent to the number of messages generated when there exists no transmission delay. This assertion holds true because if all messages are received instantaneously, there are **no** extra messages generated due to transmission delay and all messages are therefore necessary for the Suppression algorithm to work properly. The definition for $E[\#\ required]_{l=p,\Delta=d}$ becomes:

$$E[\#\ required]_{l=p,\Delta=d} \quad = \quad E[\#\ messages]_{l=p,\Delta=0}$$

The implication is that regardless of the value of Δ, the number of required messages remains constant for a given level of loss.

Uncorrelated Loss. Uncorrelated loss leads to higher numbers of required messages. The analysis for $E[\#\ required]$ is identical to and follows from the previous section:

$$\begin{aligned} E[\#\ required]_{l=p,\Delta=d} &= E[\#\ messages]_{l=p,\Delta=0} \\ &= \sum_{1 \le i \le N} i \times Pr[exactly\ i\ messages\ sent] \end{aligned}$$

$$= \sum_{1 \leq i \leq N} i \times P(i, N)$$

3.5 $E[\#\ extra]$ Re-visited: Extra Messages with Loss

Whereas $E[\#\ messages]$ characterizes the network traffic and $E[\#\ required]$ the necessary traffic, $E[\#\ extra]$ can be thought of as network overhead in the system.

When there is no transmission delay and no loss, $E[\#\ extra]$ equals zero and only the earliest message is generated. When loss is present, but there is no delay, $E[\#\ extra]$ still equals zero; we know from Section 3.4 that all messages generated are required messages. When delay is introduced between processes, but there is no message loss, $E[\#\ extra]$ is calculated as the number of processes for which t_i falls within Δ of $E[t_{min}]$ (Chapter 2). The first message sent is "useful" in that it is the only one that suppresses processes in the system; the remaining messages are redundant. These extra messages result due to transmission delay and are mistakenly sent due to an earlier message not arriving in time.

With delay and loss present, the calculation of $E[\#\ extra]$ is more subtle. $E[\#\ extra]$ still reflects the number of messages that are sent due to transmission delay of the earliest message sent. However, it also captures the number of messages that are within Δ of any other processes generating messages, rather than strictly within Δ of the minimum time selected. Thus, the interplay between transmission delay and loss contributes to $E[\#\ extra]$, as the impact of Δ and l are no longer separable. For example, a message generated by a process is considered extra if it is received but does not suppress any other process **and** if after it is sent the local process receives a message with an earlier t_i.

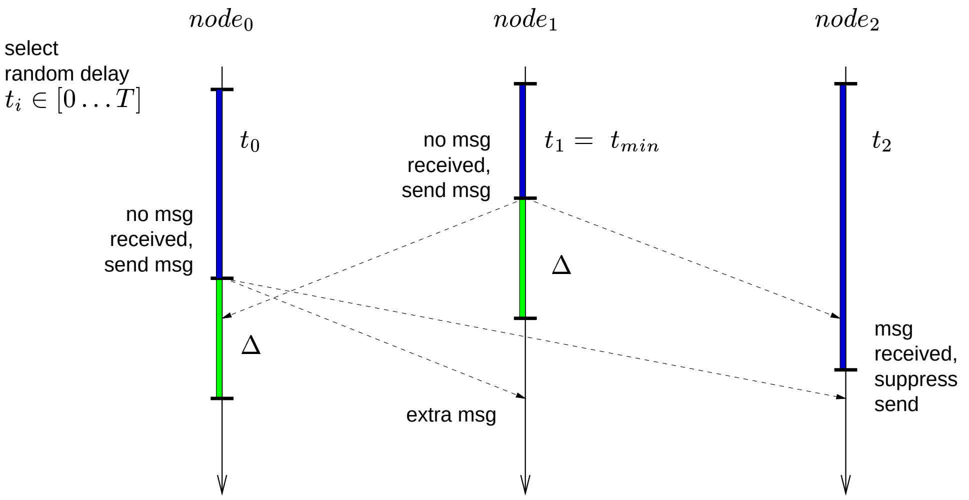

Figure 3.3: **Suppression Algorithm.**

For example, consider the three processes depicted in Figure 3.3. Assume that $T_{min} = (t_1, t_0, t_2)$.

If t_1 is within Δ of t_0, then process 0 does not receive message 1 in time to avoid sending its message. If process 1 actually receives the message from process 0, then message 0 is considered an extra message – because process 0 does not suppress other processes and it eventually receives an earlier message than its own. However, if the message from process 1 was dropped before it reached process 0, then message 0 is **not** considered extra. Note that the number of messages is a count of the extra messages sent (vs. received). Although there are N messages received, there is only 1 message generated. Copies are made at the branching points in the network.

Correlated Loss. Simulations show that correlated loss leads to many more extra messages than uncorrelated loss. This is in part a consequence of uncorrelated loss resulting in higher $E[\#\ messages]$ and $E[\#\ required]$. It is also due to the fact that for correlated loss, all messages up to and including the first successfully delivered message are considered required (as all processes lose or receive the same messages), and any successfully delivered messages within Δ of the first successfully delivered message are classified as extra.

Because of the complexity of deriving $E[\#\ extra]$, we simply relate it to $E[\#\ messages]$. We can determine the impact of a given transmission delay, $\Delta = d$, on the algorithm overhead, by subtracting the number of required messages generated when $\Delta = d$ from the total number of messages generated when $\Delta = d$. We assume all other parameters are kept constant.

$$\begin{aligned} E[\#\ extra]_{l=p,\Delta=d} &= E[\#\ messages]_{l=p,\Delta=d} - E[\#\ required]_{l=p,\Delta=d} \\ &= E[\#\ messages]_{l=p,\Delta=d} - E[\#\ messages]_{l=p,\Delta=0} \end{aligned}$$

We have already derived $E[\#\ messages]_{l=p,\Delta=0}$ analytically in the uncorrelated case. Here we present the analysis for the correlated case:

$$\begin{aligned} E[\#\ required] &= \sum_{1\le k\le N} k \times Pr[k\ messages\ sent] \\ &= \sum_{1\le k\le N} k \times (1-l)l^{k-1} \\ &= (1-l) \sum_{1\le k\le N} k \times l^{k-1} \\ &= \frac{l(1-l^N)}{(1-l)} - N\ l^N \end{aligned}$$

In other words, the probability that k messages are sent is the likelihood that $k-1$ messages were lost and the k^{th} message was successfully received.

As we have not derived an expression for the more general $E[\#\ messages]_{l=p,\Delta=d}$, to find $E[\#\ extra]_{\Delta=d}$, we subtract the analytic results for $E[\#\ required]_{\Delta=d}$ from the simulated results for $E[\#\ messages]_{\Delta=d}$.

3.6 Other Metrics

There are two related metrics that measure the impact of the reception of messages, but which we do not model here. The first is $E[\#\ messages\ received]$, the number of messages actually *received* on average. With message loss, the number received may be less than the number generated or sent. In the correlated loss case, all receivers will experience the same number of messages received. However, in the uncorrelated loss case, $E[\#\ messages\ received]$ may be different on a per node basis. Therefore, it becomes an averaged quantity. In effect, $E[\#\ messages\ received]$ gauges CPU resources used, as it represents the average number of interrupts a node receives. We also define $E[\#\ extra\ received]$, the number of extra messages received, which measures the CPU processing overhead experienced by the system. It too becomes an averaged metric in the uncorrelated loss scenario. On a per node basis, $E[\#\ extra\ received] = E[\#\ messages\ received] - 1$, assuming that one's own Suppression message is included in the number of messages received.

3.7 Distributions

Below, we examine the following probability distribution functions: a uniform distribution, and a decaying exponential distribution. We calculate the bounds for metrics derived in the previous sections of this chapter: $E[t_{min_e}]$, the effective minimum delay; $E[t_{min_k}]$, the kth smallest wake-up time; $E[t_{max}]_{\Delta=0}$, the maximum time elapsed; and $E[\#\ messages]_{\Delta=0}$, the total number of messages generated.

3.7.1 Uniform Distribution

For this case, $p(t) = 1/T$, and $P(t) = t/T$. Let $c = (1-l)$ and $Q(t) = cP(t)$. Then $q(t) = cp(t)$ and $q(t) = dQ/dt$.

$$
\begin{aligned}
E[t_{min_e}] &= \frac{-Tl^N + \int_0^T (1-Q(t))^N\,dt}{(1-l^N)} \\
&= \frac{-Tl^N + \int_0^T (1-cP(t))^N\,dt}{(1-l^N)} \\
&= \frac{T}{(N+1)}\left[\frac{1}{(1-l^N)}\left(\frac{(1-l^{N+1})}{(1-l)} - (N+1)(l^N)\right)\right]
\end{aligned}
$$

$$
\begin{aligned}
E[t_{min_k}] &= \int_0^T Ntp(t)\binom{N-1}{k}P(t)^k\left(1-\left(\frac{t}{T}\right)\right)^{N-k-1}dt \\
&= \int_0^T N\binom{N-1}{k}\left(\frac{t}{T}\right)^{k+1}\left(1-\left(\frac{t}{T}\right)\right)^{N-k-1}dt
\end{aligned}
$$

$$= \frac{T(k+1)}{(N+1)}$$

$$\begin{aligned}
E[t_{max}]_{\Delta=0} &= \sum_{0\le k<N} t_{min_k} \times Pr[t_{max} = t_{min_k}]_{\Delta=0} \\
&= \sum_{0\le k<N} t_{min_k} \times \sum_{0\le i\le k} Pr[exactly\ i\ messages\ sent] \times l^i(1-l^{i+1})^{N-k-1} \\
&= \sum_{0\le k<N} \frac{T(k+1)}{(N+1)} \times \sum_{0\le i\le k} P(i,k) \times l^i(1-l^{i+1})^{N-k-1}
\end{aligned}$$

$$\begin{aligned}
E[\#\ messages]_{\Delta=0} &= \sum_{1\le i\le N} i \times Pr[exactly\ i\ messages\ sent] \\
&= \sum_{1\le i\le N} i \times P(i,N)
\end{aligned}$$

3.7.2 Decaying Exponential Distribution

For this case, $p(t) = e^{-t/\alpha}/\alpha$, $P(t) = 1 - e^{-t/\alpha}$. Let $c = (1-l)$ and $Q(t) = cP(t)$. Then $q(t) = cp(t)$ and $q(t) = dQ/dt$.

$$\begin{aligned}
E[t_{min_e}] &= \frac{-Tl^N + \int_0^\infty (1-Q(t))^N\,dt}{(1-l^N)} \\
&= \frac{-Tl^N + \int_0^\infty (1-cP(t))^N\,dt}{(1-l^N)} \\
&= \frac{\alpha}{N}\left[\frac{-Tl^N N}{(1-l^N)\alpha} + \frac{(1-l)^N}{(1-l^N)}\right]
\end{aligned}$$

$$\begin{aligned}
E[t_{min_k}] &= \int_0^\infty Ntp(t)\binom{N-1}{k} P(t)^k (1-P(t))^{N-k-1}\,dt \\
&= \int_0^\infty \frac{Nt}{\alpha}\binom{N-1}{k}\left(1-e^{-t/\alpha}\right)^k e^{-t(N-k)/\alpha}\,dt
\end{aligned}$$

While we are unable to produce a closed form solution to the integral for $E[t_{min_k}]$, in Section 3.8 we numerically integrate the formula and display results for when N is an integer.

$$\begin{aligned}
E[t_{max}]_{\Delta=0} &= \sum_{0\le k<N} t_{min_k} \times Pr[t_{max} = t_{min_k}]_{\Delta=0} \\
&= \sum_{0\le k<N} t_{min_k} \times \sum_{0\le i\le k} Pr[exactly\ i\ messages\ sent] \times l^i(1-l^{i+1})^{N-k-1} \\
&= \sum_{0\le k<N} t_{min_k} \times \sum_{0\le i\le k} P(i,k) \times l^i(1-l^{i+1})^{N-k-1}
\end{aligned}$$

Metric	*Description*
t_{min}	earliest time sent
t_{minr}	earliest time sent and received, t_{min_e}
t_{max}	latest time sent
$avg\ t_{min}$	average earliest time received
$avg\ t_{max}$	average latest time received

Table 3.2: **Time Metrics Simulated.**

$$\begin{aligned} E[\#\ messages]_{\Delta=0} &= \sum_{1\leq i\leq N} i \times Pr[exactly\ i\ messages\ sent] \\ &= \sum_{1\leq i\leq N} i \times P(i,N) \end{aligned}$$

3.8 Analysis and Simulation

In our initial simulations, N, T and Δ were fixed. Each simulation was run 1000 times when $N \leq 100$, 100 times when $100 < N \leq 500$, and 20 times when $500 < N \leq 1000$.[2] Each node chose a delay time, t_i, based on the Unix `srandom()` function that had been seeded with the simulation start time. The simulations specifically explored the behavior of the Suppression algorithm with increasing packet loss probabilities, ranging from 0 to 1 in increments of 0.1.

Below, we compare performance along several axes. First we study the impact of loss on the various performance metrics; time and messaging overhead. We follow this with an analysis of the differences between correlated and uncorrelated loss. In addition, we contrast the uniform versus exponential distributions, expanding upon the results in Chapter 2.

3.8.1 Time Metrics

In Table 3.2 we summarize the values we tracked in our simulations for the Suppression with Loss algorithm. We were interested in definitive values for the earliest (t_{min}) and latest (t_{max}) messages sent, as well as the earliest message both sent and received ($t_{minr} = t_{min_e}$). In addition, we tracked the average values ($avg\ t_{min}$ and $avg\ t_{max}$) across all processes, since uncorrelated loss causes different processes to receive different messages.

Minimum Time Elapsed. Uncorrelated loss leads to smaller simulated t_{minr} (t_{min_e} in our analysis) than correlated loss for small N (Figure 3.4). However, the values converge with large N, and the point of convergence shifts upward (in N) as l increases (Figure 3.5). Simulations show that the uncorrelated t_{minr} values are identical to t_{min} in the lossless case, up to a large percentage of loss

[2] For the simulation variable t_{minr}, which identifies the earliest time of a message both sent and received, and which is discussed in the next section, the normalization factor was the number of iterations where not all messages were lost.

(Figure 3.6). This suggests that there is always one process within the group of participants that receives the smallest time selected, other than the sender of the message. However, once loss levels are extremely high, the group size N must be higher to counteract the effects of message loss.

In the correlated loss case, the analytic results for t_{min_e} match the simulation results for t_{minr}, which is evident in Figures 3.4, 3.5, and 3.6. Thus t_{min_e} serves as a good upper bound on the behavior of t_{minr}.

Nonetheless, the averaged values ($avg\ t_{min}$) are very close for uncorrelated and correlated loss. For uncorrelated loss, $avg\ t_{min}$ slightly underestimates t_{min_e} for small N, and overestimates it for large N.

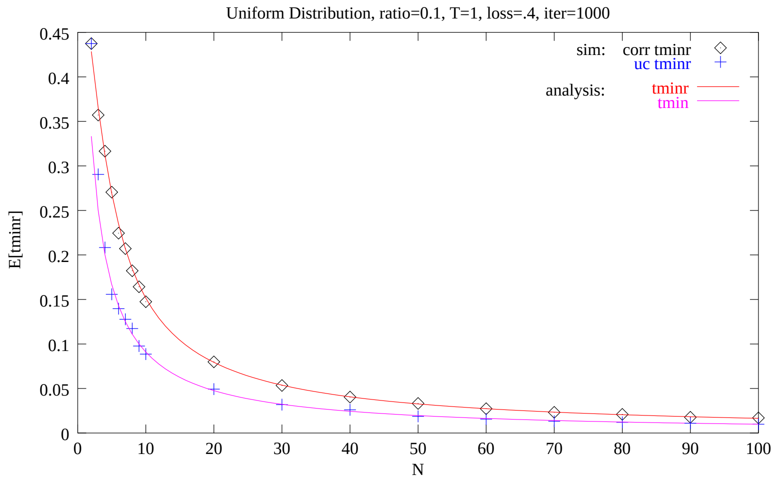

Figure 3.4: $E[t_{minr}]$ **vs.** N **(**$l = .4$**): Correlated vs. Uncorrelated.**

Maximum Time Elapsed. As the number of processes N increases, the value t_{max} becomes asymptotic; above a certain number of processes the same $E[t_{max}]$ (or $E[avg\ t_{max}]$) results (Figure 3.7). Not surprisingly, as packet loss l increases, the point at which the curves flatten out increases, i.e., it requires the participation of many more processes before the asymptote is reached. In effect, greater numbers of N are required to offset the impact of greater l.

For the parameters selected, the asymptotic value begins at $\Delta = .1$ for all t_{max} values (average, definitive, correlated and uncorrelated), and increases as l increases. As expected, when there is little loss, the message with the minimum time will reach and suppress most other processes, except for those selecting a t_i within Δ of the minimum time. As more messages are dropped, fewer processes receive the message containing the minimum time, thus resulting in additional messages generated, which results in the last message being sent further out in time.

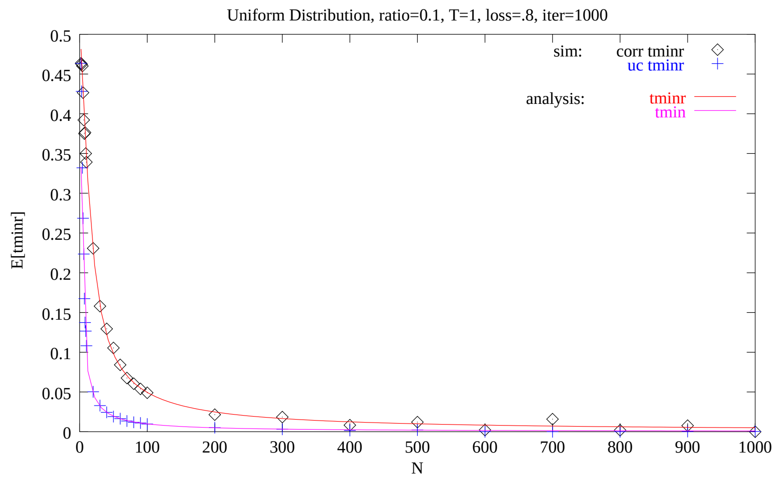

Figure 3.5: $E[t_{minr}]$ **vs.** N $(l = .8)$**: Convergence.**

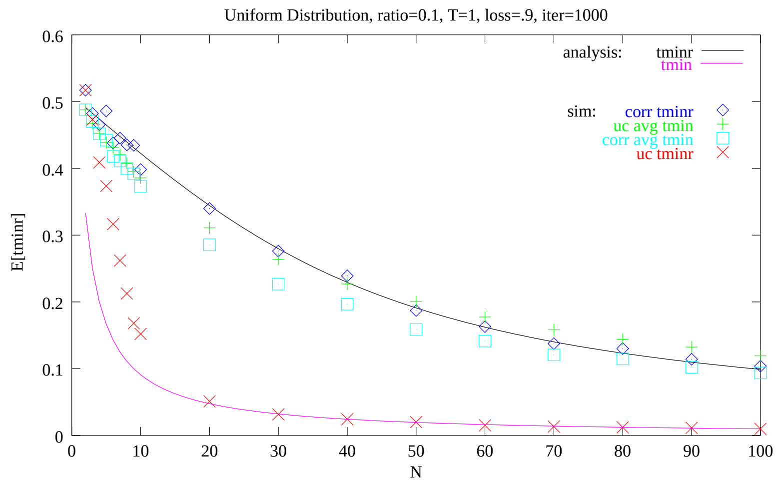

Figure 3.6: $E[t_{minr}]$ **vs.** N $(l = .9)$**: Large Loss.**

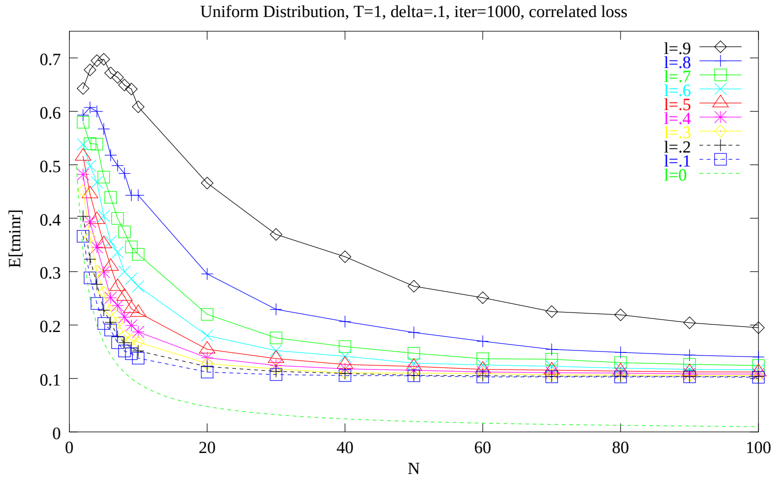

Figure 3.7: $E[t_{minr}]$ **vs.** N**: Varying** l**.**

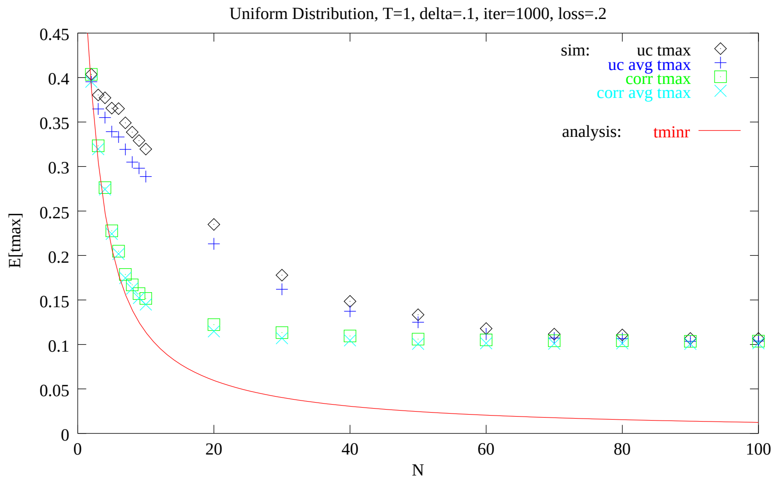

Figure 3.8: $E[avg\ t_{max}]$ **and** $E[tminr]$**: Small Loss (**$l = .2$**).**

As packet loss increases, values of t_{max} and $avg\ t_{max}$ diverge, with average values being lower than definitive values and correlated loss values being lower than uncorrelated loss values (Figure 3.8). At loss rates of $l \geq .5$ (Figure 3.9), the average and definitive values for uncorrelated loss begin to diverge significantly.

In the uncorrelated case, the shape of the t_{max} curve changes substantially when the message loss l is high. The value of t_{max} increases at first, plateaus, and eventually decreases again to reach an asymptotic value. To explain this phenomenon we consider the probability that there exists a *straggler* process. We define a straggler as a process that loses all messages sent except its own. Consider the scenario where all processes send messages. Let l equal the probability that a message is lost and $N-1$ equal the number of other processes (besides self). The probability that a process

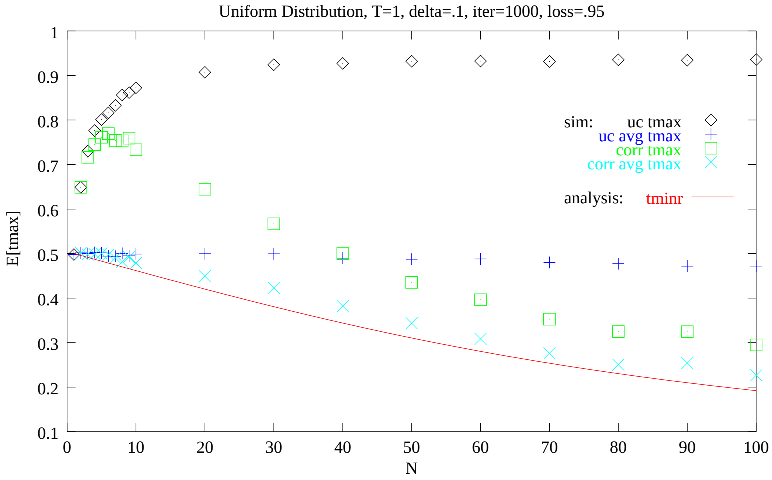

Figure 3.9: $E[avg\ t_{max}]$ **and** $E[tminr]$**: Large Loss** ($l = .95$)**.**

is a straggler:

$$
\begin{aligned}
P_s &= Pr[a\ process\ is\ a\ straggler] \\
&= l^{N-1}
\end{aligned}
$$

The probability there exists a straggler in a group of N processes is the same as 1 minus the probability that no stragglers exist in the group.

$$
\begin{aligned}
P_{s_N} &= Pr[there\ exists\ at\ least\ one\ straggler\ in\ a\ group\ of\ N\ processes] \\
&= 1-(1-P_s)^N \\
&= 1-(1-l^{N-1})^N
\end{aligned}
$$

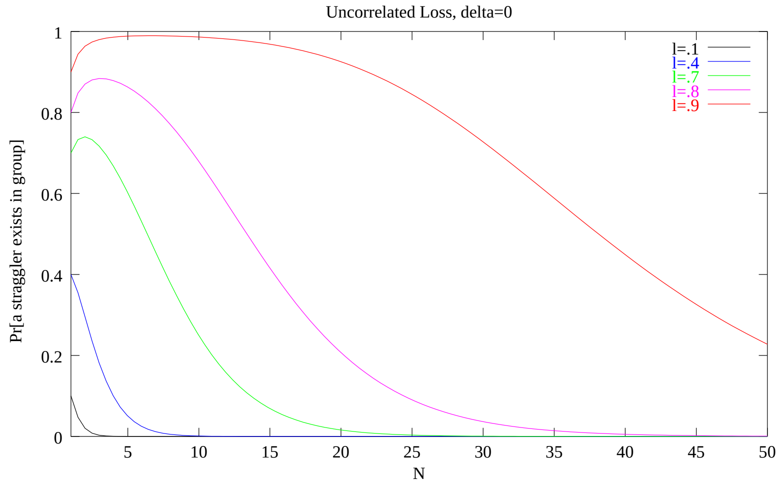

Figure 3.10: **Probability of a Straggler in a Group of Size N.**

These calculations show that small group sizes are prone to stragglers, as are groups with high loss rates (Figure 3.10). In the Suppression algorithm even smaller numbers of processes actually generate messages, so this phenomenon is even more pronounced.

Note also that, as N grows, the value of the largest t_{min_k} increases. For example, when time selection is from a uniform distribution, the largest t_{min_k} approaches the bound on the timer interval T. The impact of a single straggler on t_{max} is potentially greater with larger N, but has a lower probability.

For uncorrelated loss, as $l \to 1$, $avg\ t_{max}$ approaches the mean of the distribution (Figure 3.9). This occurs because, when all N messages are dropped in the network, each node does not receive any of the $N-1$ announcements of its peers. In that case, a node's locally selected time becomes the suppression time, as well as the minimum time received. Basically, each node reverts to using its own value for suppression. Each process i awakening at t_i sends its message. Although none of the messages are received, each individual node believes it has suppressed the others. In fact, there is no way for a process to distinguish between when it has suppressed all its peers and when all the peers' messages have been lost.

For correlated loss, as $l \to 1$, $avg\ t_{max}$ approaches the analytic results for t_{min_e} (Figure 3.9).

Uncorrelated loss consistently produces a larger than or equal t_{max} than the value produced under correlated loss. With small l the uncorrelated and correlated loss graphs converge as N increases (Figure 3.8). With increases in l, convergence no longer occurs not even at larger N (Figure 3.9). However, the individual plots are still asymptotic.

Finally, we note that the simulations match the analysis presented earlier (Figure 3.11).

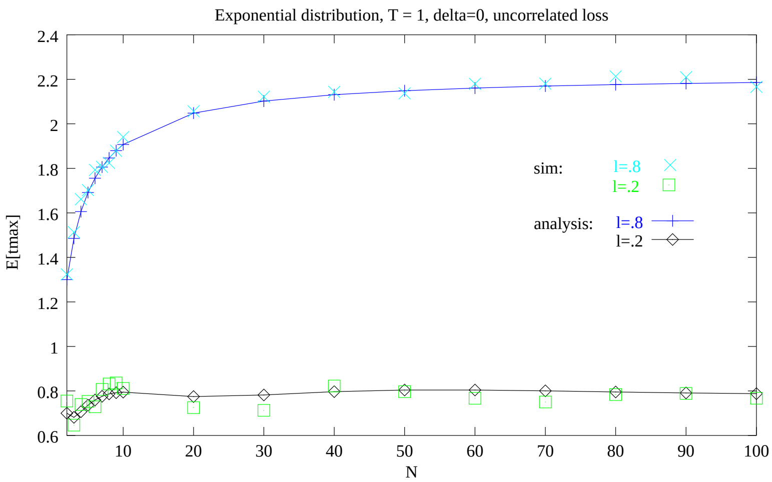

Figure 3.11: $E[t_{max}]$ **vs.** N**: Simulation vs. Analysis**

Averages. We observe from the graphs that, regardless of loss model, $avg\ t_{min} > t_{min}$ and $avg\ t_{max} < t_{max}$. When packet loss occurs, some processes will lose the message containing the definitive minimum time sent, thus the average becomes higher than t_{min}, because additional suppression messages will be generated which have later times. Likewise, when some processes lose the message containing the definitive maximum time sent, the average becomes lower than t_{max}.

