

# RELIABLE COMMUNICATION FOR DATACENTERS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Mahesh Balakrishnan

January 2009

© 2009 Mahesh Balakrishnan

ALL RIGHTS RESERVED

# RELIABLE COMMUNICATION FOR DATACENTERS

Mahesh Balakrishnan, Ph.D.

Cornell University 2009

Datacenter platforms have dominated the systems landscape over the last decade, offering applications the promise of scalability, availability and responsiveness at very low costs. Delivering on this promise is a significant research challenge — datacenters consist of thousands of inexpensive fault-prone components, running commodity operating systems and protocols ill-fitted for high-performance applications. Further, datacenter applications have unconventional scaling requirements and bursty workloads that frequently push systems into delays and down-time.

This thesis seeks to provide systems with low-latency primitives for reliable communication that are fundamentally scalable and robust to faults and attacks. Our focus is on the design and implementation of two protocols: Maelstrom and Ricochet. Maelstrom is a transparent network appliance for reliable and rapid communication over high-speed optical networks *between* datacenters. Ricochet is a low-latency messaging layer for clustered applications running *within* datacenters. An important aspect of these two protocols is the use of *proactive* fault-handling techniques such as Forward Error Correction (FEC) and gossip to achieve low delays and stable performance. Reactive protocols do too much too late, imposing extra delays and overheads that often send systems into spirals of degrading performance. In contrast, proactive protocols recover from faults almost instantly and impose stable, predictable overheads that prevent transient overloads and failures from translating into application unavailability.

Both Maelstrom and Ricochet use fast and simple XOR operations in novel ways that allow datacenter applications to scale in new and vital dimensions. In particular, they

create XORs at strategic points in the network (respectively, within an appliance and at multicast receivers) and from different data channels to obtain excellent recovery and latency properties. Together, these protocols enable the development of highly available applications that coordinate within and across datacenters while maintaining scalable and robust responsiveness.

## **BIOGRAPHICAL SKETCH**

Mahesh Balakrishnan grew up in New Delhi, where he spent the 90s trading computer games on floppy disks, downloading them on dial-up modems from choked BBSes and occasionally hacking them to get infinite health. A career in Computer Science seemed inevitable and he joined Georgia Tech in 2000. Three years later, he graduated with a B.S. degree and the firm resolve to find a career path that did not involve waking up before noon. He entered Cornell's Ph.D. program in the fall of 2003 and spent the next five years exercising his right to work whenever he felt like it. In the summer of 2006 Mahesh interned at Microsoft Research Silicon Valley, learning to play table tennis and to operate an espresso machine. Deeply impressed by the sunny dispositions of the Bay Area and its people, he is now heading there to accept a full-time position as a researcher at Microsoft Research.

## ACKNOWLEDGMENTS

From my advisor, Ken Birman, I learnt much about the art and philosophy of system building. Ken pushed me from day one to be an independent researcher and to ‘own’ my projects. His infectious enthusiasm for research and his irrational belief in my abilities made his office my first recourse in times of doubt. I will especially treasure the many hours spent listening to his vivid accounts of the history of systems research — and to his insightful predictions of its future.

Robbert van Renesse was a source of infallible systems wisdom and illustrated by example the importance of balance in research (and life). Dahlia Malkhi and Venugopalan Ramasubramanian at Microsoft Research were thoughtful mentors, shepherding me through the last part of the Ph.D. and making my internship with them an enriching experience. Danny Dolev’s visits to Cornell gave me an opportunity to learn distributed systems theory from one of its pioneers.

Systems research cannot be done in isolation, and I owe a big debt to my many collaborators. Tudor Marian provided hacking assistance on demand and built key pieces of the Maelstrom system. Amar Phanishayee played an important role in the evaluation of the Ricochet and Plato protocols. Both were the best collaborators one could ask for — generous with their assistance and patient with my whims.

Stefan Pleisch mentored me in my first two years at Cornell, and the white-boarding sessions in his office were among my earliest and most rewarding experiments at collaboration. Hakim Weatherspoon was an invaluable ally in the final two years, providing critical mass to a number of projects that would have been still-born without his participation. Einar Vollset was a great source and sounding board for wacky ideas, and saved me much time by discovering flaws in half-formed plans.