3.8.2 Messaging Overhead Metrics

In Table 3.3, we summarize the metrics tracked in our simulations for the Suppression with loss algorithm. The parameter num represents the number of messages sent, $extra$ indicates the number of extra messages generated, and $required$, the number of messages necessary to completely suppress all participating processes. The $avg\ num$ parameter tracks the average number of messages received by the processes, as each process may receive different numbers of messages in the case of uncorrelated loss. The results reported below apply equally well to both uniform and exponential random timers.

Metric	*Description*
num	number messages sent
$extra$	number extra messages sent
$required$	number required messages sent
$avg\ num$	average number messages received

Table 3.3: **Messaging Overhead Metrics Simulated.**

Messages Generated. Regardless of message loss level, $E[num]_{corr} < E[num]_{uc}$, and becomes more apparent with larger N and with greater loss (Figures 3.12 and 3.13). In addition, the average number of messages received is approximately the same for both the uncorrelated and correlated cases. $E[num] > E[avg\ num] > E[required]$ holds across all l. As Δ increases in size, $E[extra]$ becomes a larger component of $E[num]$.

In Figure 3.14, we show that simulations validate the analysis of $E[\#\ messages]_{\Delta=0}$.

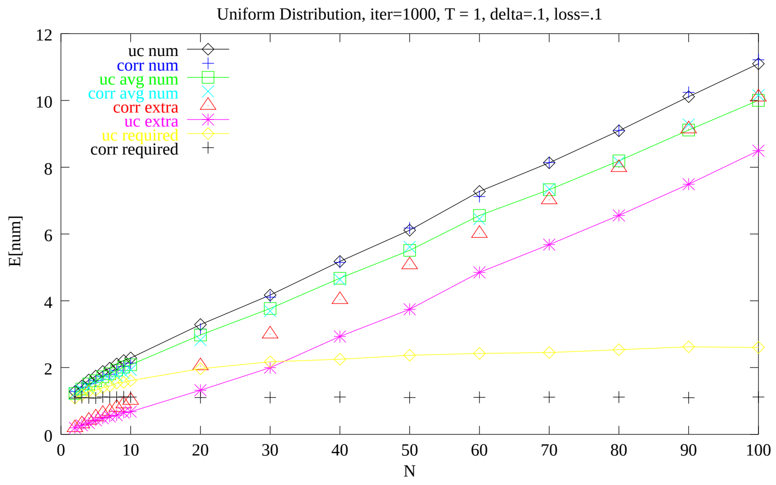

Figure 3.12: $E[num]$ **vs.** N **(**$l = .1$**): Correlated vs. Uncorrelated.**

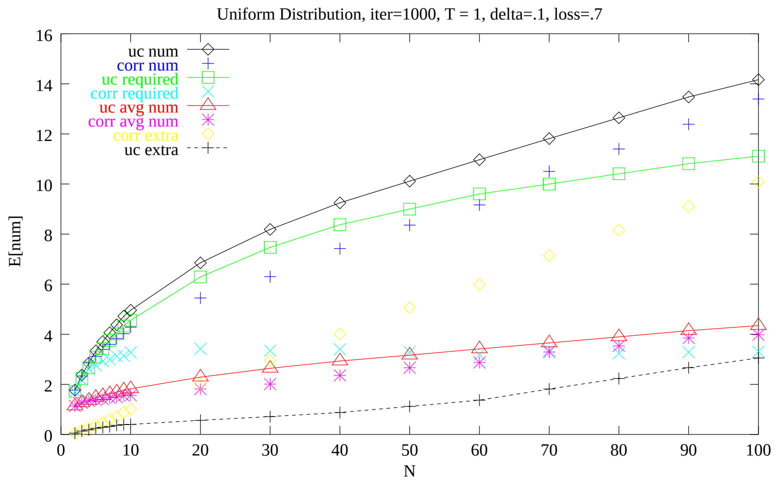

Figure 3.13: $E[num]$ **vs.** N **(**$l = .7$**): Correlated vs. Uncorrelated.**

Extra and Required Messages. In Figures 3.12 and 3.13, we also observe that $E[required]_{uc} > E[required]_{corr}$, and because of the inverse relationship between required messages and extra messages (due to the definition of required messages), $E[extra]_{corr} > E[extra]_{uc}$. As $l \to 1$, the ratio of required messages to messages sent rapidly approaches 1, meaning many more messages are needed for Suppression to work properly. More importantly, $E[required]$ becomes asymptotic as N increases, for all l. The implication is that beyond some N, the same number of messages are

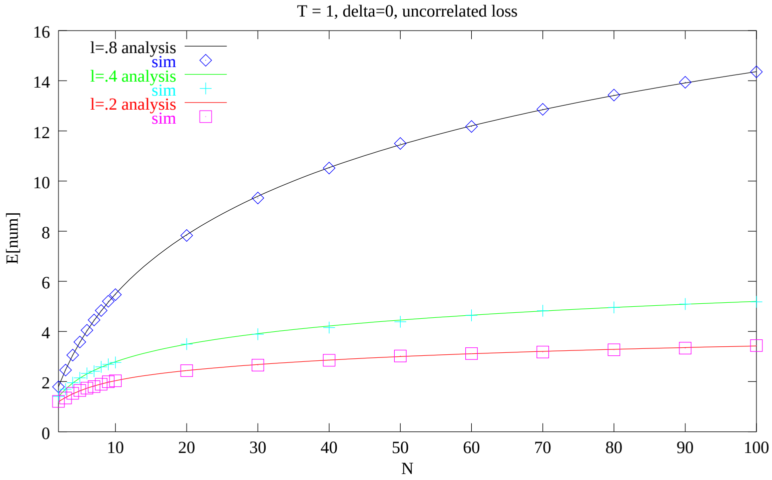

Figure 3.14: $E[\#\ messages]_{\Delta=0}$ **vs.** N**: Simulation vs. Analysis.**

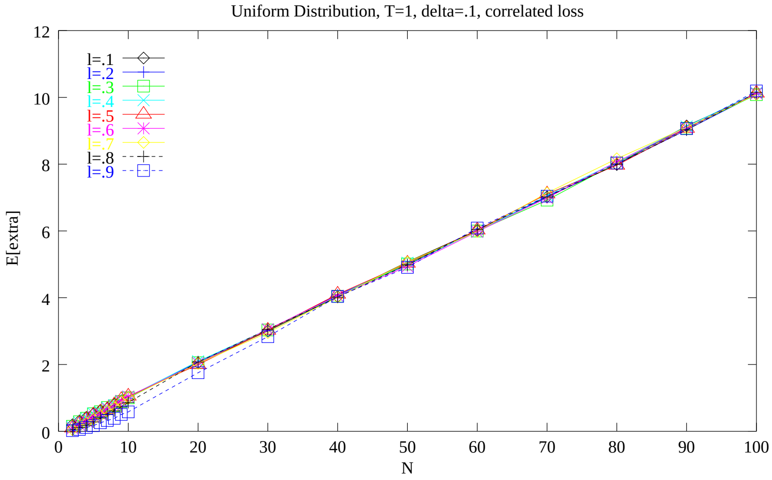

Figure 3.15: $E[extra]$ **vs.** N**: Correlated Loss (Varying** l**).**

necessary for Suppression to operate correctly.

In addition, $E[extra]_{corr}$ is nearly constant across loss levels (Figure 3.15). This phenomenon is due to the fact that, in the correlated case, there is only one message that suppresses all the processes; if it reaches one remote process, it reaches all of them. Therefore, the number of extra messages will be the number of messages arriving within Δ of the message that suppresses the sender. This number is a function of the density of the t_{min_k} values and the ratio of Δ to T; for example, with a uniform distribution where the expected values of the t_i are equally spaced, one would expect that a Δ/T ratio of .1 would lead to $N/10$ extra message arrivals, which corresponds with simulation results.

3.8.3 Loss Models

To summarize our observations relating to the loss model, we note that uncorrelated loss produces lower $E[t_{min_e}]$, as well as higher $E[t_{max}]$. This is due to the fact that probabilistically it is likely that at least one process other than the sender receives the earliest message generated, and probabilistically when N is large enough there will exist an outlier process that does not receive multiple messages sent. Thus, probabilistically uncorrelated loss will produce longer delays before the last message is sent.

Moreover, correlated loss leads to an optimal message ordering. Because all messages are either lost or received, each message sent will be in T_{min} order. The first message will be sent at t_{min_0}. If that is unsuccessful, it will be followed by one sent at t_{min_1}, followed by at t_{min_2} and so forth. Uncorrelated loss doesn't have that property. The earliest message, while lost by some processes, may be received by others. Therefore, t_{min_0} would be followed by t_{min_i}, where i is at best equal to 1, but will be the minimum of the remaining unsuppressed processes.

As a result, uncorrelated loss leads to higher $E[messages]$ and $E[required]$, whereas correlated loss leads to higher $E[extra]$.

3.8.4 Distributions

As in the lossless case, the uniform distribution performs better than an exponential distribution with regard to timing metrics, but performs worse with regard to messaging overhead metrics. These findings typically are more pronounced the larger the loss.

This makes more sense in light of what we know about the distribution of $E[t_{min_k}]$. As k increases, the expected values for t_{min_k} remain equally spaced from each other in the uniform distribution, while they are increasingly further apart for the exponential distribution (Figures 3.16 and 3.17).

For a metric such as $E[t_{max}]$, the expected time of the last Suppression message generated, this can result in substantial performance differences between the uniform and exponential distribution,

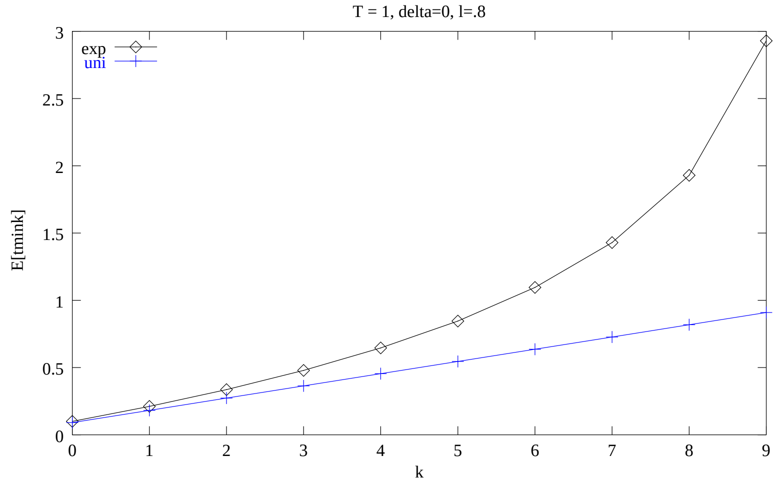

Figure 3.16: $E[t_{mink_k}]$ **vs.** k**: Uniform vs. Exponential (Small** N**).**

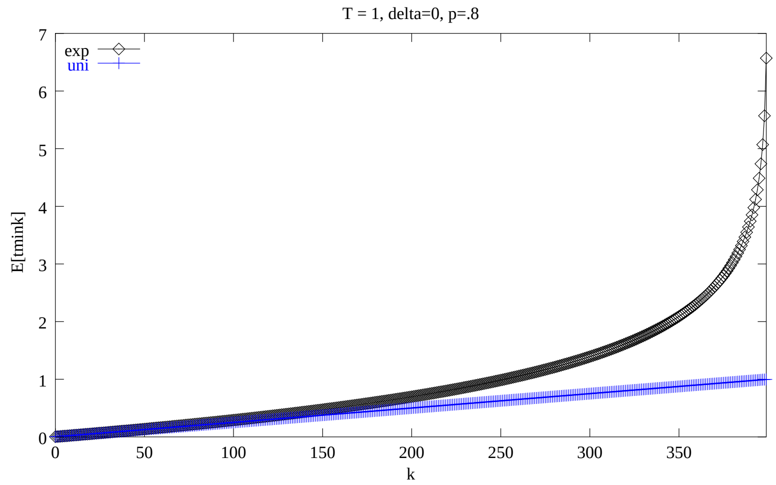

Figure 3.17: $E[t_{mink_k}]$ **vs.** k**: Uniform vs. Exponential (Large** N**).**

depending on the loss model. Thus, if t_{max} is an important delimiter for an algorithm, then the choice of an exponential distribution should be reconsidered. Although the differences are most notable with $N < 40$ in the correlated case (Figure 3.18), in the uncorrelated case the differences are noticeable up to group sizes $N = 100$ for moderate loss levels ($l = .4$), and even higher as p increases (Figure 3.19).

An interesting phenomenon is that the average number of messages received decreases as loss rate increases. This holds for both distributions. Although we only display a graph showing correlated loss, this also holds for uncorrelated loss (Figure 3.20). In the uncorrelated case, each process receives one message that constitutes the earliest Suppression message plus it receives any other messages generated as a function of the local process having a t_i within Δ of t_{min_e}. As p increases, fewer

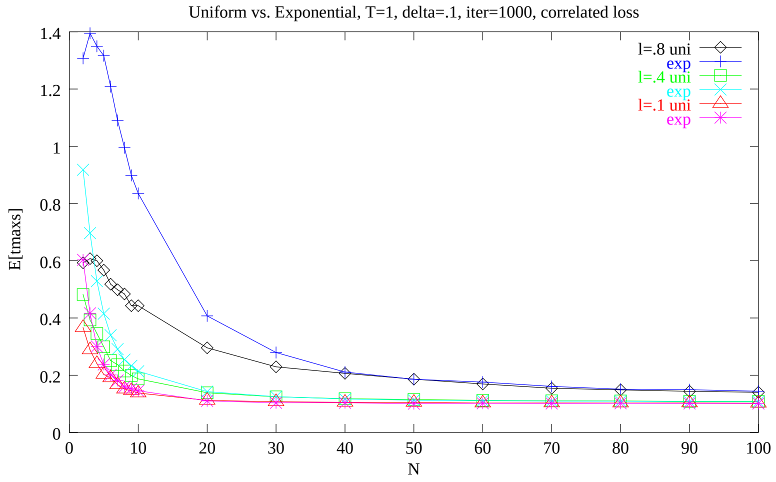

Figure 3.18: $E[t_{max}]$ **vs.** N**: Correlated Loss.**

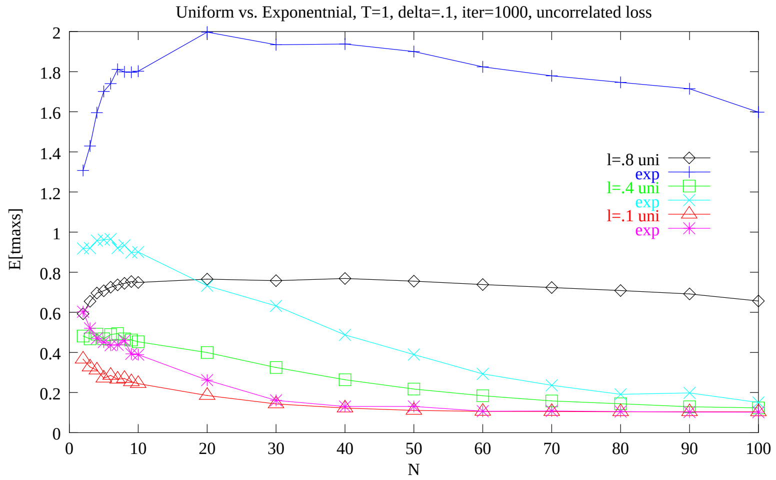

Figure 3.19: $E[t_{max}]$ **vs.** N**: Uncorrelated Loss.**

of the extra messages successfully get through and the overall average number of messages received drops. As $p \to 1$, the average number of messages received should approach 1, as fewer and fewer extra messages are delivered successfully. In the correlated case, fewer extra messages are delivered, but proportionally fewer of the Suppression messages are delivered as well, since with uncorrelated loss each process may receive multiple Suppression messages before the algorithm completes.

Similarly, as $p \to 1$, $E[\#\ extra]$ decreases, regardless of distribution. As stated previously, uniform distributions, which exhibit better response times, produce more extra messages than the exponential distribution, particularly with uncorrelated loss.

For a particular level of message loss, the number of required messages is constant across distributions. In other words, the uniform and exponential distributions lead to identical $E[required]$ (Figure 3.21). This is due to the fact that required messages are defined as the number of messages

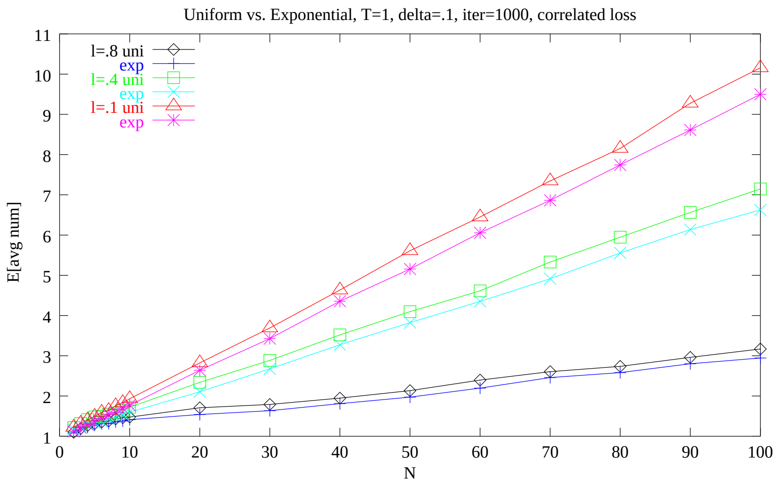

Figure 3.20: $E[avg\ num]$ **vs.** N**: Correlated Loss.**

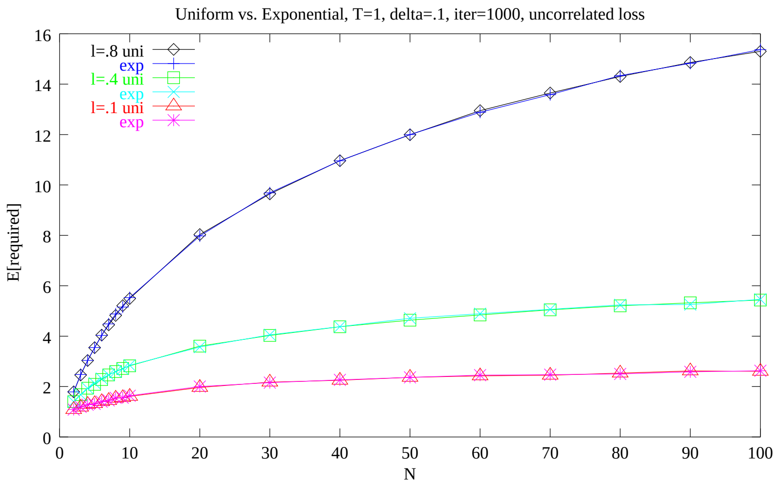

Figure 3.21: $E[required]$ **vs.** N**: Uncorrelated Loss.**

generated by the algorithm when there is zero delay present; in Section 3.4 we show that this metric is purely based on $P(i, N)$, a function only impacted by message loss. Therefore, the distribution for the Suppression interval is rendered immaterial.

3.9 Related Work

Observation of the MBone revealed that there were three general classes of loss that occur [37]; (1) 25% of receivers suffered no loss throughout the day, (2) of those experiencing loss, the median loss rate was between 5-10% for the majority of the day, and (3) a full 25% of the receivers experienced loss rates greater than 15% all afternoon. Essentially, the loss distribution was very long tailed, with large amounts of relatively low loss rates. A full 80% of all nodes reported loss rates less than 20%. Yet, these results suggest that simulations with only single packet losses easily underestimate realistic loss conditions [27] [50] [47].

Handley's results [37] corroborate findings in an earlier study conducted by Kurose et al. [59], which found that loss on backbone links was small, as compared to the average loss observed by a receiver. That is, much of the loss occurred at the edges of the network close to senders or receivers. The result is that the pattern of loss is closer to the extremes: fully correlated or fully uncorrelated.

Both studies confirm the existence of loss correlation. However, Handley [37] also found that there always exists a small number of receivers suffering very high uncorrelated loss.

Although there have been several studies that analyze the performance of Suppression with loss (in the context of reliable multicast protocols, in the form of two back-to-back Suppression phases), loss is modeled in fairly simplified terms [27] [50] [45].

While Floyd et al. offer a detailed examination of the effects of topology on Suppression, they only study the impact of a single loss of original data, not the loss of repairs and requests [27]. Even though this is a good first step toward understanding the impact of loss on Suppression, it falls short of realistic conditions. Raman et al. [50] extend the work in [27], focusing on understanding the impact of large group size on the number of duplicate NACK messages. Again, the loss model only studies the effects of single packet losses on the algorithm.

Nonnenmacher and Biersack [45] simulate slightly more complicated loss scenarios, in terms of where the losses occurs (i.e., between the sender and receivers or between the receivers themselves during a repair), and the full range of loss levels; however, they do not provide accompanying analysis.

Birman et al. analyze a bimodal reliable multicast protocol under probabilistic failures [10] [39], but only examine independent identically distributed losses.

Whereas several previous studies concentrate on the impact of single losses and/or losses close to the source [27] [50] [45], this thesis studies multiple simultaneous losses (that occur anywhere in the system), it looks at the full spectrum of loss rates, and it considers uncorrelated loss and correlated loss.

Although there are other studies, such as the ECSRM [33] and SHARQFEC [41] protocols, that study the scalability of Suppression and simulate the effects of multiple losses on it, the research does not focus on the analytic derivation of performance metrics. Nonnenmacher's work derives metrics analytically for Suppression in the lossless case [46], however no analysis of metrics is presented for either the lossy case or the combined loss and delay case.

3.10 Summary of Results

In this chapter we analytically derived Suppression metrics under the combined conditions of loss and delay, or simply in terms of loss with zero-delay when the analysis proved intractable. We re-examined the metrics introduced in Chapter 2 and introduced several new metrics to characterize the performance of Suppression.

We revisited the meaning of the metric $E[t_{min}]$, offering a more realistic definition of the minimum response time of the algorithm, $E[t_{min_e}]$, the expected time of the earliest message sent but not completely dropped in the network. In other words, the earliest hope of Suppression by another process. To complement this, we presented a bound for $E[t_{max}]$, the expected time after which all processes are considered suppressed, i.e., the completion time of the algorithm. We defined the vector $\vec{T}_{min}$, an ordering of the Suppression wake-up times selected, and calculated $E[t_{min_k}]$, for $0 \leq k < N$, for each probability density function. The comparison explained why the uniform distribution outperforms the exponential distribution for time-related metrics, and vice versa for overhead-related metrics.

On closer examination of $E[\#\ extra]$ we found that it is just one component of $E[\#\ messages]$, the total number of messages generated by the algorithm. Although we were unable to derive a closed form expression to describe $E[\#\ messages]$ analytically, we were able to observe its behavior in simulation and offered a recurrence relation to explain $E[\#\ messages]_{\Delta=0}$, the number of messages generated in the zero-delay case. We postulated that $E[\#\ messages]$ is at least as great $E[\#\ messages]_{\Delta=0}$.

$E[\#\ extra]$ was redefined more broadly as the overhead of the algorithm, whereas $E[\#\ required]$ was established as the expected number of messages required to fully suppress a group of size N. Another way to think about $E[\#\ required]$ is as the number of messages generated when there exists no transmission delay. The implication is that the number of required messages remains constant across all Δ. More importantly, the number of required messages is the same across distributions, as it only has a loss component and no delay component. Finally, $E[\#\ required]$ is asymptotic. That is, beyond a certain N, the same number of messages are necessary for Suppression to work properly. In the next section, we discuss the important implication of this result; perhaps a redesign of the Suppression algorithm would optimize performance further.

Our analysis of the metrics was validated through extensive simulations, which employed a more sophisticated and more realistic loss model, compared to all known studies of Suppression analytically and all of which limit the number of dropped messages, the placement of the loss in the topology, or the level of lossage. For each metric, we presented bounds for worst-case performance, after a thorough examination of both the correlated and uncorrelated loss scenarios. Finally, our simulation and analytic results matched.

3.11 Future Work

As stated in Chapter 2, there are refinements that would be beneficial to make to the model. In particular, we would like to re-evaluate these metrics under conditions of variable delay.

Presently, we have expressions for $E[\#\ messages]_{l=0}$ and $E[\#\ messages]_{\Delta=0}$, which are complementary. However in the future, we would like to derive an expression for $E[\#\ messages]$ with both delay and loss present in the system. This in turn would allow an analytic solution for $E[\#\ extra]$. In addition, it would be helpful to have a closed form solution for $E[t_{min_k}]$ in the exponential case when making the assumption that N is an integer. Similarly, a general expression is sought for $E[t_{max}]$ in the presence of both delay and loss.

An interesting byproduct of knowing $E[\#\ required]$, given the other system parameters p, T, N, and Δ, is that we can calibrate a *believability* factor. In other words, a node must decide whether or not to believe that the correct course of action is to suppress itself when it receives a suppression message from another node. Or should it send its message anyway? For instance, if $E[\#\ required]$, the number of messages required to suppress all N nodes is three, and a receiver has only received two messages before it awakens and must choose whether to suppress or to send its own message, then perhaps it should send its message anyway.

To examine this thread of reasoning, we could create a new parameterized Suppression algorithm that sends a certain number of messages at wake-up, as shown in Program 3.1. The number of messages to send at wake-up would be based on a function $g(m)$ where m is the number of messages already received. While this approach may be quite effective when Δ/T is small, it may be much less so as Δ becomes a substantial part of the timer interval, T.

Alternatively, to minimize $E[t_{max}]$, the node that awakens earliest could issue $E[\#\ required]$ messages immediately. Another scheme could send the $E[\#\ required]$ messages from different senders by basing the to-send-or-not-to-send decision on a probabilistic estimate. For instance, if the time t_i selected by node i is smaller than $E[t_{min_r}]$, where $r = \#\ required$, then issue a message immedi-

```
SUPPRESSION-PARAMETERIZED (d, T, N, Δ, p))
    t = random(d, T)
    sleep(t)
    m = num_messages_received()
    r = required_messages (d, T, N, Δ, p))
    if (m < r) then
        send_message(g(m))
```

Program 3.1: **Parameterized Suppression Algorithm.**

ately, thus sending the messages from different topological regions. Because $E[\#\ required]$ becomes constant beyond a particular group size, these algorithms would be robust to large fluctuations in group membership. These approaches of course may lead to additional extra messages (or other drawbacks). Future investigation should explore if there are benefits across the different loss models and random timer distributions.

Chapter 4

Announce-Listen

In this chapter, we explore Announce-Listen as a scalable form of group communication. We present the core Announce-Listen algorithm, as well as an enhanced version with caching. We describe the algorithm's salient parameters and several important metrics for gauging its performance: consistency, convergence time, messaging overhead and memory usage. We focus on the derivation of a model for the consistency metric, providing analysis and simulation, then discuss the other metrics in terms of how they meet consistency requirements. This chapter concludes with an assessment of previous work on this topic, a summary of our findings, and a discussion of future directions for our research.

4.1 Core Algorithm

Each process participating in the Announce-Listen algorithm makes announcements at a fixed periodicity, T (Figure 4.1). An announcement from process p to process q experiences a transmission delay of Δ_{pq} and contains one piece of essential information, a key-value pair that is stored in a table at the recipient. The key is an identifier for the announcing process and the value is the state that the process disseminates. For example, a process might announce its location, its load, or even the time it expects to send the next announcement. Each process also listens for announcements from other processes. We call this table a registry and the collection of registries at each listener forms a distributed directory.

Announcements are sent to a multicast group address where they are disseminated simultaneously to all participating processes (Figure 4.2). Due to a combination of transmission delay and loss in the network, it may take multiple announcement periods for a particular piece of data originating at the sender to reach all of the receivers.

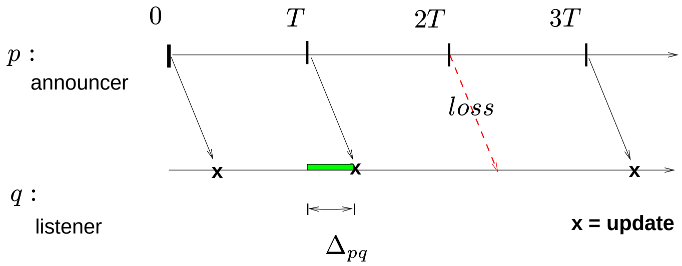

Figure 4.1: **Periodic Announcements and Updates.**

The basic algorithm is shown in Program 4.1. Each process acts as an announcer and sends its state information to the multicast address every T units of time. It accomplishes this by sending an announcement, then setting an announce timer to expire T units of time in the future. When the timer expires, the process sends the next announcement. Concurrently, each process acts as a listener on the same address, listening for announcements from other processes. On the receipt of an announcement, process state information is stored in the listener's directory, where it is stored indefinitely. Specifically, the entry in the registry indexed by *msg.key* is updated to store the value *msg.value*. In our studies we consider every process to be simultaneously both an announcer and a listener.[1]

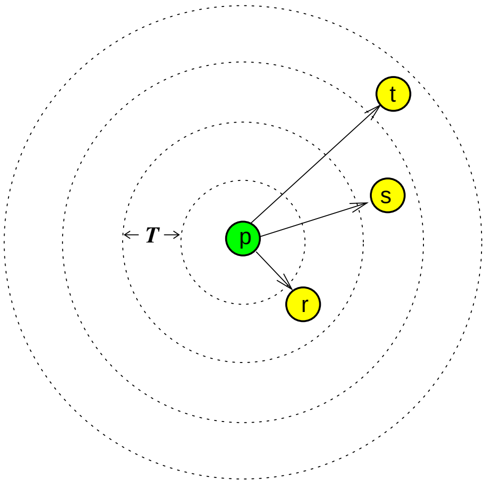

Figure 4.2: **Announcing to Multiple Processes.**

A more realistic representation of the AL algorithm is that a process caches entries for a limited time. Thus, we introduce a modified version of the algorithm in Program 4.2 that ages state information and removes it from the cache upon final expiration. We use this algorithm as the basis for the analysis in this chapter.

[1] Although we use the same address for announcing and listening, an implementation might use separate group addresses for each of these tasks. The effect of this would be that the process could be an "announce-only" process for one address, and a "listen-only" process for the other address.

```
ANNOUNCE-LISTEN (T)
    send_announcement ()
    set_announce_timer (T)
    do
        if receive_announcement (msg) then
            update_registry_entry (msg.key, msg.value)
        if announce_timer_expired () then
            send_announcement ()
            set_announce_timer (T)
```

Program 4.1: **Announce-Listen Algorithm.**

```
ANNOUNCE-LISTEN-WITH-CACHING (T_A, T_L, max_age)
    send_announcement ()
    set_announce_timer (T_A)
    do
        if receive_announcement (msg) then
            update_registry_entry (msg.key, msg.value)
            set_listen_timer (msg.key, T_L)
        if i = listen_timer_expired () then
            age = age_entry (i)
            if age > max_age then
                remove_registry_entry (i)
            else set_listen_timer (i, T_L)
        if announce_timer_expired () then
            send_announcement ()
            set_announce_timer (T_A)
```

Program 4.2: **Announce-Listen Algorithm with Caching.**

The new algorithm relies on multiple timer values: T_A, the announcement periodicity, and T_L, a cache entry renewal periodicity. T_A behaves identically to T in the original program, and is used as the bound on the announce timer that reminds an announcer to send the next announcement. T_L is the bound on the wake-up timer that reminds a listener either to expire or to renew an entry in the registry.

A process begins the algorithm by sending an announcement message, then setting the announce timer to expire in T_A units of time. The process waits for one of three events to occur:

1. If the process receives an announcement, it caches a key-value pair in its registry, as before; the entry *msg.key* is updated to *msg.value*. However, now *each* registry entry is associated with an expiration timer. Thus, the listener resets the listen timer for entry *msg.key* to expire T_L units of time into the future.

2. If the listen timer expires, the process did not hear an update from the announcer associated with the i^{th} entry within the last T_L units of time. The process ages the registry entry, incrementing its age by one. The variable `age` depicts how many consecutive expirations have occurred for an entry at a given point in time. The expiration process allows *max_age* such expirations, before removing the data permanently. Whenever an announcement arrives for a given entry, the `age` variable is reset to 0 inside `update_registry_entry`().

3. If the announce timer expires, the process issues its next announcement and resets the announce timer.

Although there exist variations of AL that send announcements using adaptive announce timers [55], for simplicity we examine the behavior of the algorithm when all entries use the same fixed announcement periodicity. Adaptive timers have been used to handle an influx of group members while keeping the overhead of communication fixed. If the membership grows large, the idea is to reduce the frequency of messages by increasing the periodicity, T_A, resulting in fewer messages. Likewise when the group size shrinks, the frequency of messaging increases.

There are also implementations that use progressively larger listen timer expiration values between each round of expiration, but we assume processes use the same fixed listen timer value, not only between each other, but also between rounds of aging cache data (e.g., step 13 in Program 4.2).

Here, we make the simplifying assumption that $T_A = T_L = T$. However, we assume `max_age` is set high enough to allow data to transit the network (e.g., $max\_age \geq 2$) and to accommodate the level of message loss in the network. In effect, announcements are issued at regular intervals of T_A and their state is cached for a duration of up to $T_L = kT_A$, where $k =$ `max_age`, typically a small integer value.

4.1.1 Scalability

Announce-Listen (AL) is an algorithm that uses periodic messaging to propagate local state information to remote processes. The idea of using announcements, with no form of feedback, is in contrast to traditional acknowledgment-based messaging, where the receipt of a message triggers an explicit acknowledgment (ACK) or where the detection of a missed message triggers a negative acknowledgment (NACK). One motivation behind AL is that an acknowledgment-style handshake between processes works well between pairs of processes, but can be problematic among groups of processes. As stated earlier, when a message is multicast to a large group of processes, there is the potential that implosion may result from simultaneous ACKs. Even when only NACKs are used to indicate negative vs. positive receipt of a message, implosion may still result when losses are correlated and simultaneous NACKs might be generated within a whole branch of the multicast routing tree. Therefore, AL offers groups of processes the potential to avoid implosion and to scale more gracefully than traditional ACK/NACK messaging.

Another motivation for AL is to allow large groups of processes to be unencumbered by a feedback mechanism. Whereas Suppression attempts to reduce the number of responses and to spread them out over the Suppression interval, Announce-Listen attempts to eliminate explicit process interaction altogether. This is important in scenarios where time is of the essence. The ability to "not wait" for an acknowledgment removes not only a round-trip time (RTT) of delay, but also eliminates the considerable extra delay incurred when a message goes unacknowledged and must be retransmitted several times. AL can be thought of as decoupling the sending process from the receiver process(es), allowing each to behave asynchronously. Thus, AL not only generates fewer messages by removing feedback messages, but it also removes the delay associated them.

However, the removal of an explicit, tightly-coupled feedback mechanism has a direct impact on the way in which reliable data distribution can be implemented in such a system. An implicit form of feedback is possible, for example by piggy-backing feedback information in announcements themselves (see Section 4.9), but the information is now delayed until such time as an announcement is scheduled by the recipient (the reporter of feedback) in the direction back to the original sender.

Nonetheless, a compelling argument for AL is that there are applications in which reliable data delivery is not of critical importance. There are also applications where eventual data delivery is an acceptable level of service, and still others where we can forego reliability entirely. For instance, systems with a real-time element, or ones where data changes rapidly, may not require reliable delivery because by the time data is retransmitted it has changed anyway. An example that falls in this class of applications is a mobile device that announces location coordinates on a regular basis.

The reasoning behind why AL is a suffiicient alternative is that, through repeated announcements, a message eventually will reach its intended destination(s), and with repeated announcements the state at receivers will track the state of the announcer. Thus, for applications that can tolerate a more loose form of communication, guaranteed delivery of every message (in the form of acknowledgments or otherwise) becomes less necessary.

In short, Announce-Listen offers scalability to groups of communicating processes because it removes feedback messages thereby avoiding message implosion, and eliminates waiting for feedback messages. It is beneficial to use AL in lieu of acknowledged messaging in situations where there is little to no back-channel bandwidth or, even if there is a modicum of bandwidth, where feedback messages would overrun the sender. AL messaging is resilient to faults in the network and changes in group membership because state is continually replenished so processes can quickly learn the state of the system. However, scalability hinges on the fact that data changes happen at a high enough rate to warrant continual updates. Furthermore, AL is most appropriate for those applications where the eventual consistency of this loosely-coupled messaging model can be tolerated.

4.1.2 Model Parameters

In addition to T_A, the announcement periodicity, and T_L, the cache entry expiration timer, there are several other parameters used in our model: N, the number of processes participating in the algorithm; Δ, initially a fixed message transmission delay between processes; p, the probability of message loss; k, the number of times a cache will age an entry before expiring it entirely.

The update periodicity, the frequency with which the content of announcement messages changes (separate from process arrivals or departures) is given by the parameter r, where $r \geq T$. When $r = \infty$, the data value never changes, so each announcement contains the same data as the first announcement. When $r = T$, the data value changes with each announcement. Although the data may change more frequently than T, the announcement periodicity, there is no way to distribute it more quickly. Therefore, initially we assume $r \geq T$.

A process is considered to have *departed* from the system when it stops making announcements. Otherwise, a process is considered *alive*. $D(t)$ is the cumulative distribution function of the probability that a process departs t units of time after it arrives. $A(t)$ is the cumulative distribution function of the probability that a process arrives into the system at time t. The relationship between $D(t)$ and $A(t)$ is highly group specific. For instance, in a collaborative session that is scheduled to begin at a specific time, arrivals are likely to cluster around the beginning of the session, whereas departures are likely to cluster at the ending time. Membership is very stable during the middle of the session. In an application with mobile group members, arrivals and departures may be extremely dynamic.

These parameters are summarized in Table 4.1.

Parameter	*Description*
T_A	announcement periodicity
T_L	cache entry renewal interval
N	number of processes participating
Δ	transmission delay
p	message loss probability
k	maximum entry age
r	data change periodicity
$D(t)$	cumulative probability of process departure
$A(t)$	cumulative probability of process arrival

Table 4.1: **Announce-Listen Parameters.**

4.1.3 Metrics

There are several metrics by which we can evaluate Announce-Listen. A key metric is the level of consistency achieved among the N communicating processes as a function of time. Consistency is a measure of how closely information stored in caches at the listeners compares with the actual state of the announcers. Consistency is challenged whenever new information arrives into the system (through an announcement either updating old information or reporting the arrival of a new process) or old information departs (as might happen when data expires). A common goal for the Announce-Listen technique is to maximize consistency while minimizing the other metrics: convergence time, messaging overhead and memory usage.

Given consistency as a function of time, $C(t)$, we can determine the convergence time of the listeners, $t(c_o)$, i.e., the minimum time it takes to reach a particular level of consistency, c_o.

We derive a metric for messaging overhead, in terms of the amount of bandwidth consumed by the algorithm; we predict the number of messaging attempts that are necessary to attain a given level of consistency and discuss the fraction of those that are redundant due to the update periodicity (of information inside of announcements) or due to the nature of the multicast distribution.

Additionally, we present a metric for memory usage, the amount of memory consumed to achieve a certain level of consistency. This metric is particularly useful for properly tuning the cache expiration strategy for listeners, which must weigh the cost of storing information for too long against storing it for too short a time. While the optimal cache expiration strategy may be unimportant for memory-rich systems or ones with secondary storage, it may be critical for devices that are memory poor, such as hand-held devices or embedded sensors.

Whereas we focused on time and overhead metrics for the Suppression technique, for Announce-Listen we concentrate on consistency.

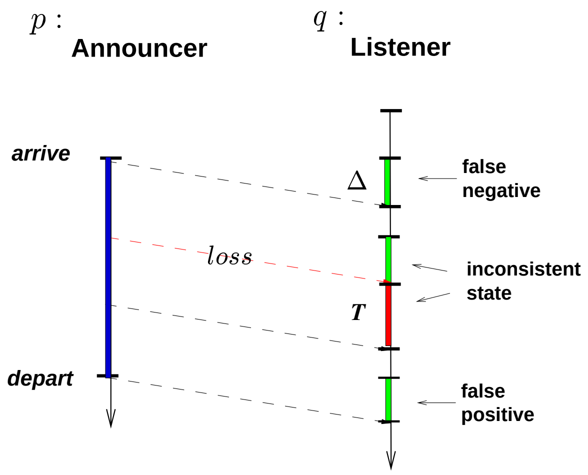

Figure 4.3: **Listener Errors.**

4.2 Consistency: Registry Cost

We can reformulate the issue of consistency by asking how good is the distributed registry built from processes using Announce-Listen? By *good* we mean how accurate is the registry. To answer this question, we define an objective function that attaches a cost to several types of errors: false negatives, false positives, and inconsistent state. We illustrate these errors using two communicating processes, an announcer and a listener process as shown in Figure 4.3.

- **false negative**: $p.announcing$ is true (process p is issuing announcement messages), but state information (a key-value pair) for p does not appear in the registry associated with the listener process q, $q.registry$. The cost of omitting information from the registry is $cost_n$.

- **false positive**: process p is a member of $q.registry$, but it is no longer issuing announcement messages. The cost of keeping expired information in the registry is $cost_p$.

- **inconsistent state**: process p is a member of $q.registry$ and is issuing announcement messages, but the information pertaining to p in $q.registry$ does not match the information for p in its own registry, $p.registry$. The cost of maintaining incorrect information in the registry is $cost_s$.

The severity of these three errors is not the same. While a false positive is misleading, in that it provides stale information about a process no longer participating in the registry, a false negative provides no information at all. Similarly, a piece of inconsistent information could be better than no information. For example, if the registry stores resource location information, a false positive is potentially useful in that it provides a hint of past history which may be used to track down the process in the future, if the process resumes announcements to the group at a later time. In this context, wrong state information also has potential benefit, even more so than false positives. Because the process is actually still participating, the information in the registry, while stale, may allow the process to track down the correct value through proxying or forwarding entities.

Thus for this example, $cost_n \geq cost_p \geq cost_s$ and the objective is to minimize the overall cost of inconsistencies. Other scenarios may differ as this is application dependent. For instance, if a theatre is announcing its movie schedule, a false positive may mean someone may make an unnecessary trip. On the other hand, a false negative may mean one simply schedules to see the movie later or elsewhere.

The cost of q's registry, $Cost(q)$, is given by:

$$\begin{aligned}
&Cost(q) = \\
&\quad (\#\, p :: p.announcing \wedge (\forall v :: (p, v) \notin q.registry)) * cost_n + \\
&\quad (\#\, p :: \neg p.announcing \wedge (\exists v :: (p, v) \in q.registry)) * cost_p + \\
&\quad (\#\, p :: (\exists v :: (p, v) \in q.registry) \wedge (p, v) \notin p.registry)) * cost_s
\end{aligned}$$

$Cost(q)$ is a random variable that depends on which processes are in the system, and on which messages are lost by the network. Therefore, we evaluate AL by examining the average cost of a registry, $E[Cost(q)]$.

Below we derive the likelihood of false positives, false negatives, and inconsistent state. First we create a model of the listener state transition probabilities. Next we relate it to an estimate of any given registry entry being in error, then revise these estimates to accommodate process departures and discuss the impact of process arrivals.

Once we can approximate the degree of error on a per entry basis, we can deduce the overall consistency in a given registry, and from that we can extrapolate the overall consistency of a collection of distributed registries participating in the Announce-Listen algorithm.

4.2.1 Listener State Transition Probabilities

A model of the listener process is depicted in Figure 4.4, which displays listener state transition probabilities.

When a listener receives an announcement with probability $1 - p$ from a remote process, the process becomes newly `Alive`, i.e., the announcer's state is entered into the listener's registry. Once registered, the remote process refreshes its state periodically by sending renewal messages. These renewal messages are received with probability $1 - p$, and dropped with probability p. If dropped, the listener cycles through k `Not Sure` states before marking the announcing process as `Departed`, i.e., expiring the announcer's state from the registry. As the listener progresses from one `Not Sure`

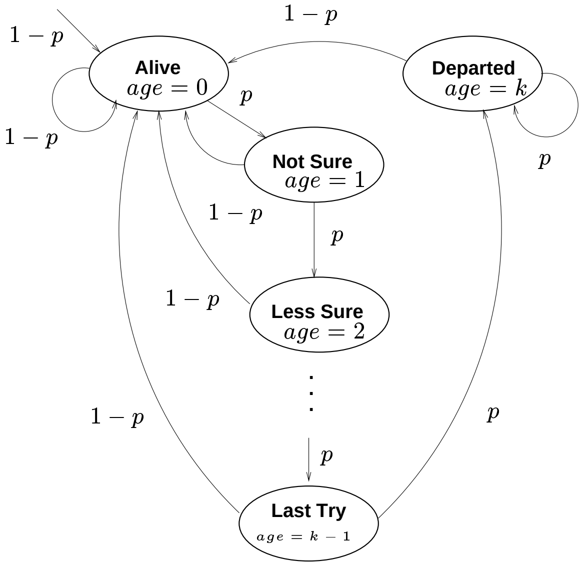

Figure 4.4: **State Transition Probabilities.**

state to the next, the listener becomes progressively less certain as to the state of the announcer. The state `Last Try` is the last state in which the listener maintains a valid registry entry for the announcer. If the listener regains communication with the announcer, the announcer is considered `Alive` again and its state is refreshed.

Each state is also associated with the *age* of the entry, which appears as a state name in Figure 4.4. For an entry to have $age = m$, m listen timers must have expired and the m^{th} timer expired at time mT_L (relative to when the entry was inserted into the registry). The age changes to $(m+1)$ at time $(m+1)T_L$ if no announcements were received in the interval $[mT_L, (m+1)T_L]$. That means all announcements sent in $[mT_L - \Delta, (m+1)T_L - \Delta]$ were lost. The number of announcements is:

$$a \quad = \quad \left\lfloor \frac{(m+1)T_L - \Delta}{T_A} \right\rfloor - \left\lfloor \frac{mT_L - \Delta}{T_A} \right\rfloor$$

Therefore, the state transition probability from $age = m$ to $age = m+1$ is p^a. When $T_L = T_A$, $a = 1$, giving the probabilities shown in Figure 4.4.

4.2.2 Error Model

Figure 4.5 associates listener actions with each state transition and depicts the listener's event-driven caching and aging strategy. We use the convention that state transitions are labelled with $\frac{A}{B}$, where A is an event that occurs, and B is the consequence of the event occurring. The diagram also offers another way to think about the listener model, by comparing it to the model maintained by the announcer.

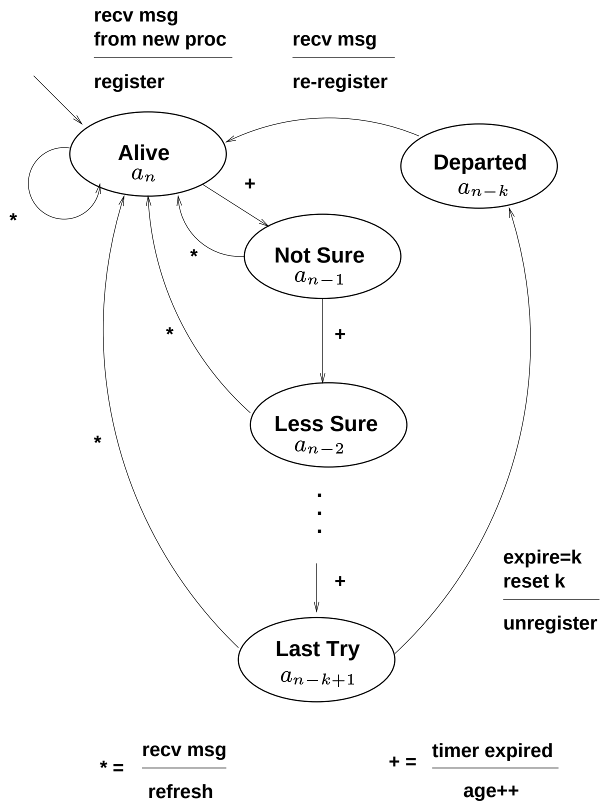

Figure 4.5: **Event-Driven Caching and Aging Strategy.**

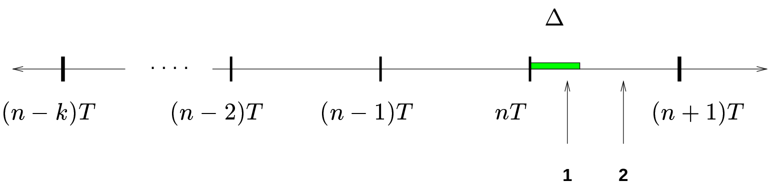

Figure 4.6: **Boundary Condition at Δ.**

When a listener receives an announcement from a process not currently in the registry, it registers the announcer as being `Alive`. The listener refreshes the announcer's registry entry with the receipt of each subsequence announcement message. If the listen timer T_L expires, the listener ages the announcer's state information and becomes `Not Sure` of the announcer's current state. The listener is willing to progress the announcer through $k-1$ stale states. If k expiration periods pass, the announcer is considered `Departed` from the system and its entry is removed from the registry. At such time as a new message is received from the announcer, the listener re-registers the announcer as being `Alive` once again.

If an announcer process begins at time 0, then the number of announcements sent to a listener process by time t is effectively $n = \lfloor t/T \rfloor$. If the listener stores what it thinks is the current state of the announcer, and the announcer has issued n announcements, then ideally the listener state is a_n, the data value sent in the announcer's n^{th} announcement. However if the n^{th} announcement is lost, the listener may contain a previous data value from an earlier announcement. The probability that the listener has cached old state from the m^{th} announcement, a_m, is the probability that the listener actually received the m^{th} announcement and lost all subsequent announcements between m and n, the current announcement:

$$\begin{aligned} Pr[\textit{listener state is } a_m] &= Pr[\textit{received } m^{th} \textit{ announcement}] * \\ &\quad Pr[\textit{missed all states after } a_m] \\ &= (1-p) * p^{n-m} \end{aligned}$$

Now consider the fact that the listener ages the announcer's entry through k old states before removal from the registry: states a_n through a_{n-k+1}, which also appear as state names in diagram 4.5. The *cost* of being in one of these states is captured by its distance from the `Alive` state, a_n. If the listener state is a_s, then the *cost* is given by $n-s$. If the distance $n-s > k$, then the announcer is considered `Departed` and its state expired.

An application may have different requirements regarding the consistency of the registry. We would like to calculate the likelihood that the error in the registry is less than some tolerance e, denoted $Pr[Err \leq e]$, where $0 \leq e \leq k$.

Given the current time t, we know that $a_{\lfloor t/T \rfloor}$ is the last message sent by the announcer. This message takes time Δ before it can be received. We consider two cases: $t < \Delta + \lfloor \frac{t}{T} \rfloor T$ and $t \geq \Delta + \lfloor \frac{t}{T} \rfloor T$. In the former case, we can never be consistent with the announcer because of the network delay. The boundary condition that arises at the listener is displayed in Figure 4.6.

Therefore, $Pr[Err \leq e]$ is evaluated separately in each of these intervals. The probabilities of the listener process being in each of the states are shown in Tables 4.2 and 4.3. The tables are similar, except that when t is within Δ of the Announce message, the listener could not possibly have received the last message, a_n, regardless of whether or not the network loses the message.

State	*Name*	e	$P[Err = e]$
a_n	*Alive*	0	0
a_{n-1}	*Not Sure*	1	$1-p$
a_{n-2}	*Less Sure*	2	$(1-p)p$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_{n-k+1}	*Last Try*	$k-1$	$(1-p)p^{k-2}$
a_{n-k}	*Departed*	k	p^{k-1}

Table 4.2: **Inconsistent State:** $nT \leq t \leq nT + \Delta$.

State	*Name*	e	$P[Err = e]$
a_n	*Alive*	0	$1-p$
a_{n-1}	*Not Sure*	1	$(1-p)p$
a_{n-2}	*Less Sure*	2	$(1-p)p^2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_{n-k+1}	*Last Try*	$k-1$	$(1-p)p^{k-1}$
a_{n-k}	*Departed*	k	p^k

Table 4.3: **Inconsistent State:** $nT + \Delta \leq t \leq (n+1)T$.

The probability within each interval that $Err \leq e$ is a sum of the probabilities that $Pr[Err = i]$ for i from 0 to e. The probability of the current time t falling within (versus after) Δ of the n^{th} announcement is proportional to the ratio of the transmission delay to the announcement periodicity Δ/T (versus $1 - \Delta/T$). By within Δ, we mean $t < \Delta + \lfloor \frac{t}{T} \rfloor T$. By after Δ, we mean $t \geq \Delta + \lfloor \frac{t}{T} \rfloor T$.

$$\begin{aligned} Pr[Err \leq e \mid within\ \Delta] &= \sum_{i=0}^{e} Pr[Err = i]_{within\ \Delta} \\ &= (1-p)(1 + p + p^2 + \cdots + p^e) \\ &= 1 - p^{e+1} \end{aligned}$$

$$Pr[Err \leq e \mid after\ \Delta] = \sum_{i=0}^{e} Pr[Err = i]_{after\ \Delta}$$

$$
\begin{aligned}
&= (1-p)(1+p+p^2+\cdots+p^{e-1}) \\
&= 1-p^e
\end{aligned}
$$

$$
\begin{aligned}
Pr[within\ \Delta] &= \frac{\Delta}{T} \\
Pr[after\ \Delta] &= \frac{T-\Delta}{T}
\end{aligned}
$$

$$
\begin{aligned}
Pr[Err \leq e] &= Pr[Err \leq e \mid within\ \Delta] * Pr[within\ \Delta] + Pr[Err \leq e \mid after\ \Delta] * Pr[after\ \Delta] \\
&= 1 - p^{e+1} - p^e(1-p)\left(\frac{\Delta}{T}\right)
\end{aligned}
$$

4.2.3 Departures

Tables 4.2 and 4.3 are based on the condition that the announcer process is alive and they assume that the announcer only leaves at the state `Departs`. In fact, the announcer may have departed earlier. Therefore, we need to refine the earlier tables and take the departure distribution probability, D(t), into account, focusing in particular on the probability of the announcer being alive at time t, $\overline{D}(t) = 1 - D(t)$.

Below we consider the current time t falling in the interval after Δ, as well as within Δ. In each of these intervals, we present examples of different states and explain that each state represents multiple possibilities, i.e., each leads to different types of errors that arise in the system (false negatives, false positives, and inconsistent state).

Appendix A contains Tables A.1 and A.2 summarizing the observations below. Although not in the tables, we discuss the impact of the rate of data change, r, on the analysis in Section 4.3.

After Δ. Correct states in the system only occur when the current state at the announcer and listener are identical; when the listener considers the announcer `Alive` and the announcer process is not dead, or when the listener thinks the announcer `Departed` and the announcer has departed. According to Figure 4.5, this happens in state a_n when the announcer is alive and the message was received successfully by the listener. This also happens in state a_{n-k} when the announcer has departed and the listener finally expires the announcer's state.

For example, here is how we can refine the `Departed` state by considering departure probabilities, when t occurs after Δ. When a process is not alive at time t, it could have departed at anytime in the interval $[0, t]$. Therefore, the probability of a process not being alive at time t, X_t, is the sum of the probability that the process died at time $(n-k)T$ (the earliest point at which the listener could have detected the departure), plus the probabilities that the process departs within any announcement interval between $(n-k)T$ and the current time t. If the process departs in an intermediate interval, it must also have been preceded by message loss for the listener to be in state a_{n-k}.

$$\begin{aligned}
X_t &= D((n-k)T) + \\
&\quad (D((n-k+1)T) - D((n-k)T)) \times p + \\
&\quad (D((n-k+2)T) - D((n-k+1)T)) \times p^2 + \\
&\quad \vdots \\
&\quad (D(t) - D(nT)) \times p^k
\end{aligned}$$

Another departure from Tables 4.2 and 4.3 arises when the listener process receives an announcement message. In that case, there is no falsely believing the announcer has departed. Thus, a false negative error is not possible. Likewise, a false positive error is not possible when the announcer is actually departed from the system, since we cannot falsely believe the announcer is alive.

Within Δ. When we consider the interval within Δ, there is only one correct state in which the listener and announcer match. This occurs in state a_{n-k} (Figure 4.5) when the announcer has departed and the listener detects it. In that state, it is not possible to have a false positive error where the listener falsely believes that the announcer is alive when it is not. However, it is possible for a false negative to occur if k messages have been lost and the announcer is actually still alive. Except for that case, a false negative error is not possible because a message is always received; therefore, the listener cannot falsely think the process has departed. Finally, it is not possible for the listener to be within Δ of the announcement and for its registry to contain the correct data value for the announcer, since the listener has not received the update message yet.

The calculation for the probability of a process not being alive at time t, Y_t, is similar to that of X_t. However, there is one fewer message to account for because a message could not have possibly reached the listener yet, due to network transmission delay, Δ.

$$\begin{aligned}
Y_t &= D((n-k)T) + \\
&\quad (D((n-k+1)T) - D((n-k)T)) \times p + \\
&\quad (D((n-k+2)T) - D((n-k+1)T)) \times p^2 + \\
&\quad \vdots \\
&\quad (D(t) - D((n-1)T)) \times p^{k-1}
\end{aligned}$$

4.2.4 Overall Registry Consistency

We have seen how to calculate the probability of false positives, false negatives and inconsistent state for one announcer's registry entry. Since each of these corresponds to an error in the registry state, the total probability of error per entry is given by:

$$Pr[Err] = Pr[False\ -] + Pr[False\ +] + Pr[Inconsistent\ State]$$

The expected cost per entry is given by:

$$E[cost] = Pr[False\ -] * cost_n + Pr[False\ +] * cost_p + Pr[Inconsistent\ State] * cost_s$$

At time t, $Pr[Err]$ is the probability of error for one registry entry given one announcer, and $Pr[C] = 1 - Pr[Err]$ is the likelihood of its consistency. We can use $Pr[C]$ to derive $Pr[C_N]$, the probability of consistency for one listener registry with N announcers.

$$Pr[C_N] = (1 - Pr[Err])^{N-1}$$

Note that the state for the local announcer is never inconsistent, thus the usage of $N-1$. We also can differentiate between different types of inconsistency based on the number of announcements that have been missed, as shown in Appendix A.

4.2.5 Arrivals

If we look at each registry entry and start time at $t = 0$, the tables are accurate (Tables 4.2 and 4.3, and Tables A.2 and A.1). However, when we look at global registry consistency, each entry may have a different start time. Thus, let us examine the impact of the arrival distribution on $Pr[Err(t)]$, the probability of error at time t.

Let $A(u)$ be the distribution for process arrivals into the Announce-Listen algorithm. If a process arrives into the system at time u, then for that process the new probability of error at time t is shifted in time becoming $Pr[E(t-u)]$.