I’d like to thank Bill Hogan for his invaluable help in all things administrative as well as the entertaining conversations in his office. My many co-inhabitants at Upson

Hall kept me good company through the years; these include Selcuk Aya, Hitesh Ballani, Lakshmi Ganesh, Saikat Guha, Maya Haridasan, Art Munson, Benyah Shaparenko, Yeejiun Song and Bernard Wong.

Life outside Upson Hall was full of interesting people and amazing friends. In reverse order of appearance, these include Parvati, Vivek, Lavanya, Kunal, Swati, Ajay, Aishu, Anand, Vidhya, Reshmi, Saurabh, Anisa, Shobhit and Sandy. It would take me many pages to thank each of them for their support and affection.

Of course, this Ph.D. would never have been undertaken without the freedom and encouragement given to me by my parents, who ensured that schooling did not get in the way of my education. And last in this list is Surabhi — for her love, support, and the endless cups of *chai* that kept me awake through the writing of this dissertation.

## TABLE OF CONTENTS

Biographical Sketch . . . . .	iii
Acknowledgments . . . . .	iv
Table of Contents . . . . .	vi
List of Figures . . . . .	viii
<b>1 Introduction</b>	<b>1</b>
1.1 The Modern Datacenter . . . . .	1
1.2 Proactive Reliability for Datacenter Communication . . . . .	3
<b>2 Related Work</b>	<b>6</b>
2.1 A Brief History of Reliable Multicast . . . . .	9
2.2 High-Speed Long-Distance Transport . . . . .	12
2.3 Forward Error Correction . . . . .	15
2.3.1 Overview of FEC codes . . . . .	15
2.3.2 FEC in Systems . . . . .	17
<b>3 Maelstrom</b>	<b>18</b>
3.1 Loss Model . . . . .	23
3.2 Existing Reliability Options . . . . .	26
3.2.1 The Case For (and Against) FEC . . . . .	28
3.3 Maelstrom Design and Implementation . . . . .	30
3.3.1 Basic Mechanism . . . . .	30
3.3.2 Flow Control . . . . .	31
3.3.3 Layered Interleaving . . . . .	33
3.3.4 Back-of-the-Envelope Analysis . . . . .	38
3.3.5 Local Recovery for Receiver Loss . . . . .	40
3.3.6 Implementation Details . . . . .	41
3.3.7 Buffering Requirements . . . . .	43
3.3.8 Other Performance Enhancing Roles . . . . .	44
3.4 Evaluation . . . . .	44
3.4.1 Throughput Metrics . . . . .	45
3.4.2 Latency Metrics . . . . .	48
3.4.3 Layered Interleaving and Bursty Loss . . . . .	50
<b>4 Ricochet</b>	<b>56</b>
4.1 System Model . . . . .	59
4.2 The Design of a Time-Critical Multicast Primitive . . . . .	62
4.2.1 The Timeliness of (Scalable) Reliable Multicast Protocols . . . . .	62
4.3 Receiver-based Forward Error Correction in a Single Group . . . . .	66
4.3.1 Analysis of Receiver-based FEC . . . . .	69
4.4 Lateral Error Correction for Multiple Groups . . . . .	72
4.4.1 Algorithm Overview . . . . .	74

4.5	Details of the Ricochet Implementation . . . . .	81
4.5.1	Implementation of Repair Bins . . . . .	82
4.5.2	Staggering for Bursty Loss . . . . .	82
4.5.3	Multi-Group Views . . . . .	83
4.5.4	Membership and Failure Detection . . . . .	84
4.5.5	Performance . . . . .	86
4.5.6	Buffering and Loss Control . . . . .	86
4.5.7	NAK Layer for 100% Recovery . . . . .	87
4.5.8	Optimizations . . . . .	88
4.5.9	Message Ordering . . . . .	88
4.6	Evaluation . . . . .	88
4.6.1	Distribution of Recoveries in Ricochet . . . . .	90
4.6.2	Tunability of LEC in multiple groups . . . . .	92
4.6.3	Scalability . . . . .	93
4.6.4	Loss Rate and LEC Effectiveness . . . . .	95
4.6.5	Resilience to Bursty Losses . . . . .	97
4.6.6	Effect of Group and System Size . . . . .	99
<b>5</b>	<b>Future Work and Conclusion</b>	<b>100</b>
5.1	Rethinking Transport for the Datacenter . . . . .	100
5.2	The Three P's — Power, Privacy, Parallelism . . . . .	102
	<b>Bibliography</b>	<b>104</b>

## LIST OF FIGURES

3.1	Example Lambda Network . . . . .	19
3.2	Maelstrom Communication Path . . . . .	20
3.3	Loss Rates on TeraGrid . . . . .	24
3.4	Interleaving with index 2: separate encoding for odd and even packets . . . . .	28
3.5	Basic Maelstrom mechanism: repair packets are injected into stream transparently . . . . .	31
3.6	Flow Control Options in Maelstrom . . . . .	32
3.7	Layered Interleaving: ( $r = 3, c = 3$ ), $I = (1, 10, 100)$ . . . . .	33
3.8	Layered Interleaving Implementation: ( $r = 5, c = 3$ ), $I = (1, 4, 8)$ . . . . .	36
3.9	Staggered Start . . . . .	37
3.10	Comparison of Packet Recovery Probability: $r=7, c=2$ . . . . .	39
3.11	User-Space Throughput against Loss Rate (Top) and One-Way Link Latency (Bottom) . . . . .	46
3.12	Kernel Throughput against Loss Rate (Top) and One-Way Link Latency (Bottom). . . . .	47
3.13	Throughput of Split-Mode Buffering Flow Control against One-Way Link Latency. . . . .	49
3.14	Per-Packet One-Way Delivery Latency against Loss Rate (Top) and Link Latency (Bottom) . . . . .	50
3.15	Packet delivery latencies . . . . .	51
3.16	Relatively prime interleaves offer better performance . . . . .	52
3.17	Layered Interleaving Recovery Percentage and Latency . . . . .	53
3.18	Latency Histograms for $I=(1,11,19)$ (Left) and $I=(1,19,41)$ (Right) — Burst Sizes 1 (Top), 20 (Middle) and 40 (Bottom) . . . . .	54
3.19	Reed-Solomon versus Layered Interleaving . . . . .	55
4.1	Datacenter Loss is bursty and uncorrelated across nodes: receiver $r_1$ (Top) joins groups $A$ and $B$ and exhibits bursty loss, whereas receiver $r_2$ (Bottom) joins only group $A$ and experiences zero loss. . . . .	60
4.2	SRM's Discovery Latency vs. Groups per Node, on a 64-node cluster, with groups of 10 nodes each. Error bars are min and max over 10 runs. . . . .	64
4.3	Ricochet Packet Structure . . . . .	67
4.4	LEC in 2 Groups: Receiver $n_1$ can send repairs to $n_2$ that combine data from both groups $A$ and $B$ . . . . .	73
4.5	$n_1$ belongs to groups $A, B, C$ : it divides them into disjoint regions $abc, ab, ac, bc, a, b, c$ . . . . .	75
4.6	$n_1$ selects proportionally sized chunks of $c_A$ from the regions of $A$ . . . . .	76
4.7	Mappings between repair bins and regions: the repair bin for $ab$ selects 0.4 targets from region $ab$ and 0.8 from $abc$ for every repair packet. Here, $c_A = 5, c_B = 4$ , and $c_C = 3$ . . . . .	78
4.8	Distribution of Recoveries: LEC + NAK for varying degrees of loss . . . . .	91

4.9	Tuning LEC : tradeoff points available between recovery %, overhead % (left y-axis) and avg recovery latency (right y-axis) by changing the rate-of-fire ( $r, c$ ). . . . .	92
4.10	Scalability in Groups . . . . .	93
4.11	CPU time (Top) and XORs (Bottom) per data packet . . . . .	95
4.12	Impact of Loss Rate on LEC . . . . .	96
4.13	Resilience to Burstiness . . . . .	97
4.14	Staggering allows Ricochet to recover from long bursts of loss. . . . .	98
4.15	Effect of Group Size . . . . .	99

# CHAPTER 1

## INTRODUCTION

### 1.1 The Modern Datacenter

The modern datacenter had its genesis in the mid-90s, when dot-com companies found that they could scale their operations to millions of users by deploying large server farms of inexpensive machines. The technology behind commodity datacenters was conceptually simple: early Internet applications such as webpages and search were inherently parallel and could easily be distributed across a cluster of machines. However, the economic implications of commodity datacenters were revolutionary — organizations were able to acquire and maintain massive infrastructure at bargain prices, opening the door to a world of diverse online services.