Consider an example in the discrete realm. Process i $(0 \leq i \leq N)$ arrives at time i (relative to time 0 when the algorithm begins) and each stays through time A, the last arrival time.

$$\begin{aligned} Average\ Pr[Err(t)] = {} & Pr[Err(t)] \cdot u_0 + Pr[Err(t-1)] \cdot u_1 + \\ & Pr[Err(t-2)] \cdot u_2 + \ldots + Pr[Err(t-N-1)] \cdot u_N \end{aligned}$$

In the continuous domain, the average probability of error at time t becomes the integral

$$Average\ Pr[Err(t)] = \int_0^A A(u) \cdot Pr[Err(t-u)] du$$

4.3 Inconsistent State: Analysis and Simulation

In this section, we focus on the likelihood of inconsistent state. We discuss our intuition about the behavior of the system, then ground it in analysis for $r \geq T$, i.e., the periodicity of data changes is greater than or equal to the announcement periodicity. We derive expressions for the expected probability of single entry consistency, as well as overall registry consistency. We validate these results using simulation.

4.3.1 Analysis

Assuming T and Δ are identical across announcers, $Pr[\textit{one entry is inconsistent}]$ can be determined by summing the entries in the table in Appendix A that correspond to inconsistent state.

First, consider our intuition about the case where $D(t) = 0$, the numbers of late arrivals are 0, and $r = T$, meaning each announcement contains new data. Either an announcement is lost en route to the listener with probability p or it is received with probability $(1 - p)$. In the event that a message is received, the registry is inconsistent for that entry for a period of Δ/T in each announcement interval. Accounting for the fact that the message may not be received, we posit that

$$Pr[\textit{one entry is inconsistent}] \quad = \quad p + (1-p) \times \frac{\Delta}{T}$$

Now let us actually model the impact of $r \geq T$ and compare the results. There are *fewer* inconsistent state errors due to the fact that error only increases when we cross an "r boundary." In other words, a_n could be the first announcement in a cluster of r/T identical announcements. If we assume the cache is infinite, then the number of announcements that have arrived at the listener by time t will be $n = \lfloor t/T \rfloor$. As performed in Section 4.2.2, to arrive at $Pr[Err \leq e \mid within\ \Delta]$ or $Pr[Err \leq e \mid after\ \Delta]$, we just add the columns of the tables. Only now, the columns are infinitely long and lead to the results:

$$\begin{aligned} Pr[Err \leq e \mid within\ \Delta] &= 1 - p^{\lfloor t/T \rfloor - 1} \\ Pr[Err \leq e \mid after\ \Delta] &= p(1 - p^{\lfloor t/T \rfloor - 1}) \end{aligned}$$

If a_n is the second announcement in a series of r/T similar announcements, we can drop the first line in both columns and then sum the remaining columns. If a_n is the k^{th} announcement in a series of r/T similar announcements, $1 \leq k \leq r/T$, we drop the first $(k-1)$ lines. The sums become

$$\begin{aligned} Pr[Err \leq e \mid within\ \Delta] &= p^{k-1} - p^{\lfloor t/T \rfloor - 1} \\ Pr[Err \leq e \mid after\ \Delta] &= p\ (p^{k-1} - p^{\lfloor t/T \rfloor - 1}) \end{aligned}$$

Therefore,

$$\begin{aligned} Pr[\textit{one entry is inconsistent, given } k] &= \left(\frac{\Delta}{T}\right)\left(p^{k-1} - p^{\lfloor t/T \rfloor - 1}\right) + \\ &\quad \left(1 - \frac{\Delta}{T}\right) p \ \left(p^{k-1} - p^{\lfloor t/T \rfloor - 1}\right) \\ &= \left(p^{k-1} - p^{\lfloor t/T \rfloor - 1}\right)\left(\frac{(1-p)\Delta}{T} + p\right) \end{aligned}$$

We take the mean for k over $1 \dots r/T$,

$$\begin{aligned} Pr[\textit{one entry is inconsistent}] &= \frac{1}{r/T} \times \sum_{k=1}^{r/T} \left(p^{k-1} - p^{\lfloor t/T \rfloor - 1}\right)\left(\frac{(1-p)\Delta}{T} + p\right) \\ &= \frac{T}{r}\left(\frac{(1-p)\Delta}{T} + p\right)\left(\frac{1-p^{r/T}}{1-p} - \frac{r}{T} p^{\lfloor t/T \rfloor - 1}\right) \end{aligned}$$

When we assume $t \gg T$, the expression becomes

$$\begin{aligned} Pr[\textit{one entry is inconsistent}] &\simeq \frac{T}{r}\left(\frac{(1-p)\Delta}{T} + p\right)\left(\frac{1-p^{r/T}}{1-p}\right) \\ &= \frac{\Delta}{r}\left(1 - p^{r/T}\right) + \frac{T}{r}\frac{p}{1-p}\left(1 - p^{r/T}\right) \end{aligned}$$

We can see from these results that when the data change rate, r, is large, the same message gets announced multiple times and the loss probability goes down exponentially. We also can check this formula against our intuition for $r = T$ and find that the results match.

$$\begin{aligned} Pr[\textit{one entry is inconsistent}]_{r=T} &= \frac{\Delta}{r}(1-p) + \frac{T}{r}p \\ &= \frac{\Delta}{T}(1-p) + p \end{aligned}$$

From $Pr[\textit{one entry is inconsistent}]$ we calculate the expected number of incorrect entries in the registry,

$$\begin{aligned} E[\#\ \textit{incorrect entries}]_{r=T} &= \sum_{i=1}^{N-1} Pr[\textit{one entry is inconsistent}]_{r=T} \\ &= \left(p + (1-p) \times \frac{\Delta}{r}\right) \times (N-1) \end{aligned}$$

as well as the expected fraction of inconsistent entries in the registry, $E[\%\ \textit{inconsistency}]_{r=T}$, and consistent entries in the registry, $E[\%\ \textit{consistency}]_{r=T}$.

$$E[\%\ \textit{inconsistency}]_{r=T} = (E[\#\ \textit{incorrect entries}]_{r=T})/N$$

$$= \left(p + (1-p) \times \frac{\Delta}{r}\right) \times \frac{(N-1)}{N}$$

$$\begin{aligned} E[\%\ consistency]_{r=T} &= 1 - E[\%\ inconsistency]_{r=T} \\ &= 1 - \left(p + (1-p) \times \frac{\Delta}{r}\right) \times \frac{(N-1)}{N} \end{aligned}$$

In the next section, we validate these results with simulation. We concentrate on the results for $r = T$, as they serve as a worst case scenario.

4.3.2 Simulation Results

Each simulation was run 50-100 times using the `ns` simulator [6]. As in the Suppression simulations, we used a star topology with participating processes at the edges of the star, resulting in fixed delays between all processes. We observed the behavior of AL with between $2-10$ nodes, in increments of one node, and between $20-100$ nodes, in increments of 10 nodes. We also simulated loss probabilities between 0 and 1 in increments of .1.

We experimented with a fixed announce timer $T = T_A$, as well as with T adjusted by mean-zero white-noise; with fixed and random sample intervals for when to observe the simulation; and with snapshot and event-driven simulations (assess the state of the system at random points in time versus only when significant events occurred). The results for these configurations were qualitatively similar.

Therefore, for our experiments, we chose to use an announce timer T that was adjusted in each interval by mean-zero white-noise; the announce timer value was selected from the uniform interval $[.5T, 1.5T]$, rather than being kept rigidly "fixed" [28]. We did this to avoid bursts of congestion when all processes announced simultaneously. We calculated the state of the system by using event-driven simulations, where error in a listener registry was evaluated with the receipt of each message.

The exact content of each announcement message is shown below in Table 4.4. The `key` identifies the process making the announcement and the `value` is state shared by the process. The `seqno` provides a sequence number (message count) for the announcements from the process, allowing the listener to detect losses in the data stream, whereas the `version` tracks the number of times the content has changed since the process began announcing. We send both `seqno` and `version` to more accurately determine wrong state; it may happen that a message is lost but the data in an announcement does not change between the last and next message received. An announcement message also contains the time it was sent, from which the listener can determine round-trip time delay (see Section 4.9), as well as the expected time of the next announcement, from which the listener can adapt its cache entry expiration strategy to variable timer periods.

However, in the simulations discussed below, every announcement constituted a data change

Field	Description
`key`	process identifier
`value`	process state
`seqno`	sequence number
`version`	version of data
`send_time`	time message sent
`next_time`	anticipated next transmission

Table 4.4: **Announcement Message Content.**

$(r = T)$ and data was never expired from the cache $(k = \infty)$. The actual state stored by each listener about each announcer was extremely simple; a key-value pair consisting of a unique source identifier and the data version number $<$ *source id, data version#* $>$. We were not interested so much in the state information sent, as we were interested in the version of the state.

The registry was examined during "steady state," i.e., those times when N (the membership) was constant. In particular, the system was examined at a point after all processes arrived (within a very short time of $t = 0$) and before they departed (within a short time of an agreed-upon completion time). Thus, any inconsistencies in the system were due to transmission delays or packet loss rather than $A(t)$ or $D(t)$. We studied this behavior because it emerges in several existing applications. For instance rendezvous style meetings that are slated to begin (or end) at a set time and systems where nodes simultaneously reboot at initialization (or leave concurrently when there is a failure). They have the property that arrivals (and departures) cluster around specified times. In these systems, group membership remains fairly static in between session initiation and completion. We show some representative results below.

In Figure 4.7, we show a comparison of the simulation vs. the analysis, with $T = 1$, $\Delta = .1$, the data change rate $r = 1$ and $p = 0$; the simulation validates the analysis. This experiment was repeated by varying $2 \leq N \leq 100$, $.001 \leq \Delta/T \leq .5$, $0 \leq r \leq 10$, and the results matched under these circumstances as well.

Figure 4.8 displays the impact of increasing Δ/T (the network delay relative to the announcement period) on the level of simulated consistency, $E[c]$. As Δ/T increases, consistency decreases. This can also be seen in Figure 4.9, which depicts the change of Δ/T over several orders of magnitude. Holding Δ/T constant, Figures 4.10, 4.11 and 4.12 show that as r increases (i.e., the rate at which data change slows relative to the announcement periodicity T), consistency increases. When data changes with every announcement, $r = T$, and there is no loss in the system, $p = 0$, the probability of an inconsistent entry reduces to Δ/T. This analysis is validated in the simulations in Figures 4.10, 4.11 and 4.12, which are scaled versions of each other. The graphs show the effects of decreasing Δ/T on the expected level of consistency.

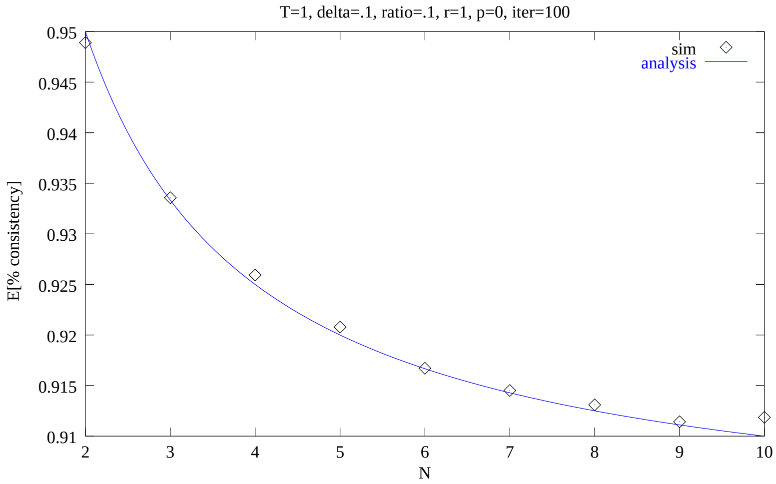

Figure 4.7: **Simulation vs. Analysis: Inconsistent State.**

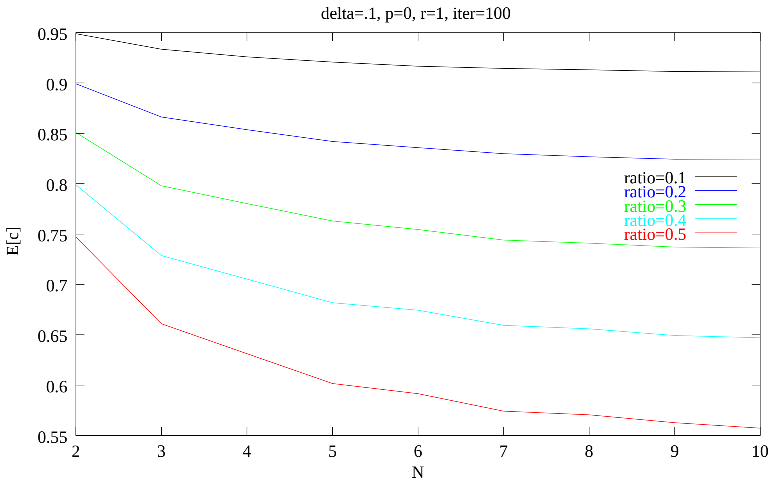

Figure 4.8: **Simulation: Inconsistent State (Large** $ratio = \Delta/T$**).**

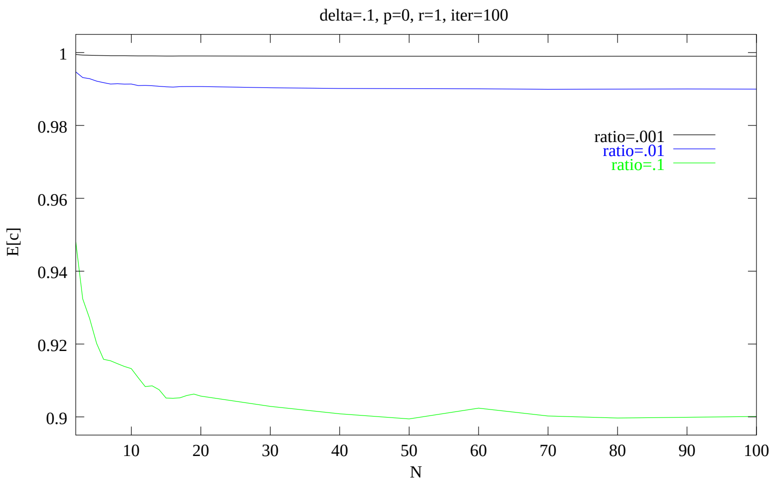

Figure 4.9: **Simulation: Inconsistent State (Small** $ratio = \Delta/T$**).**

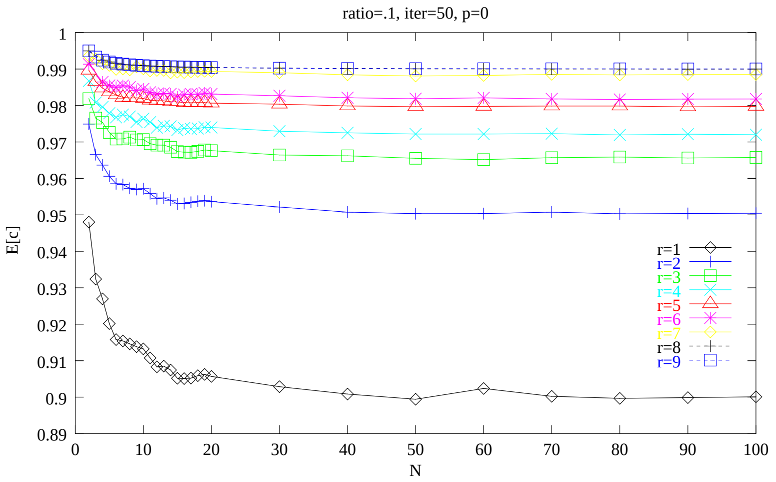

Figure 4.10: **Simulation: Data Change Rate (Δ/T = .1).**

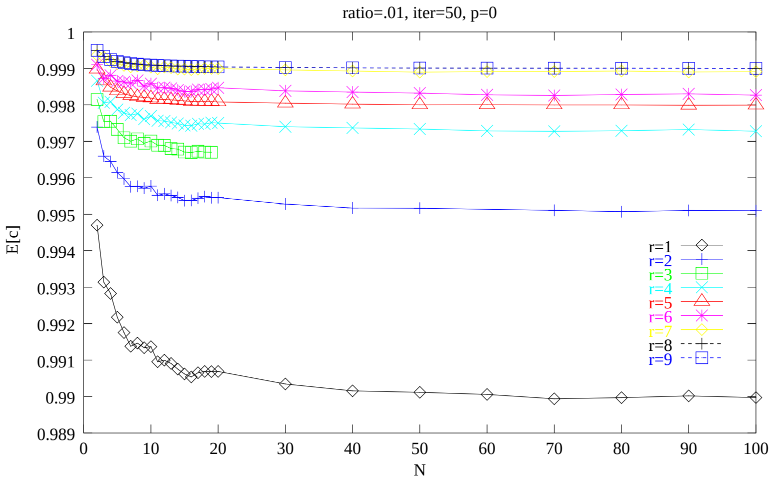

Figure 4.11: **Simulation: Data Change Rate (Δ/T = .01).**

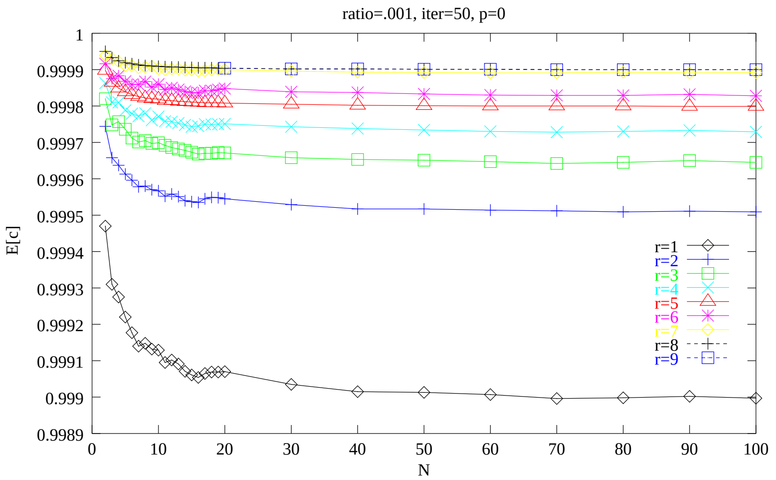

Figure 4.12: **Simulation: Data Change Rate (Δ/T = .001).**

4.4 Convergence Time

Convergence time is defined as the minimum time before which the Announce-Listen algorithm reaches a particular level of consistency. It is a function of group size, announcement periodicity, transmission delay and packet loss.

Consider the lossless case. When there is no loss and transmission delay is a fixed amount, Δ, announcements are trivially calculated to take Δ units of time to reach all listeners. Each announcement reaches destination registries simultaneously. When transmission delay varies between pairs of processes, the time it takes for an announcement to propagate to all registries is bounded by the maximum Δ_{ij}, where Δ_{ij} is the pairwise delay between the announcer process i and listener process j $(\forall\ i, j :: 0 \leq i, j < N)$.

When loss is introduced, the calculation for convergence time depends on knowing the number of attempts (retransmissions) it takes to send an announcement successfully. In this regard there is some resemblance to the Suppression metric for $E[\#\ messages]$. For Suppression, we ask how many messages does it take to suppress a group of N processes. In the case of Announce-Listen, we are interested in knowing how many attempts, $E[\#\ attempts]$, are needed for a given announcement to reach all N listeners.

Below, we provide a rough estimate of propagation time to provide some intuition about the metric. On average, each announcement reaches only a subset of the listeners of size $(1-p)N$. So we can determine the number of iterations i or rounds of repeat announcements it takes to cover the whole set. On average, the 1^{st} message reaches $N(1-p)$ processes, the 2^{nd} reaches $N(1-p)p$ additional processes, the 3^{rd} reaches $N(1-p)p^2$ and so forth. Therefore, after i attempts, the number of listeners that the message has reached is $N(1-p^i) = N(1-p) + N(1-p)p + \ldots + N(1-p)p^{i-1}$, whereas the number of listeners the message has not reached is Np^i.

Let us define ε as the acceptable fraction of the membership N that goes unreached; ε therefore defines a target level for inconsistency. The goal for convergence is for the fraction of unreached membership to be less than the acceptable level of inconsistency, or $Np^i < \varepsilon N$. When we solve for i, and note that $0 \leq p \leq 1$ and $0 \leq \varepsilon \leq 1$, the number of iterations it takes for Np^i to get sufficiently small is

$$E[\#\ attempts] \quad > \quad \frac{log(\varepsilon)}{log(p)}$$

For instance, if the goal is for all processes to hear a given announcement, i.e., that the number of unreached processes should be less than one $(Np^i < 1 = \varepsilon N)$, then $\varepsilon = 1/N$, and the number of attempts must be larger than

$$E[\#\ attempts] \quad > \quad \frac{log(\frac{1}{N})}{log(p)}$$

When it is acceptable for a percentage of processes not to receive the announcement, e.g., $\varepsilon = .1$,

$$E[\#\ attempts] \quad > \quad \frac{log(.1)}{log(p)}$$

Once $E[\#\ attempts]$ is parameterized properly, the time it takes for the registry to reach a consistent state, $E[convergence\ time]$, is bounded in the worst case by the time between the first and last announcement, plus the one-way transmission delay for the last message to arrive at the furthest listener (Figure 4.13):

$$\begin{aligned} E[convergence\ time] &= (E[\#\ attempts] - 1) \times T_A + \Delta_{max} \\ &= \left(\frac{log(\varepsilon)}{log(p)} - 1\right) \times T_A + \Delta_{max} \end{aligned}$$

This is an oversimplification because we have assumed that the content of the message does not change between announcements, which may not be the case. If there is a premium on having the registry attain a certain level of consistency by a given time (i.e., having announcements reach a certain percentage of listeners), but each announcement contains new content ($r = T_A$), then it may not be possible to reach a certain level of consistency given p and Δ. In that case, it will be necessary to adjust r or to use an alternate form of dissemination than AL (i.e., a more reliable transport protocol).

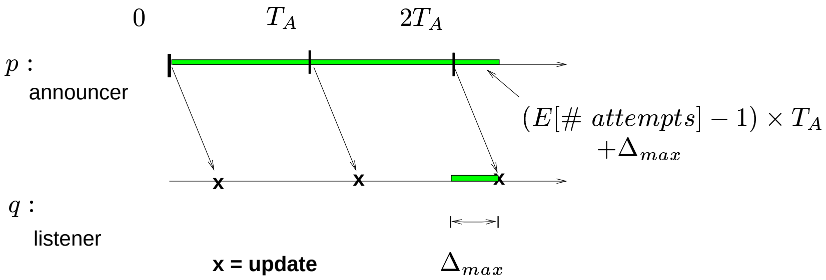

Figure 4.13: **Convergence Time.**

4.5 Messaging Overhead: Bandwidth

How does Announce-Listen consume network bandwidth? There are two aspects to answering this. First there is the question of how much bandwidth does AL require to achieve a certain level of consistency. Second, there is the question of how much of the bandwidth is wasted, i.e., is consumed by messages that are not necessary or that are redundant.

Let us consider the operational requirements given the periodicity of data changes, r. In Section 4.4, we derived $E[\#\ attempts]$, the number of messages required by AL to achieve a level of consistency of $1 - \varepsilon$. We use that knowledge to determine $E[\#\ wasted]$, the number of messages unnecessarily sent.

- **No Change.** As stated in Section 4.2.2, if an announcer process begins at time 0, then by time t the process has sent $n = \lfloor t/T_A \rfloor$ messages. When there is no change in the announcements, $E[\#\ wasted] = \lfloor t/T_A \rfloor - E[\#\ attempts]$.

- **Each Announcement is an Update.** If each announcement is an update of the state information ($r = T_A$), then all announcements are necessary and no bandwidth is considered wasted; $E[\#\ wasted] = 0$. The caveat is that if p is high, there may be a permanent level of inconsistency in the registry, because pN of the processes never receive the announcement.

- **Data Changes Slower than T_A.** When $r > T_A$ (and r is an integer multiple of T_A), there will be r announcements that repeat the same data before the content in the announcement changes. If $r > E[\#\ attempts]$, then there will exist $E[\#\ wasted] = E[\#\ attempts] - r$ rounds of redundant announcements.

So far, we have assumed no new arrivals into the system. Static announcements are always useful for newcomers in the case where $r > T$. The implications are (1) the $E[\#\ wasted]$ calculations above should be adjusted to reflect the number of new members during each announcement period, and (2) after $E[\#\ attempts]$ messages are sent, an announcer should adjust r to be a function of $A(t)$. One way to accomplish this is to actually adjust T_A instead. Of course, convergence time would now be a function of any new announcement periodicity.

Now let us discuss the impact of multicast on the level of redundancy of each announcement. An announcement is deemed entirely unnecessary and a waste of bandwidth if it does not update the registry entry for *any* listeners. Unnecessary messages are akin to extra messages in the Suppression algorithm. However, it is more complicated to determine which messages are actually necessary, due to the multicast nature of announcements. Due to message loss, some listeners will receive an announcement, while others do not. If the same announcement is re-issued, even though the announcement provides a useful update for those listeners who did not receive the announcement earlier, it will be redundant for others in the group who have received it already. Therefore, bandwidth can be wasted not only as a result of the lack of data change in announcements, but also due to the multicast nature of the distribution tree. Therefore, $E[\#\ attempts] - 1$ of the required messages for consistency are only partially necessary: in other words, each time a message is sent, on average, the message reaches only $(1-p)N$ of the nodes and does not reach pN of the nodes. The first message is necessary, but the subsequent messages are potentially redundant for any nodes that have already received it. Although we do not present an exact result here, we raise the issue for comparison with the non-multicast case.

On the other hand, if a process that receives an announcement unsubscribes from the group address immediately afterwards, then there is no redundancy.

4.6 Memory Overhead: Expiration Strategy

The cost of the registry depends in part on the data-aging model, which directly affects memory usage. The data expiration strategy can age data slowly (when k is large) or quickly (when k is small). Increasing k by adding more `Not Sure` states to the algorithm leads to:

- **Fewer False Negatives.** As a consequence of a longer expiration period, a listener waits longer before it declares an announcer `Departed`. This helps those processes that are actually alive, but whose announcement(s) might be experiencing packet loss or delay due to congestion.

- **More False Positives.** If an announcer stops announcing (because it has departed), then the listener holds the information longer. If false positives are costly to store or a listener needs to hear about departures in a more timely fashion (needing to know before k rounds of timeouts), then an explicit `Departed` announcement can offset the delay [13] [54].

Another way to look at the k expiration states is to think of them simply as providing a longer expiration timer. The aggregate expiration time (the total time an entry is cached, kT_L) should be long enough that an entry is not falsely removed because of delay or loss in the network. Therefore, to maintain a high level of consistency, an entry should remain in the cache at least as long as $E[\textit{convergence time}]$. Thus a policy to maintain high consistency sets the cost function to be small for recently stale data (within this bound), but increases it beyond this point; a consequence is that entries are expired and removed, but not before knowing definitively that an agreed-upon value has been reached.

4.7 Related Work

The idea of examining Announce-Listen probabilistically first appeared in [53]. At that time, the phrase Announce-Listen was coined to describe the class of protocols that rely on sender processes to periodically announce data and receiver processes to passively listen for updates. In this thesis, we specifically extend that work to more formally consider the range of errors that can arise in distributed registries built from AL. In addition, we incorporate other key parameters into the analysis that contribute to the operation of the algorithm. The goal has been to expose the principal variables affecting the system, in order to predict how they will behave under different operational conditions.

Raman et al. [49] develop an analytic model for the AL primitive based on classed queueing networks [7]. The focus of their research was to understand when adding feedback to Announce-Listen is beneficial for consistency. Although the model may be effective in capturing certain system phenomenon, it is entirely separate from the simulation results presented. Thus, although their model may be accurate, there are no simulations to validate it. In addition, as pointed out in their paper [49], the model is not analytically tractable when extended to perform two-level scheduling, which they propose based on the simulations.

One goal of our work is to conclusively validate our model and analysis through simulation. Once we can trust the model, it has predictive value. Given various parameters, we know how the system will behave. Given a particular performance metric to optimize, we know how to tune the system parameters to reach that optimization and the kind of impact those settings will have on other performance metrics.

There are other differences with the approach in [49]. We examine steady state, where the number of processes in the system is stable though the data in the announcements may change. Raman et al. do assign lifetimes to data, which translate to processes coming and going in our system. They characterize the system in terms of job rates and average behaviors, e.g., the rate at which new information arrives into, or information leaves the system, as well as the average probability that data is lost by one or more processes. We focus instead on individual announcements and individualized parameters, e.g., N, Δ, T_A, T_L, p, r, k. As a result, we directly expose the parameters that affect the performance of the algorithm, and we can differentiate between individual announcer's parameters and hence between policies toward individual announcements. For example, the expiration time for each entry is presented as a given at the outset of their model. By exposing p, k and T_L parameters in our model, we can understand how long it is appropriate to cache a value before removal. This allows us to correlate memory and consistency requirements. In general, by creating a model with parameters exposed, then we can adapt them as need be.

To combat the problem that some announcements are repeated ($r > T$) and will consume bandwidth unnecessarily, they discuss creating two separate queues. One to announce new data and another to announce old data, i.e., already announced at least once in the past. In their system, data migrates from new to old after the very first time it is sent. Afterwards, old data is only retransmitted upon request. Their research explores the appropriate ratio of bandwidth to allot to the two types of data.