The intervening decade has seen the rise of datacenter computing as the paradigm of choice for practically every application domain. The move to datacenters has been powered by two separate trends. On the one hand, application domains typically tied to monolithic ‘big-iron’ machines — such as finance, military and aerospace — have migrated to commodity clusters in search of leaner operational budgets. In parallel, functionality and data usually associated with personal computing has moved into the datacenter; users continuously interact with remote sites while using their local computers, whether to run intrinsically online applications such as email, chat, games and blogs, or to manipulate data traditionally stored locally, such as documents, spreadsheets, videos and photos. In effect, modern architectures are converging towards *cloud computing*, a paradigm where all user activity is funneled into large datacenters via high-speed networks.

Datacenter computing has resulted in a fundamental shift in how software companies operate. Software is no longer a product to be shipped and forgotten — it's a service that has to be continuously online and available. Companies have deployed massive datacenters that operate round-the-clock, serving content to and aggregating data from millions of users. And increasingly, these datacenters are networked with each other through high-speed optical networks that mirror data between different geographies — in order to redirect clients to closer sites, and to protect against disasters and meltdowns. This picture of massive 'information factories' [39] interlinked with each other through hidden networks of high-speed optics is increasingly applicable to any organization with an online presence.

A critical implication of the datacenter computing model is that the user of an online service expects the same performance from it as she might from an application running on her desktop computer. She expects the service to store data reliably and to always be available. And, most importantly, she expects an online service located in a remote datacenter to be as immediately *responsive* as a desktop application.

For service providers, delivering on this expectation is an engineering task of immense proportions. Datacenters are composed of thousands of failure-prone components and exhibit a bewildering array of failure modes. Each level of the software and hardware stack has its own litany of error modes, ranging from simple to byzantine — hung machines, blown fans, corrupted disks, bad network cards, overloaded routers and switches, buggy software, rogue malware, partitioned networks; the list is endless.

When faults occur, enterprises have to react extremely quickly to prevent service down-time. In Gartner surveys conducted in 2005 and 2006 [78], datacenter administrators were asked if they had real-time infrastructure, defined as a datacenter that could react to arbitrary faults and overloads within seconds or minutes. While 63% to 73%

of the respondents stated that they thought having real-time recovery capabilities was important, only 15% to 19% reported actually having any degree of such capabilities in their datacenters.

How do we build systems that are *autonomic* — detecting faults and recovering from them automatically — in real-time? This thesis examines real-time fault-tolerance for a narrow but extremely significant layer of the software stack: the network protocols used to ferry data reliably within and between datacenters. To architect networked datacenter systems that can respond to arbitrary faults within seconds, we need to build communication primitives that can recover from common network-level failure patterns — specifically, packet loss — within milliseconds.

## 1.2 Proactive Reliability for Datacenter Communication

Existing reliable communication protocols are *reactive*, and have an associated cost and latency of reaction. In many cases, they react too slowly to packet loss. And often, they over-react — flooding the system with recovery traffic that potentially causes further data loss, as well as throttling back excessively on sending speeds to avoid more loss. Protocols such as TCP/IP were designed for congested public networks and do not work well in the high-speed networks deployed within and between datacenters.

In this thesis, we describe two systems that use proactive reliability techniques. **Maelstrom** (in NSDI 2008 [16]) is a transparent network appliance for reliable and rapid communication over high-speed optical networks *between* datacenters. **Ricochet** (in NSDI 2007 [15]) is a low-latency messaging layer for clustered applications running *within* datacenters. Both protocols use fast and simple XOR operations for reliable communication in novel ways that allow datacenter applications to scale in new and

vital dimensions. In particular, they create XORs at strategic points in the network (respectively, at multicast receivers and in an appliance) and from different data channels to obtain excellent recovery and latency properties. Together, these protocols allow for the development of highly available applications that coordinate within and across datacenters while maintaining scalable and robust responsiveness.

Maelstrom and Ricochet are part of a larger family of protocols and mechanisms that use *Forward Error Correction (FEC)*. In the simplest terms, FEC involves injecting redundant repair traffic into a data path to enable the recovery of data lost in transit. For example, a sender could add extra XORs of transmitted data packets to the outgoing stream; the receiver can then recover a missing data packet using the XOR and other correctly received data packets. Decades of research into FEC for applications such as satellite communication and optical disk writing have produced a plethora of codes which optimize along different parameters such as encoding speed or scalability in input size.

Importantly, FEC eliminates the feedback loop between the receiver and the sender in a communication channel. This key property makes FEC ideally suited for settings where the receiver cannot reply back to the sender — for example, over long-distance links where the receiver’s feedback takes too long to arrive at the sender, or in multicast channels where multiple receivers can swamp a sender with feedback. Maelstrom and Ricochet apply FEC in novel ways to these settings, leveraging the absence of a feedback loop and engineering around other scalability and performance concerns.

FEC has other useful properties in the context of scalable datacenter communication. For example, it imposes a steady, constant and tunable overhead on the system — at any given point of time, the fraction of resources devoted to recovery activity can be easily quantified and modulated. This enables datacenter designers to provision their

networks accordingly. Importantly, it prevents datacenter ‘brown-outs’, where an overly aggressive protocol trying to recover a small segment of traffic can bring the entire system down.

Importantly, Maelstrom and Ricochet are examples of a ‘systems’ approach to FEC — instead of focusing on the exact algorithm to be used within an encoding channel, they explore alternative definitions of the channel itself. Both protocols aggregate data across channels for efficiency and scalability, and send redundant information along non-traditional pathways that cross channel boundaries. Accordingly, Maelstrom and Ricochet are concerned with two important questions: *where* in the system is FEC generated, and *what* is it generated from?

## CHAPTER 2

### RELATED WORK

The commodity datacenter was born in a research lab. A number of research projects in the mid-90s — such as the NOW project at Berkeley [13] — demonstrated that a combination of commodity workstations and interconnects could be used to create first-class cluster platforms. Subsequent efforts in academia and industry [37,40,41] showed that these commodity clusters were ideal from a functionality and cost perspective for deploying large-scale Internet services.

Experience with real-world datacenters has led researchers and practitioners [33,36,72] to posit that the core principles of large-scale system design lie in a simple mantra — partition for scalability, replicate for availability. Partitioning involves dividing the state and processing of a service over multiple machines according to some index. For instance, an email service could be partitioned using the user identifier as an index, with the mailboxes of different users on separate machines. The partitioning function can be arbitrary, as long as it maps each identifier to a single partition; for example, all users with identifiers starting with the letter ‘A’ could be directed to partition  $p_1$ , those with identifiers starting with the letter ‘B’ could be directed to partition  $p_2$ , and so on. In practice, the function is chosen to provide good load-balancing properties across partitions.

Partitioning is a simple and effective primitive that allows a single service to scale almost limitlessly in key dimensions during failure-free operation — for example, Internet services such as mailboxes can support millions of users by partitioning state across thousands of machines. A far more difficult challenge for datacenter designers involves ‘hardening’ the partitioned service — ensuring that it stays responsive and available at massive scales even when failures and overloads occur.

The key to achieving these properties is *redundancy* — of service functionality and data. Practically every piece of data stored within a datacenter is copied to multiple locations within the datacenter as well as outside it. When used wisely, redundancy adds a host of critical properties to systems, including fault-tolerance, responsiveness, availability, disaster-tolerance, throughput scalability and stability; however, it comes at the significant cost of having to keep multiple copies of data consistent across the system. Depending on the purpose, semantics and mechanism of maintaining data consistency, there are many different categories of redundancy. Three common examples are *replication*, *caching* and *mirroring*.