Our analysis technique suggests that the appropriate point for the new-to-old transition should occur after $E[\# \textit{ attempts}]$ announcements, or beyond $E[\textit{convergence time}]$, the point in time that all N listeners have received the data. This would save the retransmission delay of the NACK to be received and queueing delay to switch the data from old to new queue.

Both studies are interested in the consistency metric as a driving force behind AL. However, the metrics for latency slightly differ: they define an average latency from the instant a new or updated key-value pair is introduced into the system to the first time it is received correctly; we define the convergence time as the average time beyond which an updated key-value pair has successfully reached all participating processes. This difference results because we are considering an N process system, whereas they consider the behavior of two processes.

Sharma et al.[55] focus on adapting the periodicity of announcement messages, in order to keep bandwidth usage below a fixed threshold. They propose techniques for receivers to estimate senders' update intervals, both of which effectively reduce bandwidth usage over the lifetime of announcers. As there is typically a correlation between announce timers and listen timers, the techniques to adapt announce timers are coupled with modifications to the expiration strategies of receivers. Although our work does not investigate the usage of adaptive announce timers, adaptive timers are complementary in spirit to the goal of proper parameterization of the AL algorithm and could be incorporated into the model proposed in this chapter.

4.8 Summary of Results

In this Chapter, we discussed the usage of the Announce-Listen algorithm as a means to disseminate state information among groups of processes and to create a replicated distributed registry. We highlighted that AL is part of the protocol spectrum that does not rely on any feedback mechanisms for communication among processes.

We identified several metrics to gauge the performance of such an algorithm: consistency, convergence time, network overhead and memory usage. We focused primarily on the metric of consistency and pinpointed three different types of errors that arise and work against it: false negatives, false positives and inconsistent state. We derived a model of the probability of these types of errors occurring and calculated the likelihood of any given registry entry being in error, $Pr[one\ entry\ is\ inconsistent]$. From this, we were able to deduce the overall consistency in a given registry, $E[\%\ inconsistency]$, and the overall consistency of a collection of distributed registries participating in the Announce-Listen algorithm.

We cast the other metrics in terms of how they meet consistency requirements. We presented an analysis of the convergence time, $E[convergence\ time]$, as a function of the number of attempts it will take to reach N participating registries, $E[\#\ attempts]$, noting that parameters for r and T_A may need adjustment to attain a given level of consistency. Based on the update periodicity, r, the network overhead metric, $E[\#\ wasted]$, can be expressed in terms of $E[\#\ attempts]$ as well. In our discussion of memory overhead, we described a method to parameterize the cache so that entries were not prematurely expired: ensure that $kT_L > E[convergence\ time]$.

Our main contribution with regard to these metrics is establishing a working model that can be used to parameterize them. Although other researchers have studied some of these metrics in the past, they have done so only with simulation and without the accompanying analysis to support or predict their findings. In addition, our new analytic model for AL exposed several key parameters not accounted for in previous models: timers both for announcement periodicity and cache expiration, transmission delay, and group size. With a sound model in hand, algorithms based on AL can be fine tuned to operate comfortably within the range for which they were designed.

4.9 Future Work

Extensions to the Model: Sporadic Listening, Proxies and Hierarchies. The model thus far has assumed that announcements are periodic and that listening is continuous. What happens if the listening interval is bounded? An example of such a system is a sensor network where sensors turn listening on briefly, then off again in order to conserve power. By giving up persistent listening, a process may need to rely on other processes more capable of listening full-time. When the intermittent process awakens, it can request updated information from the proxy cache.

Such a process may also require that a proxy make announcements on its behalf. This leads to the idea of proxy processes that not only collect, but also re-distribute information on behalf of multiple processes. A proxy announcer becomes a secondary (versus primary) source of information. However, in a distributed registry that includes proxy services, multiple announcers may announce the same information, and an algorithm for conflict resolution will need to be devised.

We are interested in investigating the tradeoffs between the various conflict resolution approaches: to embed additional information in announcements such as timestamps, versioning of data, or the "distance" from the actual information source. Distance might measure how many levels of indirection exist between the node that owns the information and the one propagating the information, or supply a topology measurement of delay time or number of hops. Are there substantial differences in the impact on the consistency in the registry? Furthermore, we want to understand the effect of redundant announcement messages on consistency, i.e., the likelihood that process p is in $q.registry$, when it should or should not be.

Finally, to what extent can we build a hierarchical multicast registry, and parameterize it based on the metrics we have derived in the non-heirarchical case? Are the metrics additive?

Approximate Knowledge. In the future, we would like to explore alternate caching strategies. The idea is to store more state information than a single key-value pair. A simple approach is to extend the cache to store more than one data value per entry. Cache policies might include:

- Cache j values per announcer in general, where j is small or is a function of memory constraints.
- Store all observed data values within the aggregate expiration timeout period.

By storing multiple key values, the registry can track the percentage of time a given announcer spends in each state. When the registry is queried for information pertaining to a given process, these statistics might dictate the order in which it returns the possible options and that the application subsequently tries each of the entries. The registry could also track the degree to which state values change, oscillate or go unreported. These statistics would help to estimate the proper registry response when a query for information follows the recent expiration of data.

Although there is no way to distribute updates more quickly than T (thus we assume $r \geq T$), we can use the knowledge of past history to predict the "direction" in which the data is going (e.g., akin to estimating the trajectory of an object) based on where it has been in the past. For example, if the state information for a process cycles through a small set of values, past history may allow us to predict the next value, even in the face of expired data.

Estimation of Transmission Delay. We have made the assumption that transmission delay, Δ, is easy to determine and that there are out-of-band means to calculate it. However, this assumption may not be true, nor is a fixed Δ value necessarily reliable.

We propose to explore the use of announcements themselves to derive estimates for one-way transmission delays between processes. This, in turn, could be used to parameterize the Announce-Listen algorithm, as well as other algorithms when AL is coupled to them (e.g., as in the case of Leader Election, Chapter 5).

There are really two types of Δ values of interest to each process: pair-wise Δ, the pair-wise transmission delay between a given process and every other process, and group-wise Δ, an average transmission delay that is calculated from all the pair-wise values a given process has collected. In order to make these estimations, a process must be both an announcer and a listener. We can imagine at least three uses for Δ calculations:

- **Parameter Adaptation.** Because, realistically speaking, Δ is not rigidly fixed, we can use Δ estimates to adapt the parameters of the AL algorithm itself. For example, if the registry convergence time must be below a given threshold, and p and Δ_{max} are given, then adjusting T_A may assist with that. In the future, we would like to understand the utility of incorporating this type of feedback into the operation of the algorithm. Although there exist many other adaptive multicast algorithms, we have not seen this technique incorporated into AL for the express purpose of consistency or convergence times. Sharma's work [55] focuses on adaptive timers to optimize bandwidth metrics (to keep usage below a given threshold).

- **Process Selection.** The idea is for each process to derive pair-wise Δ values from announcements (we explain the process below), then to calculate a group-wise Δ estimate, and subsequently to share the group-wise Δ information within announcements with other processes.

On receiving other process' information, a listener can compare its group-wise Δ with other processes' calculations. A process will be able to assess its relative "distance" from the rest of the group, i.e., if it is an outlier or topologically close to most of the processes. Outliers are less likely to receive announcements in time, and are therefore less likely to be consistent and more likely to remain inconsistent longer. Under the Suppression algorithm (Chapters 2 and 3), they generate more extra messages, whereas under Leader Election (Chapter 5) they are more prone to falsely elect a leader and less likely to re-elect in a timely fashion.

A process can use the group-wise calculations to determine if it makes sense for it to become an early awakener in the Suppression algorithm or a leader in the Leader Election algorithm. In short, if a process's group-wise Δ lies above other processes' group-wise estimate, then it refrains from responding quickly to SUP or LE, i.e., to try to suppress other announcements, or to become leader. Likewise, if a process is positioned well within the group of processes, its Suppression timer is shortened for SUP and LE.

In [53], a similar technique was used to calculate group-wise ttl's in order to scope multicast sessions properly.

- **Subgroup Establishment.** We could also use the group-wise Δ calculation to encourage listeners with nearly identical values to become associated with the same "class," where all the processes within a "class" share network characteristics and belong to a separate subaddress. From our Suppression analysis, we know how fixed-Δ systems behave and could predict and parameterize these groups well.

  In addition, the diameter of the processes within a "class" is now well-known and could be used to establish at which layer in the multicast registry hierarchy a subgroup belongs. The expectation is that, now that the groups are homogeneous in some network parameters, metrics pertaining to consistency are additive/subtractive as we go up/down the registry hierarchy.

The process for calculating pair-wise Δ values is as follows. Listeners are already caching information from specific announcers. If announcers were to include the local time that the message was sent, as well as the sequence number of the announcement (i.e., the number announcement since time $t = 0$), then a listener could echo this information back to the announcer when it becomes time for the listener to announce its own information. In addition, the listener must include how much time passes since the receipt of the announcement and when the listener issues its next announcement. Because announcements are multicast, the calculations targeted for different listener processes will be included in the same messages.

In Figure 4.14, we show that even though the exchange of announcements between two processes are asynchronous, it is now possible to ascertain round-trip times, and subsequently approximate Δ from that. Node 0 sends an announcement at time s_0, which is received at listener process 1 at time

r_0; the announcement includes time s_0. When node 1 subsequently makes its own announcement, it echoes back the time s_0 as well as r_0 to node 0, timestamping its own message as being sent at time s_1. On receipt of the announcement from node 1, node 0 is able to approximate the one-way Δ between the two processes:

$$\Delta_{01} \approx ((r_1 - s_0) - (s_1 - r_0))/2$$

Likewise, when node 0 issues its next announcement, it piggybacks the necessary information for node 1 to obtain Δ_{10}. Variants of this technique are used in such protocols as RTCP and SRM [54][27].

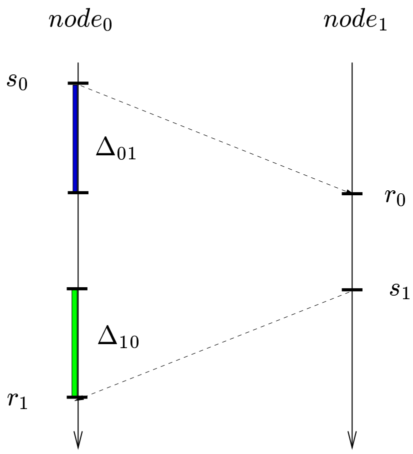

Figure 4.14: **Roundtrip Time Estimate: Calculating Pair-wise Δ.**

ACK- and NACK-based Schemes. We would like to identify the conditions under which multicast AL outperforms ACK/NACK messaging schemes, or vice versa. More specifically, we want to find the r for which ACKed messaging is a better choice than AL. By "better," we mean to evaluate the performance of different metrics, such as consistency, convergence time, and overhead (network and memory).

Chapter 5

Leader Election

The goal of the Leader Election (LE) algorithm is for a group of processes to select one of the processes to serve as its *leader*. LE is a classic distributed system problem that arises in many different venues. It has been used to establish a coordinator for re-organization after system failures [31], a central lock coordinator in distributed databases [43], a primary site for a distributed file system [5], k-resiliency in a reliable distributed system that can withstand k process failures [12], and the lead router to track host interest in multicast groups [13].

The goal is to converge to exactly one leader – no more, no less. Therefore, all processes must recognize the same process as the leader. They accomplish this by using a global leader selection criterion, for example by picking the process with the largest address, the lightest load, the most neighbors, the earliest wake-up time, etc. It doesn't matter what the criterion, so long as the same comparison is done by all processes and it results in a single leader. Although such an algorithm could be used to elect k multiple leaders, e.g., the processes with the k largest addresses within the group, we do not consider that problem in this thesis.

The challenge in a loosely-coupled domain is for processes to agree upon a given leader without using strict consensus. Instead of using acknowledgment-based messaging, processes send announcement-style messages as with SUP and AL. The point is for the group to converge upon a single leader after a small number of rounds of announcements, and, although LE may lead to periods of uncertainty due to transmission delay and packet loss, for leadership to stabilize eventually. In this chapter, we analyze the probabilistic consensus reached through LE and the resulting performance tradeoffs.

We explore two forms of leader election: the most basic, Leader Election using Announce-Listen (LE-A); and Leader Election using Suppression (LE-S), a refinement to the basic algorithm that inserts a phase of suppression before a leader begins making announcements. By contrasting these two methods, we unequivocally show the importance of Suppression as a scalability technique for group communication.

We identify several metrics that characterize the performance of these algorithms: Leadership

Delay, the delay until leadership is established; Leadership Re-establishment Delay, the delay until leadership is re-established after group membership changes; the Number of Messages Generated by LE; and Inconsistent State, the fraction of time processes are in disagreement about the leadership. We study the behavior of the metrics with no loss, correlated loss and uncorrelated loss. We also examine their behavior when all processes begin simultaneously, when group membership remains constant, when processes join and when they depart.

It is common practice for Leader Election algorithms to define the leader as the process with the greatest address. Although it is a widely used approach in many LE algorithms [31] [2] [25] [16] [48] [13], we reveal through analysis and simulation the critical need to consider alternate leader selection criteria for multicast-based Leader Election algorithms.

What is particularly interesting about Leader Election is that it is composed out of other techniques or slight variations of them. Consequently, we can assess its metrics in terms of the metrics for components we already have analyzed. Therefore, we compare the performance metrics for Leader Election with the bounds we have already established for the Suppression and Announce-Listen algorithms.

Before concluding this chapter, we present a summary of our findings, contrast our research with prior art, and present ideas for future work in this area.

5.1 Basic Algorithm

We explore two approaches to Leader Election. They vary in terms of how a process begins the algorithm.

If all processes were to wait to hear an announcement from the leader, then all processes might end up waiting indefinitely, with none making forward progress. Therefore, a process eventually must declare leadership, even if leadership only lasts temporarily. In both of the LE algorithms that we examine, when a process starts participating, it always assumes it is the leader, until such time as it hears an announcement from another process that challenges its leadership.

In the simplest form of the algorithm, Leader Election using Announce-Listen (LE-A), processes announce their leadership immediately, whereas with the Leader Election using Suppression (LE-S) form of the algorithm, processes delay leadership announcements until a Suppression phase completes.

In either case, while a process believes it is the leader, it announces this fact periodically to the rest of the processes. However, during times of flux, there may be multiple processes that believe they are leader. When a process that believes it is a leader receives an announcement from another process, a conflict resolution algorithm is needed. For the purpose of the examples throughout the chapter, we will use the standard method of conflict resolution that chooses a leader based on greatest address. Later in the chapter we also discuss the benefits of using alternate leader selection criteria.

Thus, if the announcement message received has a greater sender address than the process receiving the message, then the receiver gives up leadership. The receiving process defers to the new leader by refraining from sending further announcements. If a non-leader process no longer receives leadership announcements, it times out and begins the algorithm anew.

5.1.1 The Simplest Case

The simplest Leader Election algorithm is described by the pseudocode in Program 5.1.1 and the accompanying state transition diagram as shown in Figure 5.1. As with earlier state transition diagrams, we use the convention that state transitions are labelled with $\frac{A}{B}$, where A is an event that occurs, and B is the consequence of the event occurring.

```
LEADER-ANNOUNCE (T_A, T_L)
leader = me.addr
announce (I am leader)
set_announce_timer (T_A)
do
    if receive_message (msg) then
      if msg.addr > leader then
        leader = msg.addr
        set_listen_timer (T_L)
        clear_announce_timer ()
    if announce_timer_expired () or
      listen_timer_expired () then
      leader = me.addr
      announce (I am leader)
      set_announce_timer (T_A)
```

Program 5.1: **Leader Election Algorithm using Announce-Listen.**

We refer to this form of the algorithm as Leader Election using Announce-Listen (LE-A). Each process begins by immediately announcing its own leadership, then setting the announce timer, T_A, to remind itself when to send the next announcement. A process then waits for one of several events to occur:

1. If an announcement is received from another process with an address greater than the current leader, the process updates the leader address and sets the listen timer, T_L, which will detect when the leader departs. If the receiving process was the old leader, it reverts to listening and clears its announce timer, as it no longer should be sending announcements.

2. If the announce timer expires, the process reaffirms its leadership, sends an announcement to the other processes, then resets the announce timer again.

If the listen timer expires, the previous leader has stopped announcing leadership, either because it was preempted by another process or because it has left the system. At that point, the leader election procedure is restarted. Any process that detects leaderlessness appoints itself leader, sends a leadership announcement and resets its announce timer, just as if it had been the leader all along.

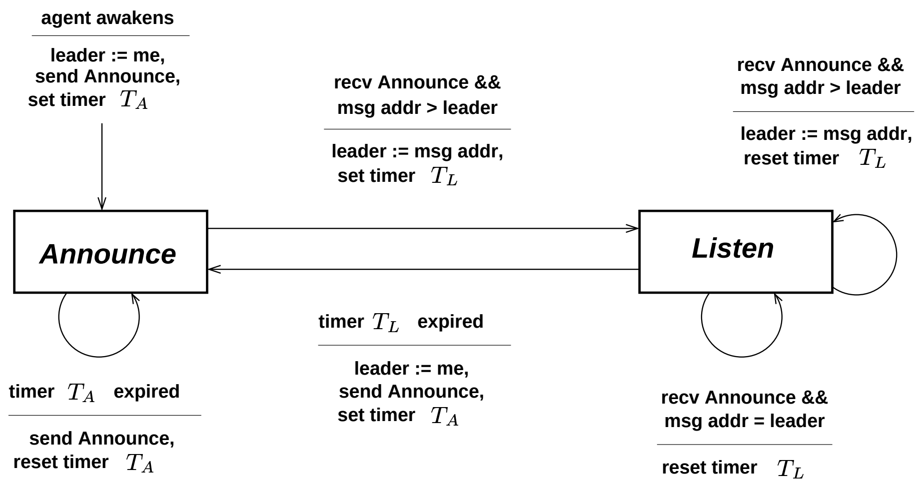

Figure 5.1: **Announce Leadership.**

If every process starts at the same time, each will propose itself as the leader and the algorithm will generate N messages, where N is the number of processes participating in the algorithm. In the lossless case, an agreed-upon leader will be elected after a single round of the algorithm. However, when messages can be lost, multiple rounds of conflict resolution will be necessary to reduce the number of leaders and to converge on a single leader.

5.1.2 Leader Election Refined

To avoid excessive message generation as well as message collision when all processes awaken simultaneously, each process may suppress sending its initial leadership announcement. Thus, this form of the LE algorithm combines Announce-Listen with Suppression. To differentiate it from the previous algorithm, we refer to it as Leader Election with Suppresion (LE-S). The pseudocode for the algorithm is given by Program 5.2 and the accompanying state transition diagram is shown in Figure 5.2.

Before a process issues its first message, it selects a random time t from the distribution d using a Suppression timer interval T_S, and sets the suppress timer. If an announcement *with a greater address* has not been received by time t, the process assumes leadership and begins to send announcements. A process then waits for one of several events to occur:

1. If an announcement is received from another process with a greater address than the current leader, the process updates the leader address and sets the listen timer, T_L, which will detect when the leader departs. The process begins listening for heartbeat messages from the leader.

   If the receiving process was the old leader, it reverts to listening and clears its announce timer, as it no longer should be sending announcements. The process also clears its suppress timer, in case the message was received while the process was waiting for its suppression timer to expire.

2. If the listen timer expires, the leader election process begins anew. The local process selects a random time t to sleep and resets the suppress timer.

3. If the announce timer expires or the suppress timer expires, the process reaffirms leadership, sends an announcement, then resets the announce timer. If the suppress timer expires, then we know that no other process challenged the process' leadership in the last t units of time.

```
LEADER-SUPPRESS (T_A, T_L, T_S d)
    leader = me.addr
    t = random (d, T_S)
    set_suppress_timer (t)
    do
        if receive_message (msg) then
            if msg.addr > leader then
                leader = msg.addr
                set_listen_timer (T_L)
                clear_suppress_timer ()
                clear_announce_timer ()
        if listen_timer_expired () then
            leader = me.addr
            t = random (d, T_S)
            set_suppress_timer (t)
        if suppress_timer_expired () or
            announce_timer_expired () then
            announce (I am leader)
            set_announce_timer (T_A)
```

Program 5.2: **Leader Election Algorithm using Suppression.**

The main difference from Suppression as discussed in Chapters 2 and 3 is that a process is not suppressed by the arrival of a message from merely another process; the message must have originated from a process with a greater address.

5.1.3 A Note about Timers

The Leader Election algorithm relies on several timer values: an Announce timer, T_A, represents the periodicity of leadership announcements; a Listen timer, T_L, represents the interval after which

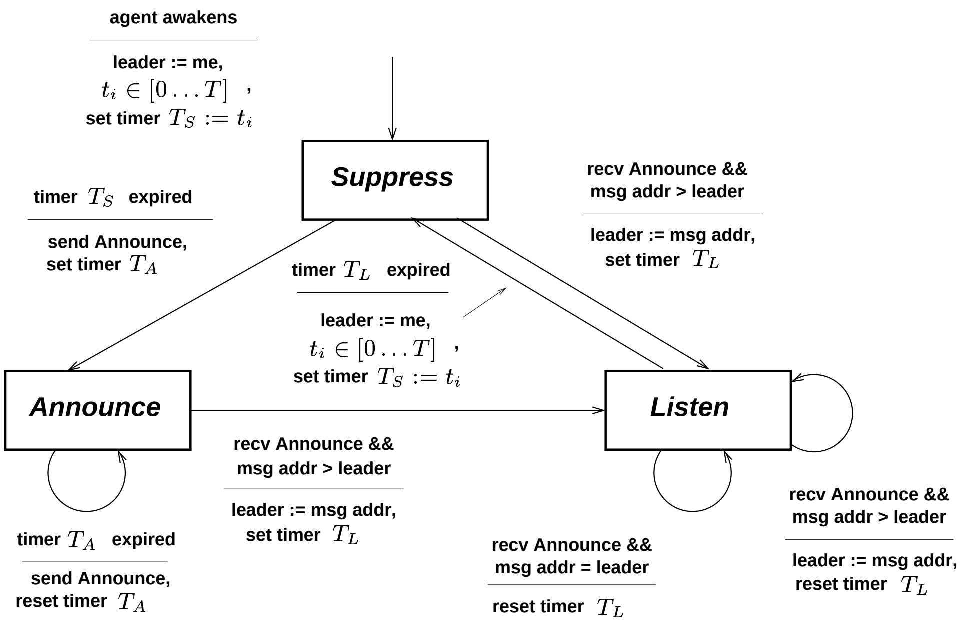

Figure 5.2: **Suppress Leadership.**

a process decides it has missed an announcement it was expecting[1]; a Suppress timer, T_S, is the upper bound of the Suppression interval from which a process selects a time to awaken.

Note that T_L is typically a small integer multiple of T_A, and T_S is meant to be large enough for most processes to hear a leadership announcement before sending their own, yet small enough not to delay consensus on the true leader.

5.2 Scalability

The way in which the Leader Election algorithm provides scalability is to appoint (ideally) a single leader process that acts on behalf of the other processes. As the representative of the group, the leader issues messages, so that the other members do not have to. In addition, LE offers all of the benefits already enumerated for Announce-Listen, as well as the benefits of Suppression in the case of LE-S: it reduces the number of overall messages generated, decreases the likelihood of implosion, adds asynchrony to group transmissions, removes the tight feedback loop required for processing ACKs and NACKs, and eliminates the delay waiting for their retransmission.

5.3 Metrics

To evaluate the performance of Leader Election, we explore several possible metrics. The metrics are reminiscent of the metrics derived to evaluate SUP and AL, but they are recast in terms of

[1]As in the AL algorithm, we could gradually age leaders, but for simplicity we are not doing that here.

leadership establishment. Although not a complete list, the ones below are representative of the kinds of performance issues that are important for efficient operation of LE.

- **Leadership delay.** Leadership delay is the amount of time it takes until an agreed-upon leader is established. Leadership is established when the announcement message sent by the process with the largest address has been received by all other group members. This metric determines the average time taken by the LE process to complete or to stabilize.

- **Leadership re-established delay.** Leadership must be re-established after a perturbation in the system, such as when the leader leaves, a new leader arrives, or when messages lost in the network cause a temporary outage of communication between the leader and other processes. To be precise, leadership is re-established when all processes agree upon the new leader. This metric determines the average time taken by the LE process to re-stabilize after a perturbation in membership.

- **Number of messages generated.** When a process is a leader, it periodically announces its leadership. The number of messages generated in a given interval (of size T_A) equals the number of processes announcing leadership during that time.

- **Number of simultaneous leaders.** We can estimate the number of leaders in a given interval by determining the number of announcement messages generated in that interval. Therefore, we focus on the number of messages generated and use that as an estimate.

- **Inconsistent state.** A process could have inconsistent state in three cases: (a) when the process that has been elected as the leader has left the system, but leadership has not been re-estabilished yet, (b) when a new leader process arrives into the system, but all processes have not yet reached agreement on this fact, or (c) a process falsely believes the leader has departed (because its listen timer expires) but the leader has not actually left.

In the upcoming sections, we discuss the performance of these metrics when there is: no loss, correlated loss and uncorrelated loss. We consider each metric under the condition that all processes begin simultaneously, as it presents the most stress on the algorithm's attempt to moderate the numbers of announcing processes. For the re-establishment delay metric, we consider additional scenarios: all processes are in a *steady state* (i.e., they remain in agreement about the leader), processes join the algorithm, and processes depart.

Our model of the network is identical to that used in previous chapters, with parameters for group size, N, loss probability, l, and fixed transmission delay between processes, Δ. Unless indicated, the analysis assumes all processes use the LE-S algorithm, since LE-A can be derived from LE-S by setting $T_S = 0$.

It is also of particular interest to us that LE-S is a combination of Suppression and Announce-Listen. As such, we posit that its performance is bounded in the best case by the performance of SUP and AL. We show these comparisons in Section 5.4.

Because LE is not strictly the composition of SUP and AL, but rather a composition of variants of these algorithms, at times it behaves in a manner that is quite different from them. For example, in LE there is *always* a leader, as long as there are processes participating in the algorithm. As such, it is impossible to find the corollary of AL errors for false positives and false negatives in this context. A false positive would occur when a process has the false belief that there is a leader, when there is none. A false negative would occur when a process has the false belief that there is no leader, when there is actually a leader present. Since leadership is immediately established when the LE algorithm begins, and always reverts back to each local process when the leader departs, there is always a leader in the LE algorithm, and hence these conditions (of false positives and false negatives) do not arise in LE. However, when a process falsely believes the leader has departed (because it has not received a leader announcement message within the timeout period), and thus reverts back to itself as the leader, we consider this condition under the inconsistent state metric.

5.3.1 Leadership Delay

What is the expected delay until leadership is established? In other words, at what point in time does the number of leaders converge to exactly one? The assumption is that all processes begin simultaneously at time $t = 0$. We examine the case where processes arrive at later times ($t > 0$) in Section 5.3.2 when we explore re-establishment delay. We also assume that $N > 1$ otherwise there is no leadership delay. If one and only one process arrives into the system, it is immediately the leader as it requires no agreement to be reached with any other processes.

5.3.1.1 No Loss

In the case of the simple LE algorithm (LE-A), the system begins with N leaders when all N processes begin simultaneously. After time Δ has elapsed, only one leader remains.

$$E[\textit{leadership delay LE-A}]_{l=0} = \Delta$$

In the case of Leader Election with Suppression (LE-S), leadership will be established at time $t_s + \Delta$, where t_s is the Suppression wake-up time selected by the process with the largest address. On average, this time will be $E[t_s] + \Delta$, where $E[t_s] = \int_0^{T_S} tp(t)\, dt$.

$$E[\textit{leadership delay LE-S}]_{l=0} = E[t_s] + \Delta$$

5.3.1.2 Correlated Loss

Message loss in the network will cause larger leadership delay because it may take multiple announcements before an announcement from the leader reaches all processes. With correlated loss, if a message is lost, it is lost by all receivers. Conversely, if a message is received by a receiver, it is received by all receivers. The question is, how many announcements will it take before the leader's announcement is successfully received? Let i be the announcement number, where $i = 0$ corresponds to the first announcement sent by the leader. Then, the expected time for leadership to be established is given by:

$$E[\textit{leadership delay LE-A}]_{corr} = \sum_{i=0}^{\infty} Pr[\textit{announcement } i \textit{ establishes leadership}]_{corr} \times (\Delta + T_A \times i)$$

The likelihood of the first ($i = 0$) announcement from the leader being received by all processes is $(1 - l)$; in general, the likelihood of the i^{th} announcement establishing leadership is the probability that the previous announcements were lost but the i^{th} announcement was received, which is $l^i(1-l)$. Therefore, the expected leadership time is given by:

$$\begin{aligned}
E[\textit{leadership delay LE-A}]_{corr} &= \sum_{i=0}^{\infty} l^i(1-l)(\Delta + T_A \times i) \\
&= \sum_{i=0}^{\infty} l^i(1-l) \times \Delta + \sum_{i=0}^{\infty} l^i(1-l) \times T_A \times i \\
&= \Delta + \sum_{i=0}^{\infty} l^i(1-l) \times T_A \times i \\
&= \Delta + \frac{T_A l}{(1-l)}
\end{aligned}$$

When we have suppression, the only difference in this metric is that the leader process starts announcing at a time selected by the suppression algorithm. Therefore,

$$\begin{aligned}
E[\textit{leadership delay LE-S}]_{corr} &= E[t_s] + E[\textit{leadership delay LE-A}]_{corr} \\
&= E[t_s] + \Delta + \frac{T_A l}{(1-l)}
\end{aligned}$$

5.3.1.3 Uncorrelated Loss

With uncorrelated loss, different processes may lose different announcements from the leader. Thus, the difference between the delay calculation for correlated and uncorrelated loss is in the last term of the leadership delay formula. This term establishes the expected number of rounds it takes for the leader's announcement message to reach all processes. A detailed analysis of that difference can be found in Appendix B. Essentially, we establish that the probability for all N processes to have received a message in less than or equal to i attempts is $(1-l^i)^N$. As a result, we can derive the probability that all N processes have received the message in exactly i attempts is $(1-l^i)^N - (1-l^{(i-1)})^N$, the difference between the probability that all processes have received the message in less than or equal to i tosses and the probability that all processes have received the message in less than or equal to $i-1$ tosses. We incorporate the results below.

$$\begin{aligned}
E[\textit{leadership delay LE-S}]_{uc} &= E[t_s] + \Delta + \\
&\quad \sum_{i=0}^{\infty} Pr[\textit{announcement } i \textit{ establishes leadership}]_{uc} \times T_A \times i \\
&= E[t_s] + \Delta + \sum_{i=0}^{\infty} \left((1-l^i)^N - (1-l^{(i-1)})^N\right) \times T_A \times i \\
&= E[t_s] + \Delta + T_A \times \sum_{i=1}^{N} (-1)^{(i+1)} \binom{N}{i} \frac{1}{(1-l^i)}
\end{aligned}$$

5.3.2 Leadership Re-establishment Delay

The leadership re-establishment process is begun after either the departure of the leader or the arrival of a new process into the system. Departures are detected by the expiration of the listen timer, T_L. However, the listener cannot discriminate between when the leader actually departs or when the leader is falsely thought to have departed but has not, i.e., if the leader's announcement messages are delayed or lost in the system. We calculate the re-establishment delay under both circumstances, referring to the former case as *leader departure* and the latter as *false departure*. We refer to arriving processes as *late joiners* and assume they occur during the steady state, after there is already an established leader.

The calculations below also make the assumption that $N \geq 1$ after the leader departs and $N \geq 1$ before a process arrives. We assert that if the last process departs the system, then there is no re-establishment delay. Similarly, when the first and only process arrives into the system, there is no re-establishment delay. It becomes the leader immediately. Note that the departure of a non-leader process has no effect on the consensus of leadership among the remaining processes.

5.3.2.1 No Loss

Leader Departure. Consider the scenario depicted in Figure 5.3. The leader sends its k^{th} and last announcement at time kT_A. All listeners receive this heartbeat message from the leader at time $kT_A + \Delta$. The leader subsequently departs at some point, $kT_A + t_d$, within the announcement interval $[kT_A,\ (k+1)T_A]$. We calculate $E[t_d] = \int_0^{T_A} t \cdot D(t)dt$ from the departure function, $D(t)$ and

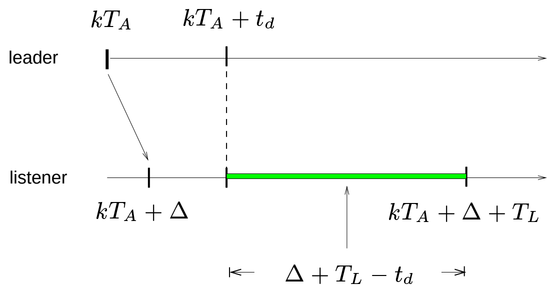

Figure 5.3: **Time to Notice the Leader Left (No Loss).**

assume $D(t)$ is Poisson, meaning in each interval it behaves similarly.[2]

The listeners time out after T_L and subsequently perform the Suppression phase of the algorithm, which amounts to the normal leadership establishment delay. The total time to notice that the leader left:

$$\begin{aligned} E[\textit{time to notice leader left}]_{l=0} &= E[(kT_A + \Delta + T_L) - (kT_A + t_d)] \\ &= \Delta + T_L - E[t_d] \end{aligned}$$

The total re-establishment delay is the time to notice the leader left plus the amount of time that the listener takes to hear from the replacement leader, $t_s + \Delta$, which is the time the new leader picks for its suppression timer plus the delay before which the listeners receive its leadership announcement. If the average time a leader picks for its suppression timer is $E[t_s]$, and the expected departure time is $E[t_d]$, the expression becomes:

$$\begin{aligned} E[\textit{re-establishment delay LE-S}]_{l=0} &= E[\textit{time to notice leader left}]_{l=0} + \\ &\quad E[\textit{leadership delay LE-S}]_{l=0} \\ &= (\Delta + T_L - E[t_d]) + (E[t_s] + \Delta) \\ &= T_L + 2\Delta + E[t_s] - E[t_d] \end{aligned}$$

False Departure. For leaderlessness to have been detected, the listen timer must have expired. The expiration of this timer occurs T_L after the last announcement was received from the leader. In the no loss case, this only occurs if the timer expired before the next announcement had a chance to arrive. The implication is that $T_L < T_A$, since we are assuming fixed transmission delay Δ. At the point when the timer expires, the local process reverts to itself as leader and begins the election

[2] Otherwise $E[t_d]$ would need to be averaged over all intervals: $E[t_d] = \lim_N^\infty (1/N)(\int_{T_A}^{0} D(T)dt + \int_{2T_A}^{T_A} (t - T_A) \cdot D(T)dt + \ldots + \int_{NT_A}^{(N-1)T_A} (t - (N-1)T_A) \cdot D(T)dt)$.

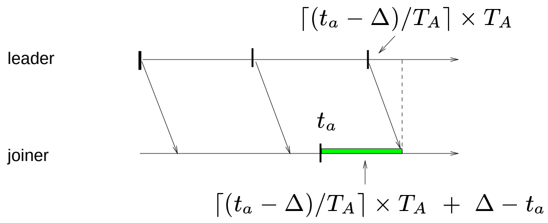

Figure 5.4: **Leadership Delay with a Late Joiner:** $joiner \neq leader$

process anew. Therefore, the re-establishment delay becomes the portion of the announcement interval that remains until the next announcement arrives,

$$E[\textit{re-establishment delay LE-S}]_{l=0,false} \quad = \quad T_A - T_L$$

Late Joiners. The system reacts differently to a late joiner depending on whether its address is smaller or larger than the current leader. If the late joiner has a larger address than the current leader and it arrives into the system at time t_a, then it will become the new leader at time $t_a + \Delta$. We calculate $E[t_a] = \int_0^{T_A} t \cdot A(t)dt$ from the arrival function $A(t)$ and assume A(t) is Poisson. We subtract the arrival time from the time when joiners' leadership announcement reaches the other processes,

$$\begin{aligned} E[\textit{re-establishment delay LE-A}]_{l=0,joiner>leader} &= E[t_a] + \Delta - E[t_a] \\ &= \Delta \end{aligned}$$

If the late joiner has a smaller address than the current leader, the late joiner must receive a message from the current leader before leadership is fully re-established (all group members agree on the leader), which happens when the next announcement made by the current leader reaches the late joiner. The previous announcement must have occurred before time $t_a - \Delta$; otherwise the late joiner would have been suppressed immediately on arrival. If the number of announcements that have occurred at a given point in time, t, is t/T_A, then the time until the late joiner receives the next announcement is $\lceil (t_a - \Delta)/T_A \rceil \times T_A + \Delta$. Therefore, the leadership delay is as displayed in Figure 5.4:

$$E[\textit{re-establishment delay LE-A}]_{l=0,joiner \neq leader} \quad = \quad E[\lceil (t_a - \Delta)/T_A \rceil] \times T_A + \Delta - E[t_a]$$

In the case of Leader Election with Suppression (LE-S), when a late joiner with a larger address arrives at time t_a, then it will become the new leader and begin announcing leadership at time $t_a + t_s$. The time at which the announcements reach the other processes is $t_a + t_s + \Delta$. The leadership delay is the same as if all processes began simultaneously:

$$\begin{aligned} E[\textit{re-establishment delay LE-S}]_{l=0,joiner>leader} &= (E[t_a] + E[t_s] + \Delta) - E[t_a] \\ &= E[t_s] + \Delta \\ &= E[\textit{leadership delay LE-S}]_{l=0} \end{aligned}$$

When a late joiner with a lower address arrives at time t_a, then the delay is equivalent to the same scenario in the LE-A case:

$$\begin{aligned} E[\textit{re-establishment delay LE-S}]_{l=0,joiner\neq leader} &= E[\lceil (t_a - \Delta)/T_A \rceil] \times T_A + \Delta - E[t_a] \\ &= E[\textit{re-establishment delay LE-A}]_{l=0,joiner\neq leader} \end{aligned}$$

5.3.2.2 Correlated Loss

Leader Departure. When there is loss in the system, the leader could depart **after** the listener timer has begun expiration already due to lost messages. When we examine the average number of consecutive lost messages, we find that

$$\begin{aligned} E[\#\ \textit{consecutive lost}]_{corr} &= E[\textit{announcement } i \textit{ establishes leadership}]_{corr} \\ &= \frac{l}{(1-l)} \end{aligned}$$

The time to notice the leader left is basically shifted by the term $(T_A \times E[\#\ \textit{consecutive lost}])$, as is the leadership delay we already derived in Section 5.3.1.2. Therefore,

$$\begin{aligned} E[\textit{re-establishment delay LE-S}]_{corr} &= E[\textit{time to notice leader left}]_{corr} + \\ &\quad E[\textit{leadership delay LE-S}]_{corr} \\ &= (\Delta + T_L - E[t_d]) - (T_A \times E[\#\ \textit{consecutive lost}]_{corr}) + \\ &\quad (E[t_s] + \Delta + \frac{T_A l}{1-l}) \\ &= T_L + 2\Delta - E[t_d] + E[t_s] \\ &= E[\textit{re-establishment delay LE-S}]_{l=0} \end{aligned}$$

Note that this is the same result as the lossless case! Basically, the gains made in the time to notice the leader left are negated by the increases in leadership delay. They cancel each other out.

False Departure. For leaderlessness to be detected, the listen timer at a receiver process must have expired T_L beyond the last successfully received announcement from the leader. In the lossy case, this arises when the listen timer is not long enough to withstand multiple message drops, i.e., $T_L \leq T_A \times E[\#\ \textit{consecutive lost}]$. The detection of leaderlessness happens T_L units into the interval $T_A \times E[\#\ \textit{consecutive lost}]$, and is corrected one announcement beyond that interval, so the re-establishment delay is:

$$\begin{aligned}
E[\textit{re-establishment delay LE-S}]_{corr,false} &= T_A \times E[\#\ \textit{consecutive lost}]_{corr} - T_L + T_A \\
&= T_A \times \left(\frac{l}{1-l} + 1\right) - T_L \\
&= T_A \times (\frac{1}{1-l}) - T_L
\end{aligned}$$

Late Joiners. When the late joiner with a larger address than the current leader arrives at time t_a, then it will become the new leader and begin announcing at time $t_a + t_s$. However, for all processes to agree on the leader, the joiner's announcement must reach all processes. With loss, we know on average that the number of announcements it will take for that to occur is $E[\#\ \textit{consecutive lost}]_{corr}$, and an additional Δ for the last announcement to reach all processes. Therefore, we use an adjustment term $(T_A \times E[\#\ \textit{consecutive lost}])$ once again:

$$\begin{aligned}
E[\textit{re-establishment delay LE-S}]_{corr,joiner>leader} &= E[t_a] + E[t_s] + \\
&\quad (T_A \times E[\#\ \textit{consecutive lost}]_{corr}) + \Delta - E[t_a] \\
&= E[t_s] + \frac{T_A l}{1-l} + \Delta \\
&= E[\textit{leadership delay LE-S}]_{corr}
\end{aligned}$$

When a late joiner with a lower address arrives at time t_a, then leadership is re-established once the late joiner receives the leader's announcement. In the correlated case, this may take $E[\#\ \textit{consecutive lost}]_{corr}$ announcements, therefore:

$$\begin{aligned}
E[\textit{re-establishment delay LE-S}]_{corr,joiner\neq leader} &= E[\lceil (t_a - \Delta)/T_A \rceil] \times T_A \\
&\quad + (E[\#\ \textit{consecutive lost}]_{corr} \times T_A) + \Delta - E[t_a] \\
&= \left(E[\lceil (t_a - \Delta)/T_A \rceil] + \frac{l}{1-l}\right) \times T_A + \Delta - E[t_a]
\end{aligned}$$

5.3.2.3 Uncorrelated Loss

The only difference between the uncorrelated and correlated loss cases is the term that indicates the number of rounds of announcements required to establish an agreed-upon leader.

Leader Departure. As can be seen from the result above, there is no loss term for this metric. Thus, all three scenarios (lossless, correlated loss and uncorrelated loss) have identical results:

$$\begin{aligned} E[\textit{re-establishment delay LE-S}]_{uc} &= T_L + 2\Delta - E[t_d] + E[t_s] \\ &= E[\textit{re-establishment delay LE-S}]_{l=0} \\ &= E[\textit{re-establishment delay LE-S}]_{corr} \end{aligned}$$

False Departure.

$$\begin{aligned} E[\textit{re-establishment delay LE-S}]_{uc,false} &= T_A \times E[\#\ \textit{consecutive lost}]_{uc} - T_L + T_A \\ &= T_A \times \left(1 + \sum_{i=1}^{N} (-1)^{(i+1)} \binom{N}{i} \frac{1}{(1-l^i)}\right) - T_L \end{aligned}$$

Late Joiners. Adjusting the term that indicates the number of rounds of announcements required to establish an agreed-upon leader, we find the corresponding metrics for uncorrelated loss:

$$\begin{aligned} E[\textit{re-establishment delay LE-S}]_{uc,joiner>leader} &= E[t_a] + E[t_s] + \\ &\quad (T_A \times E[\#\ \textit{consecutive lost}]_{uc}) + \Delta - E[t_a] \\ &= E[t_s] + \Delta + T_A \times \sum_{i=1}^{N} (-1)^{(i+1)} \binom{N}{i} \frac{1}{(1-l^i)} \\ &= E[\textit{leadership delay LE-S}]_{uc} \end{aligned}$$

$$\begin{aligned} E[\textit{re-establishment delay LE-S}]_{uc,joiner\neq leader} &= E[\lceil (t_a - \Delta)/T_A \rceil] \times T_A + \\ &\quad (E[\#\ \textit{consecutive lost}]_{uc} \times T_A) + \Delta - E[t_a] \\ &= \left(E[\lceil (t_a - \Delta)/T_A \rceil] + \sum_{i=1}^{N} (-1)^{(i+1)} \binom{N}{i} \frac{1}{(1-l^i)}\right) \\ &\quad \times T_A + \Delta - E[t_a] \end{aligned}$$

5.3.3 Number of Messages Generated

5.3.3.1 No Loss

Consider a collection of N processes, and a particular process q_k with the $(k+1)^{st}$ largest address. As shown in Figure 5.5, vector $\vec{Q}_{max}$ is an ordering of process addresses from largest to smallest. Thus, there are k processes with addresses larger than q_k. Given process q_k picks suppression time t, it will announce leadership by sending a message only if it does not receive a message from a process with a larger address. In other words, it becomes leader if all the k processes with larger addresses pick suppression times larger than $t - \Delta$. The likelihood of this event is $(1 - P(t - \Delta))^k$, where

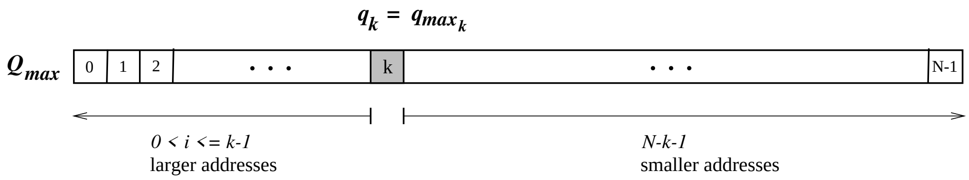

Figure 5.5: **Vector $\vec{Q}$ of Process Addresses Ordered from Largest to Smallest.**

$p(t)$ is the probability density function for the suppression timer and $P(t)$ the associated cumulative distribution function (as first appeared in Chapter 2). Therefore, the probability that process q_k sends a message is:

$$\begin{aligned} Pr[q_k \textit{ sends a message LE-S}]_{l=0} &= \int_0^{T_S} p(t)(1-P(t-\Delta))^k \, dt \\ &= P(\Delta) + \int_{\Delta}^{T_S} p(t)(1-P(t-\Delta))^k \, dt \end{aligned}$$

Therefore, the expected number of messages generated during the Suppression interval is given by:

$$\begin{aligned} E[\textit{number messages LE-S}]_{l=0} &= \sum_{k=0}^{N-1} Pr[q_k \textit{ sends a message LE-S}]_{l=0} \\ &= NP(\Delta) + \sum_{k=0}^{N-1} \int_{\Delta}^{T_S} p(t)(1-P(t-\Delta))^k \, dt \\ &= NP(\Delta) + \int_{\Delta}^{T_S} p(t) \sum_{k=0}^{N-1} (1-P(t-\Delta))^k \, dt \\ &= NP(\Delta) + \int_{\Delta}^{T_S} p(t) \frac{1-(1-P(t-\Delta))^N}{P(t-\Delta)} \, dt \end{aligned}$$

5.3.3.2 Correlated Loss

When there is loss present in the system, we need to consider the number of rounds until the leader announcement reaches the other processes, i.e., the number of leaders reduces to one. This means there will be some number of false leaders in each round that contribute to the total count. Below we calculate the number of messages generated in the initial Suppression interval.

Let t be the suppression time selected by q_k. In the correlated loss case, q_k will send a message if all the k processes with larger addresses pick suppression times larger than $t-\Delta$, or if their messages are lost. Note that when a message is lost by q_k, it is also lost by all other processes with addresses larger than it thereby eliminating the possibility that a message sent by q_i $(i < k)$ suppresses q_j where $i < j < k$. The likelihood of this event is $(1 - P(t-\Delta) + lP(t-\Delta))^k$. Therefore, the probability that process q_k sends a message is:

$$\begin{aligned} Pr[q_k \textit{ sends a message in } T_S \textit{ LE-S}]_{corr} &= \int_0^{T_S} p(t)(1-(1-l)P(t-\Delta))^k \, dt \\ &= P(\Delta) + \int_\Delta^{T_S} p(t)(1-(1-l)P(t-\Delta))^k \, dt \end{aligned}$$

Therefore, the expected number of messages generated during the Suppression interval is given by:

$$\begin{aligned} E[\textit{number messages in } T_S \textit{ LE-S}]_{corr} &= \sum_{k=0}^{N-1} Pr[q_k \textit{ sends a message LE-S}]_{corr} \\ &= NP(\Delta) + \sum_{k=0}^{N-1} \int_\Delta^{T_S} p(t)(1-(1-l)P(t-\Delta))^k \, dt \\ &= NP(\Delta) + \int_\Delta^{T_S} p(t) \sum_{k=0}^{N-1} (1-(1-l)P(t-\Delta))^k \, dt \\ &= \frac{NP(\Delta)}{1-l} + \int_\Delta^{T_S} p(t) \frac{1-(1-(1-l)P(t-\Delta))^N}{P(t-\Delta)} \, dt \end{aligned}$$

In later rounds, there may still exist multiple leaders, although we expect fewer than in the initial round (unless of course $l = 1$), and even fewer with each subsequent round. The processes behave similarly to the initial round in that they maintain their relative distances from each other in terms of when they issue announcements. Each process that did not receive a message from a process with a larger address continues to issue announcements (because it still believes it is the leader), until such time that it receives a message from process q_{N-1}.

The question becomes how many rounds occur before the algorithm stabilizes and how many messages are generated per round? We explore this more completely in the uncorrelated loss case below, as it is the case that generates the worst case performance, as we found with the Suppression algorithm (Section 3.3).

5.3.3.3 Uncorrelated Loss

To derive the number of messages when there is uncorrelated loss present, we use a similar approach to that used in the Suppression with loss chapter (See Chapter 3). Because of the dependencies between rounds of announcements, the problem is more tractable if we cast it as a recurrence relation. Because of the complexities that arise due to the interaction of message loss and transmission delay, we focus on a solution for the case when there is no transmission delay, $\Delta = 0$.

Let $Q(i, n)$ be the probability that i messages are sent by n processes $q_0 \dots q_{n-1}$, given a particular vector of times selected $\vec{T} = (t_0, t_1, \dots t_{n-1})$. While $\vec{Q}$ is fixed, $\vec{T}$ is random.

$Q(i, n)$ can be broken down into two disjoint parts: (a) Process $n-1$ sends one message, and processes $0 \leq j < n-1$ send $(i-1)$ messages; (b) Process $n-1$ does not sent a message, and processes $0 \leq j < n-1$ send i messages. In case (a), process q_{n-1} sends a message if for each process q_k, $k < n-1$, its message is either lost or is sent at a time after the time chosen by q_{n-1}, which occurs with probability $(l + (1-l)(1-P(t_{n-1})))^{i-1}$. In case (b), process $n-1$ does not send a message if it received one of the i messages. In other words the process did not lose all of the messages sent, each of which were either lost or not lost but arrived too late, which occurs with probability $1 - (l + (1-l)(1-P(t_{n-1})))^i$.

Therefore, we write:

$$\begin{aligned} Q(i,n) &= Pr[i \textit{ messages sent by } q_0, ..q_{n-1} \textit{ given } t_{n-1}] \\ &= \int_0^{T_S} Q(i-1, n-1) \times (l + (1-l)(1-P(t_{n-1})))^{i-1} + \\ &\quad Q(i, n-1) \times \left(1 - (l + (1-l)(1-P(t_{n-1})))^i\right) p(t)\, dt \end{aligned}$$

Now let us examine $Q(i,n,r)$, the probability that there are i messages sent by n processes in round r, where $r=1$ is the first round. A process *survives* round r if it sent a message in that round and does not receive a message with a larger id. If a process did not send a message in round r, then it will not send a message in round $r+1$, as it has already been suppressed by a process with a larger id. If a process sends a message in round r, but receives a message with a larger id in that round, then it does not send a message in round $r+1$.

Let $S(n,r)$ be the probability that process q_{n-1} reaches round r.

$$S(n,r) = S(n, r-1) \times Pr[q_{n-1} \textit{survives round } r-1]$$

For process q_{n-1} to survive round r, it must lose all messages sent by processes with larger addresses, $q_0, q_1, \ldots q_{n-2}$. However, the probability of those losses are dependent on the number of messages actually sent in that round.

$$\begin{aligned} Pr[q_{n-1} \textit{survives round } r-1] &= Pr[q_0 \ldots q_{n-2} \textit{ send } 1 \textit{ message and } q_{n-1} \textit{ loses } 1] + \\ &\quad Pr[q_0 \ldots q_{n-2} \textit{ send } 2 \textit{ messages and } q_{n-1} \textit{ loses } 2] + \\ &\quad \ldots \\ &\quad Pr[q_0 \ldots q_{n-2} \textit{ send } n-1 \textit{ messages and } q_{n-1} \textit{ loses } n-1] \\ &= Q(1, n-1, r-1) \times l + Q(2, n-1, r-1) \times l^2 + \ldots + \\ &\quad Q(n-1, n-1, r-1) \times l^{n-1} \\ &= \sum_{i=1}^{n-1} Q(i, n-1, r-1) \times l^i \end{aligned}$$

Note that it does not matter if a message sent by a process with a larger address is sent earlier or later than q_{n-1}'s scheduled announcement; either way, q_{n-1} must lose it to survive the round.

Substituting back into the earlier equation,

$$S(n,r) \quad = \quad S(n,r-1) \times \sum_{i=1}^{n-1} Q(i,n-1,r-1) \times l^i$$

Having derived $S(n,r)$, we now extend our earlier result for $Q(i,n)$ to consider rounds. The probability of i messages being sent by n processes in round r:

$$\begin{aligned} Q(i,n,r) &= Pr[i \text{ messages sent by } q_0, ..q_{n-1} \text{ in round } r \text{ given } t_{n-1}] \\ &= S(n,r-1) \times \\ &\quad \Big(\int_0^{T_S} Q(i-1,n-1,r-1) \times (l + (1-l)(1-P(t_{n-1})))^{i-q}) + \\ &\quad Q(i,n-1,r-1) \times \big(1 - (l + (1-l)(1-P(t_{n-1})))^i\big)\, p(t)\, dt \Big) \end{aligned}$$

We observe that each round is dependent on the previous round's results. When we recursively substitute $Q(i,n,r)$ back into $S(n,r)$, we find that $S(n,r)$ can be expressed entirely in terms of $Q(i,n,r)$:

$$S(n,r) \quad = \quad \prod_{k=1}^{r-1} \sum_{i=1}^{n-1} Q(i,n-1,k) \times l^i$$

Thus, $Q(i,n,r)$ itself can be expressed recursively:

$$\begin{aligned} Q(i,n,r) &= \prod_{k=1}^{r-2} \sum_{i=1}^{n-1} Q(i,n-1,k) \times l^i \times \\ &\quad \Big(\int_0^{T_S} Q(i-1,n-1,r-1) \times (l + (1-l)(1-P(t_{n-1})))^{i-q}) + \\ &\quad Q(i,n-1,r-1) \times \big(1 - (l + (1-l)(1-P(t_{n-1})))^i\big)\, p(t)\, dt \Big) \end{aligned}$$

Since the leader process q_0 always sends a message, $Q(0,i,r) = 0$. We can also compute $Q(i,i,r)$, because this is the likelihood that each processes survives round r, i.e., each processes loses all messages sent by all other processes, in every round up to and including the current round. Therefore,

$$\begin{aligned} Q(0,i,r) &= 0 \\ Q(i,i,r) &= (l^1 \cdot l^2 \cdot l^3 \cdots l^{i-1})^r \\ &= (l^{(i-1)i/2})^r \\ &= l^{r(i-1)i/2} \end{aligned}$$

Thus, the expected number of messages generated during leadership election in the uncorrelated case when $\Delta = 0$ is given by:

$$\begin{aligned}
&E[\textit{number messages generated LE-S}]_{uc,\Delta=0} \\
&= \sum_{r=1}^{\infty} \sum_{1 \leq i < N} i \times r \times Pr[\textit{i messages sent by } q_0, ..q_{n-1} \textit{ in round r given } t_{n-1}] \\
&= \sum_{r=1}^{\infty} \sum_{1 \leq i < N} i \times r \times Q(i, n, r)
\end{aligned}$$

5.3.4 Inconsistent State

There are three scenarios that lead to inconsistent state as discussed in the previous section on re-establishment delay (Section 5.3.2): leader departure, false departure, and late joiners.

- **Leader Departure.** When the process that has been elected as the leader leaves the system, leadership must be re-established. The time between the old leader departing and the new leader's announcement arriving at the other processes is considered time spent in an inconsistent state.

- **False Departure.** When a leader process is *thought* to depart because the listen timer T_L expires, the agreement process becomes unnecessarily perturbed. At that point, the local process that has detected leaderlessness takes back leadership and begins the LE algorithm anew. The time between when leadership reverts to the local process and subsequently transfers back to the true leader is considered time spent in an inconsistent state.

- **Late Joiners.** When a new process arrives into the system, it must become synchronized with the rest of the processes about which process is the leader. The time between the arrival of the new process and the time before which all processes receive the (possibly new) leader's announcement is considered time spent in an inconsistent state.

Any time the processes spend in disagreement about the leadership is considered inconsistent state. The $E[\textit{inconsisent state}]$ in the system is simply defined as the fraction of the time that the system is spent in an ambiguous state, trying to re-establish agreement on leadership. We sum over all such occurrences in the system, and then divide by the how long the group has been established,

$$\begin{aligned}
E[\textit{inconsistent state}] &= \frac{\sum E[\textit{re-establishment delay}]}{\textit{all of time}} \\
&= \Big(\sum E[\textit{re-establishment delay}]_{\textit{leader departure}} + \\
&\quad \sum E[\textit{re-establishment delay}]_{\textit{false departure}} + \\
&\quad \sum E[\textit{re-establishment delay}]_{\textit{late joiners}}\Big) / \textit{all of time}
\end{aligned}$$

We examine the problem in more detail by breaking down the problem into its components. If the interdeparture time from the system is T_d, then:

$$E[inconsistent\ state]_{leader\ departure} = \frac{\sum E[re\text{-}establishment\ delay]_{leader\ departure}}{T_d}$$

False departures only occur in our model (a) in the no loss case, when $T_L < T_A$, and (b) in the lossy case, on average when $T_L < T_A \times E[\#\ consecutive\ lost]$. Therefore, the fraction of the time the system is inconsistent is,

$$\begin{aligned} E[inconsistent\ state]_{l=0, false\ departure} &= 1 - T_L/T_A \\ E[inconsistent\ state]_{lossy, false\ departure} &= 1 - \frac{T_L}{T_A \times (E[\#\ consecutive\ lost] + 1)} \end{aligned}$$

If the interarrival time for late joiners into the system is T_a, then:

$$E[inconsistent\ state]_{late\ joiners} = \frac{\sum E[re\text{-}establishment\ delay]_{late\ joiners}}{T_a}$$

5.4 Analysis and Simulation

In the previous section, we derived a series of performance metrics to evaluate the Leader Election algorithm. In this section, we primarily focus on the metrics for leadership delay and the number of messages generated. We validate the analysis with simulation, comparing the behavior of LE with that of SUP and AL. We examine alternate leader selection criteria to the standard method of using the largest process identifier. We present an overview of general trends of the algorithm, contrasting the behavior between the different versions of the LE algorithm under different operating conditions.

In our simulations, the group size N remained unchanged during the algorithm, the timer intervals T_S, T_A and T_L were fixed, and the transmission delay Δ was uniform. Each simulation was run 1000 times when $N \leq 100$, and 100 times when $100 < N \leq 500$. In the Suppression phase, each node chose a delay time, t_i, based on the Unix `srandom()` function that had been seeded with the simulation start time. The simulations specifically explored the behavior of the Leader Election algorithm with increasing packet loss probabilities, ranging from 0 to 1 in increments of .1. Unless indicated, all of the graphs in this section display simulation results.

In Table 5.1, we summarize the metrics tracked in our simulations for the Leader Election algorithm. The parameter *tmax* represents the time until convergence is reached by all processes, *num_msgs* represents the number of messages generated in that time, and *rounds* indicates the number of rounds that transpired. The *avg_rounds* parameter tracks the average number of rounds until convergence, as each process may converge in a different round. Convergence is the point in time when all processes agree that they have elected as leader the process with the largest identifier.

Metric	*Description*
tmax	time until convergence
num_msgs	number messages sent until convergence
rounds	number rounds until convergence
avg_rounds	average number rounds until convergence

Table 5.1: **Convergence Metrics Simulated.**

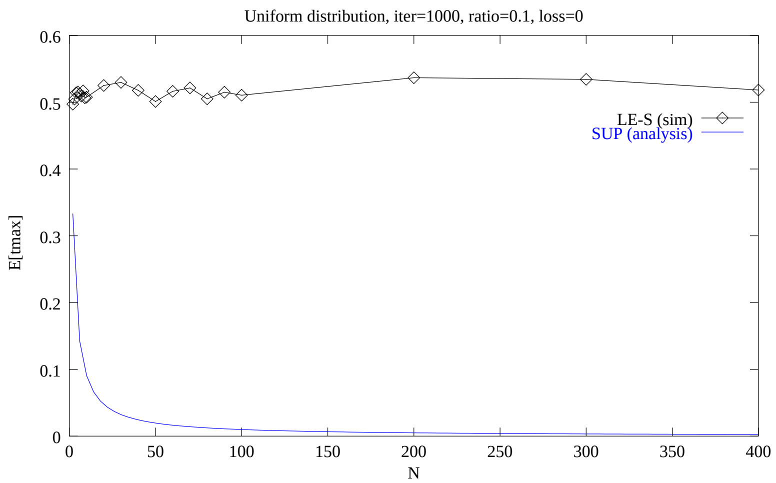

Figure 5.6: **LE vs. SUP:** $E[t_{max}]$ **vs.** N **(**$l = 0$**).**

5.4.1 Comparison with Suppression

In Figure 5.6, we display the delay until convergence, comparing the performance of LE-S simulations with that of SUP analysis in the lossless case (which matched SUP simulations that were shown in Chapter 2). Both the LE and SUP algorithms assume $T_S = T_A = 1$ and $\Delta = .1$, use a uniform timer distribution function for the selection of wake-up times, and examine the lossless case.

It is important to note that convergence has a slightly different meaning under the Suppression algorithm than it does under the Leader Election algorithm. Convergence for SUP measures the earliest halting time of the algorithm, the point in time when all processes agree that they are suppressed. We refer to this point in time as *tmax* because it is equivalent to the last time a message is sent in Suppression, t_{max}, as discussed in Chapter 3 and shown in Figure 3.1. Thus, the convergence delays in the graphs exclude Δ, i.e., they show the maximum time at which the last required message was sent not received. In addition, the Suppression algorithm never goes beyond one iteration because the receipt of any message, including ones own, is sufficient to cause "convergence." This has considerable impact on the time for convergence as compared to LE, as we show below. Nonetheless, convergence marks the completion time of the algorithm in both cases.

Convergence time for LE-S is equivalent to the metric for leadership delay $E[leadership\ delay\ LE-S]_{l=0} = E[t_s] + \Delta$. As can be seen from the plot of LE-S, when we use a uniform distribution for

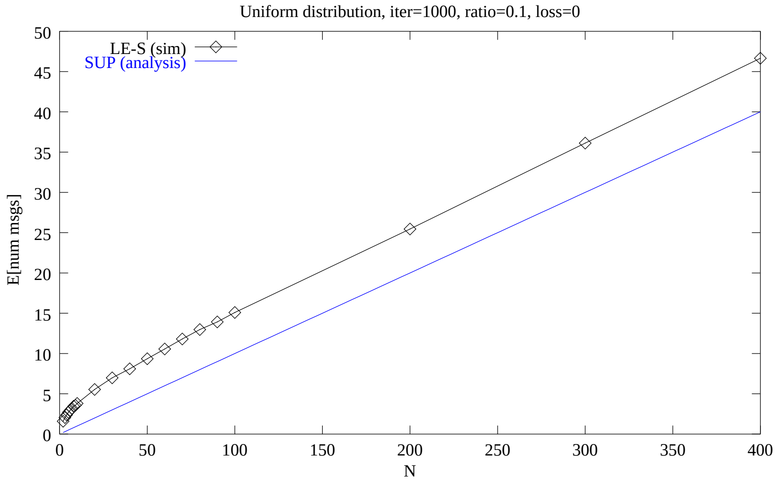

Figure 5.7: **LE vs. SUP:** $E[num\ msgs]$ **vs.** N **(**$l = 0$**).**

the selection of Suppression wake-up times, the delay until leadership is established is the mean of the distribution $E[t_s]$ (plus Δ if transmission delay were included in the graphs). On average the process with the largest process id will choose a wake-up time that is the mean of the distribution because every time is equally likely to be chosen by the leader. In this example, we have chosen a timer interval T_S bounded by 1 unit of time, and indeed the delay is .5 on average. The simulation matches the analysis.

The results for lossless LE-S are similar to the results for Suppression with uncorrelated loss. As the loss probability approaches 1, the average time at which nodes are suppressed ($avg\ t_{max}$, Section 3.8.1) approaches the mean of the distribution. This occurs because, when all N messages are dropped in the network, each node reverts to using its own wake-up time for the the time at which it becomes suppressed, or in this case the time at which it is considered converged.

Interestingly, the delay until convergence is independent of the number of participating processes. The explanation is that we are simply tracking the arrival of a message from the process with the largest identifier. With no loss, N has no impact on the receipt; only the arrival time of the process with the highest id ($E[t_s]$) and the transmission delay (Δ) of the message to the receiver processes will have any bearing on the result.

Figure 5.7 displays a comparison of LE-S and SUP in terms of the number of messages generated until convergence. As expected, Leader Election (LE-S) takes more time to converge than Suppression and generates more messages in doing so. This can be explained by the fact that suppression of a process in LE-S is no longer accomplished by the mere receipt of a message; the message must be from a process with a larger process identifier. This phenomenon is only exacerbated when the loss probability is non-zero. Therefore, convergence may take multiple iterations due to the loss of announcement messages from processes with large process ids, and in turn additional messages may

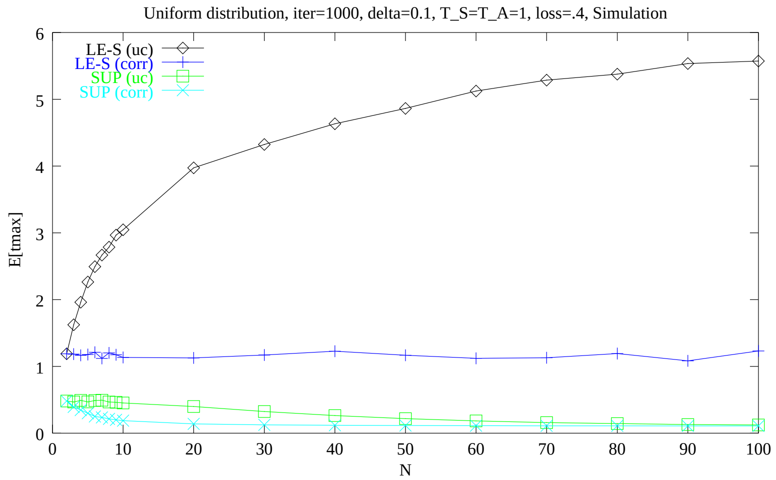

Figure 5.8: **LE vs. SUP: $E[t_{max}]$ vs. N ($l = .4$).**

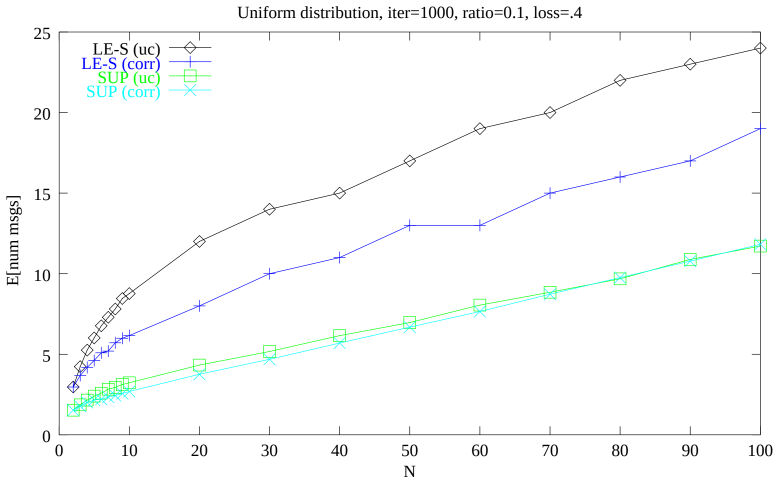

Figure 5.9: **LE vs. SUP: $E[num\ msgs]$ vs. N ($l = .4$).**

be generated.

Next we examine the behavior of SUP and LE under lossy conditions. We use the same parameters in our experiments as earlier, but now explore a loss probability of $l = .4$. Although this loss probability is high, it enables us to amplify the effects of loss and therefore to examine them more easily. In Figures 5.8 and 5.9, we see that loss takes a considerable toll on the performance of Leader Election, especially in the case of uncorrelated loss, and especially compared to Suppression. Increased loss of messages causes an increase in the number of iterations needed for convergence in LE-S, which in turn means the passage of more time and more messages before leadership is established.

In the case of correlated loss, the analysis for $E[leadership\ delay]_{corr}$ shows that convergence

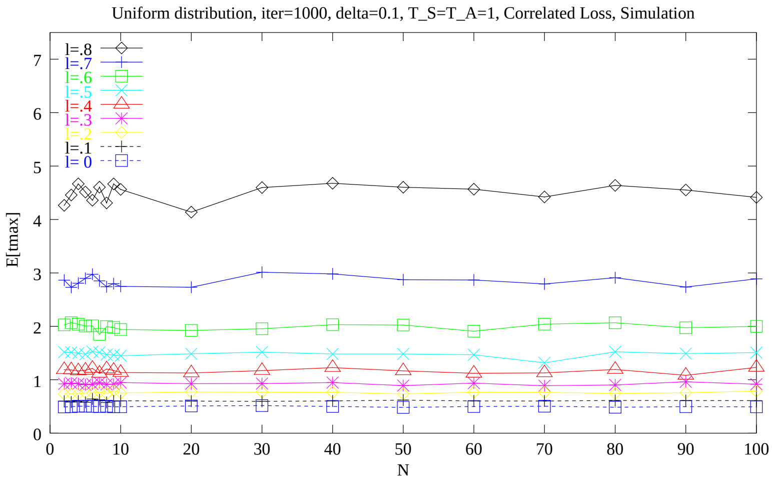

Figure 5.10: $E[t_{max}]$ **vs.** N**: Correlated Loss (Varying** l**).**

time is independent of group size N, as was the behavior in the lossless case (Figure 5.10). For example, given $T_A = 1$ and $E[t_s] = .5$, the plots reveal that the simulations match the analysis (excluding Δ, due to the definition of $tmax$). The results for $E[leadership\ delay]_{uc}$, the convergence delay for uncorrelated loss, however, are a function of group size, but in the limit (as N becomes large) these results are asymptotic as well.

The LE algorithm exhibits the same best and worst case performance as SUP. For both convergence time and number of messages, uncorrelated loss gives upper bounds for the expected behavior of the algorithm.

5.4.2 Comparison with Announce-Listen

A comparison of LE and AL shows that, for small N, LE behaves similarly in performance to AL for convergence time, in that the shape of the curves is similar (Figure 5.11). LE is like AL in that it effectively sends periodic announcement messages and converges when all processes receive the leader's announcement. In AL, convergence corresponds to the point in time when all processes are consistent with regard to the state they store for the leader. The difference between the operation of the two algorithms is the point at which announcements begin. For AL it is immediate, whereas for LE it is at $E[t_s]$. This difference can be seen in the convergence times in the graphs shown in Figure 5.11 (again, convergence time is displayed minus transmission delay Δ, thus showing when the convergence message was sent not received).

Initially the convergence times depicted as $E[tmax]$ along the y-axis show a disparity of $E[t_s]$ between AL and LE. However, note that there is a point at which LE becomes superior to AL. For Figure 5.11, where $l = .4$, that point occurs at $N = 200$ (though the degradation of the performance of AL begins at $N = 70$). This is an example of the need for an adjusted loss parameter, as discussed

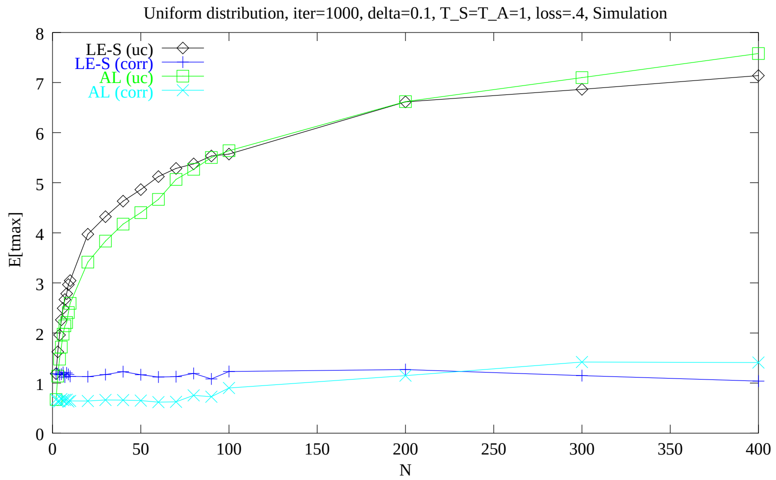

Figure 5.11: **LE vs. AL:** $E[t_{max}]$ **vs.** N **(**$l = .4$**).**

in Chapter 1. This is precisely why LE-S is more attractive than LE-A for multipoint algorithms with large N; it exhibits better scaling properties for large groups.

We do not compare the number of messages generated for LE and AL because they have quite different meanings. Under LE the number of messages generated is the number of processes that believe they are leaders accumulated over the number of rounds they hold this belief. For AL, all processes generate messages with every round.

5.4.3 Leader Selection Criteria

The analysis of the performance metric for leadership delay is quite straightforward. It is the average time it takes for the leader process to awaken in the Suppression interval, denoted $E[t_s]$, plus the time it takes for the leader's announcement to reach the other processes; $E[leadership\ delay]_{l=0} = E[t_s] + \Delta$.

Nonetheless from this simple formula, we can see that a leader selection policy based on largest process address, a traditional approach that appears in countless LE algorithms, will not perform as well as one that selects a leader based on the minimum wake-up time the process chooses in the Suppression interval. The problem boils down to the fact that the mapping between largest process id and Suppression wake-up time is purely random, which explains why, when the Suppression timer distribution function is uniform, on average the process with the largest process id will choose a wake-up time that is the mean of the distribution.

However, we know from our analysis of SUP that the leader selected for its minimal wake-up time leads to an $E[t_s]$ term that is equal to the minimum delay metric in SUP, and thus outperforms the more common approach to leader selection. In effect, this new approach creates the optimal mapping between leader selection criterion and timer wake-up function.

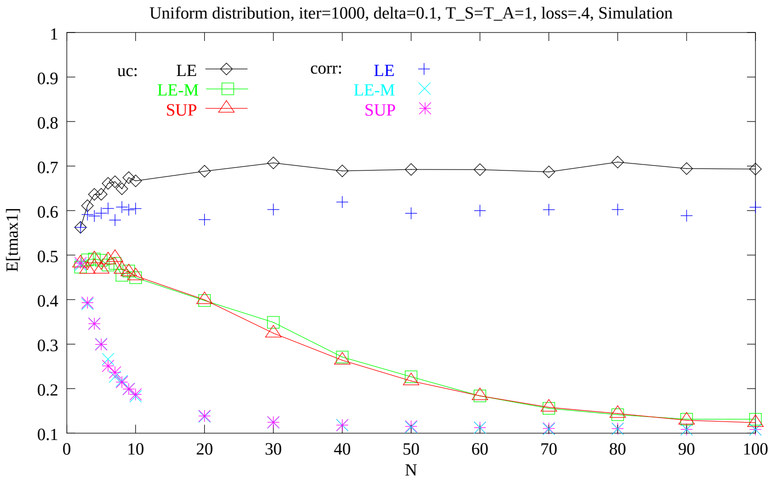

Figure 5.12: **First Round Comparison of LE-M with LE and SUP: $E[t_{max}]$ vs. N ($l = .4$).**

Therefore, for optimal convergence properties (minimal time to converge and fewest number of messages generated), $\vec{Q}$ and $\vec{T}_{min}$ would have to map to each other perfectly. In (Chapter 3), the vector $\vec{Q}$ was defined as an ordering of largest to smallest process identifiers, and the vector $\vec{T}$ was an ordering of wake-up times from earliest to latest. The process with the largest address would need to select the earliest wake-up time within the Suppression interval, the next largest address would need to select the next earliest time, and so forth: q_0 selects t_{min_0}, q_1 selects t_{min_1}, ...

However, here we suggest that minimal leadership delay occurs when $\vec{Q} = \vec{T}_{min}$. In other words, instead of using largest process id as the basis for leadership selection use the minimum wake-up time. Or alternatively, we can think of $\vec{Q} = \vec{T}_{min}$ as suggesting the need for a more direct mapping between **any** leader selection criterion (e.g., largest process id, smallest process load, greatest connectivity) and the Suppression timer wake-up function. For example, in the examples used throughout the chapter, make wake-up time a function of the process id. Processes with larger addresses probabilistically would acquire smaller wake-up values.

We compared the leadership delay for SUP and LE with a modified LE which used the minimum wake-up time as the leader selection criterion and which we refer to as LE-M. When an optimal mapping occurs, LE-M achieves its lower bounds and behaves identically to SUP, at least with regard to the first round of the algorithm. In Figure 5.12 we can see that the results for LE-M in the first round closely overlap with the results for SUP. Note that LE-M also achieves its lower bound when there is no message loss, and when LE, LE-M and SUP all complete in one round.

In Figure 5.13 we display the behavior of the three algorithms when there exists a loss probability of .4. The graphs clearly show that LE-M outperforms an LE algorithm using the standard leader policy. Because increased loss leads to multiple iterations of the announce-listen phase of the algorithm, SUP outperforms LE-M, but still serves as a lower bound for optimal performance

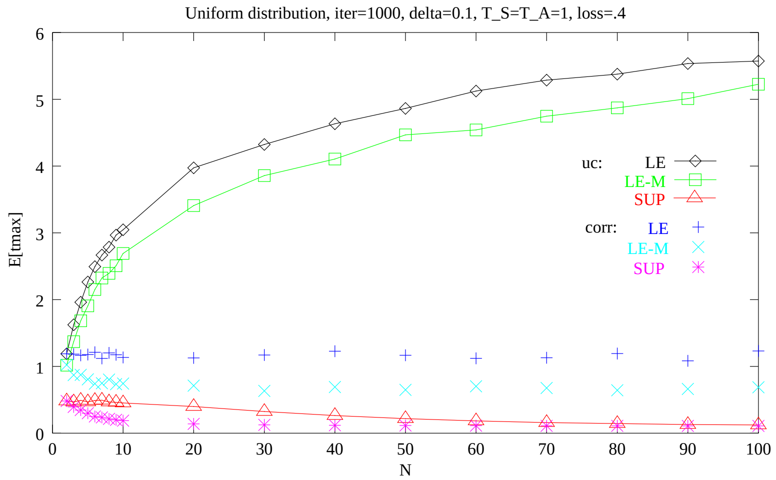

Figure 5.13: **LE-M Compared with LE and SUP:** $E[t_{max}]$ **vs.** N **(**$l = .4$**).**

overall.

The number of rounds raises an important point. We know from the recurrence relation in Section 5.3.3.3 that the number of messages generated in one round impacts the number of messages generated in the next round. Therefore, the fewer messages that are generated in the first round, the fewer messages overall.

We also know from the discussion of timer distribution functions in Suppression (Chapter 2) and from Nonnenmacher's studies [46] that exponential distribution functions lead to fewer messages under certain conditions. Therefore, for applications that are sensitive to messaging overhead, exponential distribution functions for T_S should be considered for the Suppression phase in the Leader Election algorithm.

5.4.4 General Trends

There are several general trends that appear in the metrics. For leadership delay, the performance of LE-S as compared to LE-A, not surprisingly, requires a simple adjustment of $E[t_s]$ units of time. When the leader departs from the system, the re-establishment delay is the same across the no loss, correlated loss and uncorrelated cases. When a late joiner arrives that will become the eventual leader, the re-establishment delay is the same as the leadership delay. If the late joiner is not destined to become the leader, we find that the re-establishment delay for the LE-S case is identical to that for LE-A. This phenomenon occurs because synchronization of a non-leader process with the rest of the processes is independent of the wake-up timer selected by the non-leader process. Re-synchronization is purely a function of how long it takes for the arriving process to hear from the existing leader process.

When moving from a lossless to lossy model, the metric for leadership delay is slightly modified to account for lost announcements, and the delay until all processes converge on a single leader is longer. How much longer is a function of the expected number of announcements beyond the first, given a particular loss level. This can be seen when we compare the simple metric for leadership delay, $E[leadership\ delay]_{l=0} = E[t_s] + \Delta$, with its lossy counterparts, which have additional terms.

When moving from a correlated to an uncorrelated loss model, a metric with a term that uses $E[\#\ consecutive\ lost]_{corr}$ is replaced by one using $E[\#\ consecutive\ lost]_{uc}$; this can be observed in a comparison of $E[leadership\ delay]_{corr}$ and $E[leadership\ delay]_{uc}$.

In the discussion on re-establishment delay, and the ensuing discussion on inconsistent state, we show analytically that these metrics are very much a function of process arrival and departure rates. However, they are also a function of other parameters, notably the timer values T_L and T_A, especially in the case of false departures. False detection of leaderlessness can be avoided on average, by setting $T_L \geq T_A$ in the lossless case, and by setting $T_L > T_A \times E[\#\ consecutive\ lost]$ in the lossy case. We found the same characterization of T_L in AL; the algorithm should be parameterized so that $T_L = kT_A$, where k is a small integer value that should be set according to the number of attempts required for an announcement to reach all participating processes.

We also can observe in the LE metrics and know from our analysis of SUP, that $T_A \geq T_S + \Delta$. If not, then a leader that selects a late wake-up time may not have time to make an announcement before other processes begin making regular announcements, reducing the effectiveness of the Suppression interval.

5.5 Related Work

In the seminal work of Fischer and Lynch [26], it is shown that consensus is impossible in an asynchronous system of processes when there exist unreliable processes. It is for this reason that we examine alternate methods for consensus building in a distributed system prone to failures.

In [56], Singh et al. study the election of qualified leaders. The decision is based on group voting protocols rather than process id numbers. This leads to better selection of leaders because an election can take performance considerations into account. The election schemes that they propose are important for making optimal choices, and one could apply them to the algorithms we have presented in this thesis. However, the actual technique they use for conducting the election is inappropriate in our context because it requires that each node submit a vote that is subsequently tallied and combined with other votes. Their work explores the idea of resiliency by making use of the group majority to reduce the impact of errant group members. We subscribe to a similar philosophy, achieving resiliency in our approach by removing any single-point of failure.

Cidon and Mokryn [16] address the problem of LE in a broadcast environment, taking the traditional distributed algorithm and placing it in a new context. Although quite similar in spirit to our work, we take the additional step of moving the analysis out of the LAN broadcast arena and generalizing the problem further still. By studying the algorithm in a WAN setting, our model diverges from theirs, in that we do not place constraints on communicating processes having to be neighbors, nor do we assume that the communication medium provides reliable or ordered delivery of messages. They assume a concurrent broadcast environment with no losses. Another difference is that they study the case where all processes issue acknowledgment messages to terminate message propagation and the algorithm itself. In contrast, we are free from any specific topology constraints, and only require that participating processes are members of the same group addresses, regardless of how geographically separated the processes may be. We also study the impact of a probabilistic loss model on the operation of LE in a broadcast environment.

Afek and Gafni [2] derive bounds for the performance of LE in both asynchronous and synchronous complete networks, where each process is directly connected to all other processes. Peleg [48] re-examine these results in the context of networks of diameter at most D. This bears resemblance to our approach, in that we assume a multicast group abstraction is provided by a lower layer in the network and means that all nodes are reachable from all other nodes within some maximum transmission delay, in our case Δ. We also provide loss analysis as part of our analysis, and combine Suppression with the initial propagation of leadership messages, in order to reduce messaging overhead.

Fetzer and Cristian [25] study the highly available local leader election problem, which is targetted at fail-aware systems that are prone to group partitions. Like the LE algorithms we study, their algorithm makes use of heartbeat messages for purposes of robustness. However, they design their algorithm for timed asynchronous systems, where delays associated with messages are assumed to be finite but unbounded.

5.6 Summary of Results

In this chapter, we explored two forms of the Leader Election algorithm. We presented the basic approach that establishes leadership immediately by employing Announce-Listen as a first step, and thus was referred to as LE-A. We discussed a refined approach that delays leadership slightly by inserting a phase of Suppression before beginning AL, and as a result has better scaling properties; we named it LE-S. We focused on the analysis for LE-S because LE-A can be derived from it by setting the initial Suppression interval $T_S = 0$. Although this technique is used by many protocols, LE-S had not been analyzed formally.

To evaluate the performance of LE, we explored several metrics: Leadership Delay, the delay until leadership is established at the outset of the algorithm; Re-establishment Delay, the delay until leadership is re-established when leadership is perturbed, as might happen when the leader departs, processes falsely detect leaderlessness, or when new processes arrive into the system; The Number of Messages Generated, and; Inconsistent State, the fraction of the time that the algorithm leads to ambiguity about leadership.

We discussed these metrics under several loss scenarios: no loss, correlated loss and uncorrelated loss. Where appropriate, we modelled the performance of these metrics when all processes arrived into the system simultaneously, during steady state, and when there were perturbations in the leadership.

We compared the results of these metrics with the bounds we had already established for Suppression (Chapters 2 and 3) and Announce-Listen (Chapters 4) in earlier chapters. We found that the impact on SUP of correlated and uncorrelated loss carries over to LE; the loss model leading to the best or worst case for SUP are the best or worst case for LE-S. We also discovered that leadership delay is independent of the group size N, at least for the lossless and correlated loss cases, and in the limit exhibits asymptotic behavior for the uncorrelated case.

Compared to AL, LE-S takes longer to converge for small N, and the convergence is shifted by the mean delay of the Suppression interval $E[t_s]$. As group size grows, however, the performance of LE-S is superior to AL (and LE-A for that matter) due to simultaneous messages and results in a larger effective loss probability for AL.

Not surprisingly, Leader Election generates more messages than the Suppression algorithm due to the differences in the way that participating processes become suppressed. The need to receive a message with a higher process id takes significantly longer than the time it takes for the mere receipt of another message. In addition, Suppression completes in one round, whereas LE continues for multiple rounds, meaning LE takes longer to stabilize. The increased delay incurred by Leader Election translates into increased message generation.

We showed that the traditional approach to leader selection that is based on greatest process id performs sub-optimally for protocols that incorporate the LE-S algorithm. This is because the mapping of process id to wake-up timer value is random. We discovered that a leader selection criterion based on minimal wake-up time was an improvement over the standard method. Furthermore, we suggest that making wake-up time a function of the property governing leadership would be an improvement to standard practice.

5.7 Future Work

Reducing Re-establishment Delay. As the listen timer is set to T_L, it may take a considerable period of time for a participant in the LE algorithm to detect that the leader has departed. For multicast applications that require timely detection of this event, applications have resorted to the use of explicit `Bye` messages, which are announcements containing departure state information rather than arrival state information. The idea behind this action is to decrease the delay before process leadership departure is detected.

We would like to study the impact of such an announcement on LE. If we know the level of loss in the system, then we can calculate how many `Bye` messages to send on departure. If the departing process is a non-leader, then announcement of departure is unnecessary.[3] If the departing process is the leader, then a number of messages ($E[announcement\ i\ establishes\ leadership]$ messages) must be generated to reach all members. If leaders depart frequently, this could have a substantial impact on the numbers of messages generated by the algorithm.

Other Definitions of Leadership. In the early examples throughout this Chapter, we assumed that no mapping existed from $\vec{Q}$ and $\vec{T}_{min}$, the vector of process addresses ordered from greatest to smallest and the vector of selected wake-up times ordered from earliest to latest, respectively. Later we showed that the performance metric for leadership delay can be improved when we set $\vec{Q} = \vec{T}_{min}$. In other words, LE exhibited optimal performance when we replaced the conventional definition of a leader (the process with the greatest identifier) with one based on the minimal Suppression wake-time time.

The idea of an improved mapping follows naturally from what Singh et al. [56] propose in their work. They suggest that instead of using the highest process id to elect the leader (which randomly chooses a leader), use a value that reflects a property that the application is trying to maximize (or minimize). There are several alternatives that make sense for the types of applications that are supported by our loosely-based communication model. For data consistency, a process could announce an estimate of the level of loss it has observed (e.g., a sum of the losses experienced over a given time interval, across all group members). For optimal placement of a leader within the group, use the sum of the delays observed between nodes, selecting the leader with the lowest value.

However, these alternate definitions bring us back to the issue of providing a good mapping between the property to optimize and the timer selection function for LE-S. In the future, we would like to investigate these and other definitions that would lead to "good" leaders [56]. In particular, we want to understand the effect of linking new definitions for leadership with the wake-up timer function, in effect combining the performance needs of the application with the inherent structure of the LE-S algorithm.

Relatedly, what is the impact of changing the algorithm to support multiple agreed-upon leaders? It is common practice that unicast versions of DNS require two copies of each registry to exist for reliability. With the arrival of multicast extensions to DNS [58] [21], it may fall within the province of LE-S to supply multiple leaders for registry sharing. In addition, cluster computing, client-server, and load-balancing applications also may require multiple leaders for redundancy and fault tolerance.

[3]For protocols, such as RTP[54][51] that actually track or approximate the level of group membership, `Bye` messages are useful from any group member, leader or not.

Composition and Parameterization of Existing Algorithms. We would like to examine other instances, besides LE, where SUP is used in combination with AL. This is part of the larger goal to understand if when multicast components are used in composition, whether or not their metrics can be used compositionally. We have found here that SUP and AL provide good lower bounds for LE-A and LE-S, but there is enough dissimilarity that the metrics do not compose.

Would there be utility in creating an LE algorithm based on SUP followed by AL (SUP+AL), without modification to SUP? That is, elect a leader in LE with the simple receipt of a message, as in SUP itself? This algorithm would not guarantee a single leader, as multiple nodes may elect themselves. Nor would it guarantee the presence of any leader, as the algorithm has the potential to deadlock, e.g., due to message loss or asymmetric delays, process A hears from B, who hears from C, who hears from A. Each defers to a different process to send leadership announcements, and none actively becomes the leader.

However, SUP+AL might be useful in other contexts, not only to avoid message implosion at initialization (the standard reason to precede AL by SUP), but also to limit the number of announcements sent by AL. For example, the algorithm could be used only to allow a random number of announcers to announce in each AL interval. The decision to announce or not would be based on the time a process had selected inside the Suppression interval; basically, the suppression values would be used to randomize the set of announcers allowed to send in each round. This technique could be used in lieu of adapting the announce timer T_A to keep the bandwidth usage below a particular threshold. This type of modification brings the SUP+AL algorithm closer to the individual components, and hence closer to the metrics for SUP and AL.

There are existing protocols that use SUP+AL, but SUP is used here only to spread out the announcements in time, rather than suppress most or even some of them [8]. We know from the analysis of SUP in Chapters 2 and 3, when the earliest and latest messages will be sent on average: $tmin = tmin_0$, and $tmax = tmin_{n-1}$ respectively. The latter is not normally the case, but is here because all messages are generated, thus the last message sent selected the largest t_{min}. Because SUP is merely used to separate the announcements in time, we can trivially gauge the numbers of messages required for convergence: N.

Extensions to LE. There are a few other forms of Leader Election that exist and that have metrics no doubt related to the variations of LE algorithms we examined in the thesis. They introduce additional opportunities to study the relationship between timer intervals. We would like to determine when these variations are improvements upon LE-A or LE-S.

1. **Query.** An explicit request or `Query` message is sent at the outset of the algorithm to find the leader, followed by an announcement if no leaders respond. If the Announce timer duration, T_A, is long compared to the response time needed by the application, a Query can trigger a

reply sooner than the next announcement. Note that a `Query` functions like an announcement message, in that it shares leadership information among processes. However, it generates a reply from the leader(s).

2. **Suppress, then Query.** Suppression precedes the Query and is used to reduce the number of messages generated should all processes begin at once, or become synchronized. If none are received with a greater address by the selected wake-up time, a process queries to find the leader, announcing leadership if no leaders respond. Note that suppression is also used when responding to a Query. This is an improvement over the Query technique in that it generates fewer messages. However, it introduces greater delay until leadership is established.

Chapter 6

Conclusion

With the exponential growth of the Internet, there is a critical need to design efficient, scalable and robust protocols to support the network infrastructure. In this thesis, we examined a new class of protocols that accomplish these goals by using multipoint, announcement-style communication.

We observed that many multipoint protocols rely on a small set of very powerful techniques to accomplish scalability. We studied a number of the scalability techniques that repeatedly appear in both proposed and deployed Internet protocols: Suppression, Announce-Listen and Leader Election. We primarily examined scenarios where group members arrived into the system simultaneously and where steady state was maintained. We also considered the effect of process arrivals and departures on these techniques.

We analyzed and simulated each algorithm using a probabilistic failure model. We derived performance metrics to evaluate each technique, focusing on metrics for delay, bandwidth usage, and memory overhead, where relevant. We also determined metrics for the level of consistency that can be achieved by these algorithms given their loosely-coupled messaging model and for the speed with which convergence can occur. The model exposed several key parameters that allowed us to study parts of the operating space that had not been fully investigated before. In particular, we presented a more sophisticated loss model, examining the impact of both correlated and uncorrelated loss on each algorithm, and allowing multiple versus single losses to occur in the system.

We studied the Suppression algorithm within both lossless and lossy networks. The first allowed us to scrutinize the importance of the timer distribution function on the tradeoff between metrics for response time and messaging overhead. We found that the benefits of one distribution over the other is very context dependent, which conflicts with the the standard practice of employing the uniform distribution by default. We also discovered that highly-tuned distribution functions ("designer functions") can be created that optimize the performance of a given Suppression metric, provided that certain network parameters can be bounded. In addition, examining Suppression in a lossless environment provided an opportunity to evaluate the algorithm in simplified terms, separate from other algorithms (with which it is often combined) and separate from any particular operating context (such as reliable multicast).

We then re-examined the Suppression algorithm under lossy conditions, refining the known metrics to more accurately reflect their performance in a lossy network and proposing several new metrics, including the completion time of the algorithm, as well as the expected number of required versus extra messages.

For Announce-Listen, we proposed an alternate model to existing models. The model not only allowed closer examination of parameters for caching and transmission delay, but also placed AL within a multipoint context where we could study the impact of group size more directly. We focused on the derivation of a metric for consistency, which was then used to frame the discussion of other metrics. Most importantly, we validated our model through simulation.

Unlike many other approaches to Leader Election, we did not tie the LE algorithm to any particular network topology. Instead we characterized its performance in terms of the bounds set by its transmission delay. In addition, we contrasted a more traditional version of the algorithm with one that employs a round of Suppression at the beginning of the algorithm. We found that Suppression offers superior scalability properties for large groups in lossy networks. We showed that a leader selection criterion based on greatest process identifier is limited, and that a leader selection policy based on the Suppression wake-up time is superior.

We also explored a theory of composition, that the metrics for SUP and AL were suitable metrics for an LE algorithm built from them. We discovered that they bound the performance of LE under certain ideal conditions, but were more aptly described as indicators of performance trends. The divergence between the performance of SUP and AL as separable components and the performance when combined into LE was due to changes in the functionality of SUP when it was used as a building block. An open issue is to explore the results when there exists a direct correspondence between components and composed algorithm.

Much of our analysis was validated through extensive simulation. Using our analysis, designers will be able to parameterize, as well as to predict the performance of these algorithms. Hence, designers will be better informed about how to optimize the metrics of choice. More broadly, through the analysis of these algorithms we established a methodology for examining other scalability techniques.

In the future, we would like to revise our network assumptions. We would like to examine different delay models, studying the variance in transmission delays. We would like to investigate other timer distribution functions in the Suppression algorithm, especially for differentiation between classes of group members. A future plan is to create a parameterized version of the basic Suppression technique; the decision to suppress would be based on network loss conditions and on the number of messages a process received while waiting for the Suppression timer to expire. In the future, we would like to model and to analyze the consistency of alternate versions of Announce-Listen, in which certain processes only listen sporadically to the group address or rely on proxies or hierarchies of proxies to distribute announcements on their behalf. For Leader Election, an open issue is to compare the traditional usage of process id with other election schemes, which are geared toward optimizing certain system metrics, such as response time, consistency, load, or topological placement. Finally, we would also like to identify and evaluate other scalability techniques upon which multicast algorithms are built.

Appendix A

Announce-Listen: Inconsistency and Departures

Tables A.1 and A.2 are refinements to Tables 4.2 and 4.3, in that they consider the impact of process departures on the state of the system. These tables show the likelihood for each type of error that arises in the registry of a listener process: inconsistent state, false negatives and false positives. These probabilities are associated with states that appear in the listener's transition graph as shown in Figure 4.5.

Table A.1 examines inconsistencies arising when the current time, t, occurs after Δ, meaning $t \geq \Delta + \lfloor \frac{t}{T} \rfloor \cdot T$. Table A.2 accounts for inconsistencies when t occurs within Δ, meaning $t < \Delta + \lfloor \frac{t}{T} \rfloor \cdot T$.

The notation is summarized in Table 4.1. The abbreviation $n.a.$ is not applicable, meaning the state is not possible. Recall that $\overline{D}(t) = 1 - D(t)$ is defined as the probability that the announcer is alive at time t.

Table A.1 examines inconsistencies arising when the current time $t < \Delta + \lfloor \frac{t}{T} \rfloor \cdot T$, whereas Table A.2 accounts for inconsistencies when time $t \geq \Delta + \lfloor \frac{t}{T} \rfloor \cdot T$.

A.1 Arrivals

If we look at each registry entry and pretend that time begins at 0, the tables are accurate (Tables 4.2 and 4.3, and tables A.2 and A.1). When we look at global registry consistency, each entry may have a different start time. Thus, let us examine the impact of the arrival distribution on $Pr[Err(t)]$, the probability of error at time t.

Let $A(u)$ be the distribution for process arrivals into the Announce-Listen algorithm. If a process arrives into the system at time u, then for that process the new probability of error at time t is shifted in time becoming $Pr[E(t-u)]$.

Consider an example in the discrete realm. Process i $(0 \leq i \leq N)$ arrives at time i (relative to

Listener State		*Inconsistent State*		*False*	*False*
		ε	$Pr[Err = \varepsilon]$	$-$	$+$
a_n	*Alive*	0	$\overline{D}(t)(1-p)$	*n.a.*	$(1-p)(D(t) - D(nT))$
a_{n-1}	*Not Sure*	1	$\overline{D}(t)(1-p)p$	*n.a.*	$(1-p)[(D(nT) - D((n-1)T)) +$ $(1-(D(nT) - D((n-1)T))) \times$ $p(D(t) - D(nT))]$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_{n-k+1}	*Last Try*	$k-1$	$\overline{D}(t)(1-p)p^{k-1}$	*n.a.*	$(1-p)[D((n-k+2)T) - D((n-k+1)T)) +$ $(1 - D((n-k+2)T) - D((n-k+1)T))) \times$ $p(D((n-k+3)T) - D((n-k+2)T)) +$ $\vdots$ $D(nT) - D((n-1)T) +$ $(1-(D(nT) - D((n-1)T))) \times$ $p^{k-1}(D(t) - D(nT))]$
a_{n-k}	*Departed*	0	X_t	$\overline{D}(t)p^k$	*n.a.*

Table A.1: **Inconsistency: After Δ.**

Listener State		*Inconsistent State*		*False*	*False*
		ε	$Pr[Err = \varepsilon]$	$-$	$+$
a_n	*Alive*	0	0	*n.a.*	*n.a.*
a_{n-1}	*Not Sure*	1	$\overline{D}(t)(1-p)$	*n.a.*	$(1-p)(D(t) - D((n-1)T))$
a_{n-2}	*Less Sure*	2	$\overline{D}(t)(1-p)p$	*n.a.*	$(1-p)[(D((n-1)T) - D((n-2)T)) +$ $(1-(D((n-1)T) - D((n-2)T))) \times$ $p(D(t) - D((n-1)T))]$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_{n-k+1}	*Last Try*	$k-1$	$\overline{D}(t)(1-p)p^{k-2}$	*n.a.*	$(1-p)[D((n-k+2)T) - D((n-k+1)T)) +$ $(1-(D((n-k+2)T) - D((n-k+1)T))) \times$ $p(D((n-k+3)T) - D((n-k+2)T)) +$ $\vdots$ $D((n-1)t) - D((n-2)T) +$ $(1-(D((n-1)t) - D((n-2)T))) \times$ $p^{k-2}(D(t) - D((n-1)T))]$
a_{n-k}	*Departed*	0	Y_t	$\overline{D}(t)p^{k-1}$	*n.a.*

Table A.2: **Inconsistency: Within Δ.**

time 0 when the algorithm begins) and each stays through time A, the last arrival time.

$$\begin{aligned} Average\ Pr[Err(t)] &= Pr[Err(t)] \cdot u_0 + Pr[Err(t-1)] \cdot u_1 + \\ &\qquad Pr[Err(t-2)] \cdot u_2 + \ldots + Pr[Err(t-N-1)] \cdot u_N \end{aligned}$$

In the continuous domain, the average probability of error at time t becomes the integral

$$Average\ Pr[Err(t)] = \int_0^A A(u) \cdot Pr[Err(t-u)] du$$

Appendix B

Leader Election: Rounds Needed with Uncorrelated Loss

For the Leader Election algorithm in Chapter 5, the difference between the delay metrics for the correlated versus uncorrelated case result from the difference in the expected number of rounds before everyone gets a message. In the correlated case, this is $1/(1-l)$. In this appendix, we calculate the uncorrelated case.

Since the losses are independent, we can think of each loss as a coin-toss experiment. Let $l = Pr[heads]$ and $(1-l) = Pr[tails]$. A process wants to toss the coin until it comes up tails, i.e., the process receives the message. Each process is performing an independent coin toss experiment and there are N processes in total.[1] The question is: what is the expected value of the maximum number of tosses before they all get tails?

Consider one process. It takes i tosses before a process gets a tails with probability $l^{(i-1)} * (1-l)$, just as in the correlated case (Section 5.3.1.2). So, the probability that it takes less than or equal to i tosses to get a tails,

$$\begin{aligned} Pr[\leq i \text{ tosses to get a tails}] &= (1-l) + l*(1-l) + l^2*(1-l) + ... + l^{(i-1)}*(1-l) \\ &= (1-l)(1+l+l^2+...+l^{(i-1)}) \\ &= 1-l^i \end{aligned}$$

What is the probability that given N processes the **maximum** number of tosses is k? For the maximum to be equal to k, at least one must be k, and all the others must be less than or equal to k. However it is more complicated, because any process could be the maximum and we must be careful not to double count because others being less than or equal to k does not prevent them from being equal to k.

Because each process behaves independently, we know that the probability for all processes to

[1] Actually there is one less than the number of processes in LE, which we call N here for simplicity.

have received a message in less than or equal to k tosses is

$$\begin{aligned} Pr[all \leq k] &= Pr[\leq k \text{ tosses to get a tails}]^N \\ &= (1 - l^k)^N \end{aligned}$$

Therefore, the probability that the maximum number of tosses is exactly equal to k is

$$\begin{aligned} Pr[max = k] &= Pr[all \leq k] - Pr[all \leq (k-1)] \\ &= (1 - l^k)^N - (1 - l^{(k-1)})^N \end{aligned}$$

which represents the probability that all processes have received a message in less than or equal to k attempts minus the probability that all processes received the message in less than or equal to $k-1$ attempts. In essence, we subtract out the overlap between these two probabilities and are left with the probability that exactly k attempts are required. Thus, the expected value of the maximum number of tosses, given N processes, is

$$\begin{aligned} E[max] &= \sum_{k=1}^{\infty} k \times Pr[max = k] \\ &= \sum_{k=1}^{\infty} k \times ((1 - l^k)^N - (1 - l^{(k-1)})^N) \\ &= \sum_{k=1}^{\infty} k \times \sum_{i=0}^{N} (-l^k)^i \binom{N}{i} - (-l^{k-1})^i \binom{N}{i} \\ &= \sum_{i=0}^{N} \binom{N}{i} \sum_{k=1}^{\infty} k \times ((-l^k)^i - (-l^{k-1})^i) \\ &= \sum_{i=0}^{N} \binom{N}{i} \sum_{k=1}^{\infty} k \times (-1)^i l^{(k-1)^i} (l^i - 1) \\ &= \sum_{i=0}^{N} (-1)^i \binom{N}{i} (l^i - 1) \sum_{k=1}^{\infty} k \times l^{(k-1)^i} \\ &= \sum_{i=0}^{N} (-1)^i \binom{N}{i} (l^i - 1) \sum_{k=1}^{\infty} k \times (l^i)^{k-1} \\ &= \sum_{i=0}^{N} (-1)^i \binom{N}{i} (l^i - 1) \frac{1}{(1 - l^i)^2} \\ &= \sum_{i=0}^{N} (-1)^i \binom{N}{i} \frac{-1}{(1 - l^i)} \\ &= \sum_{i=0}^{N} (-1)^{i+1} \binom{N}{i} \frac{1}{(1 - l^i)} \end{aligned}$$

The result $E[max]$ can be recast as the expected number of announcements that must be made by the leader process before leadership is established.

Bibliography

[1] Abramson, N., "The ALOHA System – Another Alternative for Computer Communications," Proceedings Fall Joint Computer Conference, AFIPS Press, pp. 281-285 (1970).

[2] Afek, Y., Gafni, E., "Time and Message Bounds for Election in Synchronous and Asynchronous Complete Networks," SIAM Journal on Computing, Vol. 20, No. 2, pp. 376-394 (1991).

[3] Almeroth, K.C., "A Long-Term Analysis of Growth and Usage Patterns in the Multicast Backbone (MBone)," IEEE INFOCOM '00, Tel Aviv, Israel (Mar 2000).

[4] Almeroth, K., Ammar, M., "Multicast Group Behavior in the Internet's Multicast Backbone (MBone)," IEEE Communications (June 1997).

[5] Alsberg, P.A., Day, J.D., "A Principle for Resilient Sharing of Distributed Resources," IEEE Proceedings 2nd International Conference on Software Engineering, pp. 562-70 (1976).

[6] Bajaj, S., Breslau, L., Estrin, D., Fall, K., Floyd, S., Haldar, P., Handley, M., Helmy, A., Heidemann, J., Huang, P., Kumar, S., McCanne, S., Rejaie, R., Sharma, P., Varadhan, K., Xu, Y., Yu, H., and Zappala, D., "Improving Simulation for Network Research," Technical Report 99-702, University of Southern California (Mar 1999).

[7] Baskett, F., Chandy, M., Muntz, R., Palacios, F. "Open, Closed, and Mixed Networks of Queues with Different Classes of Customers," Journal of the ACM, Vol. 22, No. 2, pp. 248-260 (1975).

[8] Berger, L., Gan, D.-H., Swallow, G., Pan, P., Tommasi, F., Molendini, S., "RSVP Refresh Overhead Reduction Extensions," Internet Draft (June 2000).

[9] Bhatti, N.T., Schlichting, R.D., "A System for Constructing Configurable High-Level Protocols," ACM SIGCOMM Computer Communication Review, Vol. 25, No. 4, pp. 138-150 (Oct 1995).

[10] Birman, K.P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., Minsky, Y., "Bimodal Multicast," ACM Transactions on Computer Systems, Vol. 17, No. 2, pp. 41-88 (May 1999).

[11] Birman, K.P., van Renesse, R., "Reliable Distributed Computing with the Isis Toolkit," IEEE Computer Society Press, Los Alamitos, CA (1994).

[12] Birman, K.P., Joseph, T.A., Raeuchle, T., Abbadi, A.E., "Implementing Fault-tolerant Distributed Objects," IEEE Transactions on Software Engineering, Vol. SE-11, No. 6, pp. 502-8, (June 1985).

[13] Cain, B., Deering, S., Kouvelas, I., Thyagarajan, A., "Internet Group Management Protocol, Version 3," Internet Draft (June 2000).

[14] Chandy, K.M., Schooler, E.M., "Designing Directories in Distributed Systems: A Systematic Framework," Proceedings of the Workshop on Multimedia and Collaborative Environments, High Performance Distributed Computing Conference, Syracuse, NY (Aug 1996); also available as technical report CS-TR-96-19, Department of Computer Science, California Institute of Technology, Pasadena, CA (Aug 1996).

[15] Chandy, K.M., Misra, J., "How Processes Learn," Distributed Computing, Springer-Verlag, Vol. 1, No. 1, pp. 40-52 (1986).

[16] Cidon, I., Mokryn O., "Propagation and Leader Election in a Multihop Broadcast Environment," DISC'98, Distributed Computing, 12th International Symposium, pp. 104-118, Andros, Greece, Sept 24-26 (1998).

[17] Deering, S., Estrin, D., Farinacci, D., Handley, M., Helmy, A., Jacobson, V., Liu, C., Sharma, P., Thaler, D., Wei, L., "Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification," RFC 2362 (June 1998).

[18] Deering, S., Estrin, D., Farinacci, E., Jacobson, V., Liu, C., and Wei, L., "The PIM architecture for wide-area multicast routing," ACM/IEEE Transactions on Networking, Vol. 4., No. 2, pp. 153-162 (Apr 1996).

[19] Deering, S.E., "Multicast Routing in a Datagram Network," PhD Thesis, Department of Electrical Engineering, Stanford University, Stanford, CA (Dec 1991).

[20] Deering, S., "Host Extensions for IP Multicast," RFC 1054 (May 1988).

[21] Esibov, L., Aboba, B., Thaler, D., "Multicast DNS," Internet Draft, work in progress (July 2000).

[22] Estrin, D., Farinacci, D., Jacobson, V., Liu, C., Wei, L., Sharma, P., Helmy, A., "Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol Specification," Internet Draft, work in progress (Aug 1997).

[23] Farinacci, D., Kouvelas, I., Windisch, K., "State Refresh in PIM-DM," Internet Draft (Oct 1999).

[24] Fenner, W., “Internet Group Management Protocol, Version 2,” RFC 2336 (Nov 1997).

[25] Fetzer, C., Cristian, F., “A Highly Available Local Leader Election Service,” IEEE Transactions on Software Engineering, Vol. 25, No. 5, pp. 603-618 (Sept 1999).

[26] Fischer, M.J., Lynch, N.A., Paterson, M.S., “Impossibility of Distributed Consensus with One Faulty Process,” Journal of the ACM, Vol. 32, No. 2, pp. 374-382 (Apr 1985).

[27] Floyd, S., Jacobson, V., Liu, C., McCanne, S., Zhang, L., “A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing,” IEEE/ACM Transactions on Networking, Vol. 5, No. 6, pp. 784-803 (Dec 1997).

[28] Floyd, S., Jacobson, V., “The Synchronization of Periodic Routing Messages,” *ACM SIGCOMM Computer Communication Review*, Vol. 23, No. 4, pp. 33-44 (Oct 1993).

[29] Friedman, T., Towsley, D., “Multicast Session Membership Size Estimation,” Proceedings IEEE Infocom’99, New York, NY, USA (Mar 1999).

[30] Friedman, T., Towsley, D., “Multicast Session Membership Size Estimation,” University of Massachusetts Amherst Computer Science Department, Technical Report TR-98-32 (Aug 1998).

[31] Garcia-Molina, H., “Elections in a Distributed Computing System,” IEEE Transactions on Computing, Vol. C-31, No. 1, pp. 48-59 (1982).

[32] Garey, M.R., Johnson, D.S., “Computers and Intractability: A Guide to the Theory of NP-Completeness,” W.H.Freeman and Company, San Francisco (1979).

[33] Gemmell, J., Schooler, E., Kermode, R., “A Scalable Multicast Architecture for One-to-Many Telepresentations,” Proceedings of the IEEE International Conference on Multimedia Computing Systems, ICMCS’98, pp. 128-139, Austin, TX (June 1998).

[34] RFC 2608 Guttman, E., Perkins, C., Veizades, J., Day, M., “Service Location Protocol, Version 2” (June 1999).

[35] Handley, M., Perkins, C., Whelan, E., “Session Announcement Protocol,” Internet Draft (Mar 2000).

[36] Handley, M., Schulzrinne, H., Schooler, E., Rosenberg, J., “SIP: Session Initiation Protocol,” RFC 2543 (Mar 1999).

[37] Handley, M., “An Examination of MBone Performance,” USC/ISI Research Report, ISI/RR-97-450 (Jan 1997).

[38] Hayden, M.G., “The Ensemble System,” PhD Thesis, Cornell University, Computer Science Department (Jan 1998).

[39] Hayden, M., Birman, K., "Probabilistic Broadcast," Cornell University, Computer Science, Technical Report TR96-1606 (Sept 1996).

[40] Kermode, R., Vicisano, L, "Author Guidelines for RMT Building Blocks and Protocol Instantiation Documents," Internet Draft, RMT Working Group, work in progress (June 2000).

[41] Kermode, R., "Scoped Hybrid Automatic Repeat reQuest with Forward Error Correction (SHARQFEC)," ACM SIGCOMM Computer Communication Review, Vol. 28, No. 4, (Oct 1998).

[42] Kermode, R., "Smart Network Caches: Localized Content and Application Negotiated Recovery Mechanisms for Multicast Media Distribution," PhD Thesis, Program in Media Arts and Sciences, School of Architecture and Planning, Massachusetts Institute of Technology, MA (June 1998).

[43] Menasce, D., Muntz, R., Popek, J., "A Locking Protocol for Resource Coordination in Distributed Databases.," ACM Transactions on Distributed Systems, Vol. 5, No. 2, pp. 102-138 (1980).

[44] Metcalfe, R.M., Boggs, D.R., "Ethernet: distributed packet switching for local computer networks", Communications of the ACM, Vol. 19, No. 7, pp. 395-404 (July 1976).

[45] Nonnenmacher, J., Biersack, E.W., "Scalable Feedback for Large Groups," IEEE/ACM Transactions on Networking, Vol. 7, No. 3, pp. 375-386 (June 1999).

[46] Nonnenmacher, J., "Reliable Multicast Transport to Large Groups," PhD thesis, EPF Lausanne, Switzerland (July 1998).

[47] Nonnenmacher, J., Biersack, E.W., "Optimal Multicast Feedback," IEEE, INFOCOM '98, San Francisco, CA (Apr 1998).

[48] Peleg, D., "Time-Optimal Leader Election in General Networks," Journal of Parallel and Distributed Computing, Vol. 8, pp. 96-99 (1990).

[49] Raman, S., McCanne, S., "A Model, Analysis, and Protocol Framework for Soft State-based Communication," Proceedings ACM SIGCOMM '99, pp 15-25, Cambridge, MA (Sept 1999).

[50] Raman, S., McCanne, S., Shenker, S., "Asymptotic Behavior of Global Recovery in SRM," Proceedings ACM SIGMETRICS, Vol. 26, No. 1, pp. 90-99 (June 1998).

[51] Rosenberg, J., Schulzrinne, H., "Sampling of the Group Membership in RTP," RFC 2762 (Feb 2000).

[52] Rosenberg, J., Schulzrinne, H., "Timer Reconsideration for Enhanced RTP Scalability," Proceedings of IEEE Infocom, San Francisco, CA (Mar/Apr 1998).

[53] Schooler, E.M., "A Multicast User Directory Service for Synchronous Rendezvous," Masters Thesis, Technical Report CS-TR-96-18, Computer Science Department, California Institute of Technology, CA (Aug 1996).

[54] Schulzrinne, H., Casner, S., Frederick, R., Jacobson, V., "RTP: A Transport Protocol for Real-Time Applications," RFC 1889 (Jan 1996); the revised version is an Internet Draft, work in progress (Mar 2000).

[55] Sharma, P., Estrin, D., Floyd, S., Jacobson, V., "Scalable Timers for Soft State Protocols," Proceedings of the IEEE INFOCOM, Kobe, Japan (1997).

[56] Singh, S., Kurose, J.F., "Electing 'good' leaders (election leader algorithm)," Journal of Parallel and Distributed Computing, Vol. 21, No. 2, pp. 184-201 (May 1994).

[57] van Renesse, R., Birman, K.P., Maffeis, S., "Horus: A Flexible Group Communication System," Communications of the ACM, Vol. 30, No. 4, pp. 76-83 (Apr 1996).

[58] Woodcock, B., Manning, B., "Multicast Discovery of DNS Services," Internet Draft, work in progress (Aug 1998).

[59] Yajnik, M., Kurose, J., Towsley, D., "Packet Loss Correlation in the MBone Multicast Network," University of Massachusetts, Computer Science Technical Report 96-32, Amherst, MA; IEEE Globecom Internet Workshop, London (Nov 1996).

[60] Zhang, L., Berson, S., Herzog, S., Jamin, S., "Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification," RFC 2205 (Sept 1997).

[61] Zhang, L., Deering, S., Estrin, D., Shenker, S., Zappala, D., "RSVP: A New Resource ReSerVation Protocol," IEEE Network (Sept 1993).