**Replication** traditionally refers to data redundancy with strong consistency semantics. Replication is most commonly added to systems for fault-tolerance and availability in the face of node failures and crashes. While replication typically refers to redundancy of data, a form of it called *state-machine replication* [77] is used for redundancy of functionality, allowing the execution of multiple instances of the same software on different machines.

**Caching** is key to the performance of datacenter systems at massive scale. Most modern datacenters store much of their critical state within a database ‘third tier’ that is extremely fault-tolerant but correspondingly slow. To obtain rapid responsiveness from such architectures, data values are cached extensively by second and first-tier constructs.

**Mirroring** of datacenters is used for both performance as well as disaster tolerance [52, 71]. Internet companies routinely maintain multiple datacenters to serve clients in different geographies; for example, a user in France is directed to a datacenter in Paris instead of one in Seattle. Organizations such as financial brokerages mirror critical data in remote locations to guard against localized disasters. In both cases, data written within one datacenter is mirrored across high-speed optical links to one or more remote

datacenters.

Massive redundancy of data and functionality has one significant implication — the state of a single logical service is duplicated over large numbers of discrete nodes. As a result, updating the state of the service involves contacting a set of machines simultaneously to keep all copies of the state consistent. Consequently, the communication layer within a datacenter must be highly efficient for patterns of traffic that involve simultaneous updates to sets or *groups* of machines; or, in networking terms, *multicast* patterns.

Further, updating copies of data stored in geographically remote datacenters involves communication over long-haul optical links that have vastly different properties compared to the networks within datacenters. Accordingly, the communication layer must also support efficient and timely transport of data across long-distance high-bandwidth links interconnecting datacenters.

The utility of long-haul communication extends beyond simple data mirroring; it is a vital capability for any application that needs to coordinate across datacenters, from mundane examples like SSH sessions to complex componentized systems spread over multiple geographical sites. Similarly, maintaining the consistency of replicated or cached data is only one of the many uses of multicast communication within a datacenter. Publish-Subscribe [19] or pub-sub is a popular system-building abstraction where applications *publish* and *subscribe* data to *topics* that are essentially multicast channels.

In this chapter, we describe options available to datacenter systems for multicast communication within a datacenter as well as long-haul communication between datacenters. Since the solutions we propose later in this chapter are heavily based on Forward Error Correction (FEC) techniques, we also discuss the state-of-the-art in FEC

algorithms and protocols.

## 2.1 A Brief History of Reliable Multicast

In abstract terms, *multicast* refers to the transmission of a single data packet simultaneously to multiple receivers. Network support for multicast communication was introduced in the form of IP Multicast [30] in 1988; over the next two decades, it has become the sole option provided by commodity operating systems and networks for multicast transport.

IP Multicast is *unreliable*. It provides nearly identical semantics to IP unicast, routing data with best effort guarantees to multiple receiving end-hosts. In the mid-90s, the successful deployment of IP Multicast in routers and operating systems raised an important question for networking researchers — what should the analogue of TCP be in the multicast communication stack? Accordingly, a slew of reliable protocols were developed, driven primarily by the emerging ‘killer apps’ for multicast: content dissemination, file distribution, streaming audio/video and other applications involving potentially thousands of receivers distributed across a world-wide Internet. All these protocols were extremely scalable — in the single dimension of group size.

Today, the advent of datacenter computing has seen reliable multicast used widely to spray data across multiple machines. Many of the commercial multicast products used today by datacenter systems have their roots in the reliable multicast protocols developed in the last decade for the wide-area Internet. However, these protocols were not designed to be used in clustered settings. As discussed in length later in this thesis, they fail to scale in dimensions of vital importance, such as the number of distinct multicast channels in the system or the number of senders to a single channel.

Early reliability mechanisms for multicast communication were similar to TCP, with receivers sending back positive acknowledgments (ACKs) to the sender to signaling successful packet reception. However, ACK-based schemes did not scale to more than a few receivers due to *ACK implosion* — in a group with  $n$  receivers, the sender received  $n$  ACKs for each data packet it sent. For example, a sender transmitting to a group with ten receivers would have to process ten ACK packets for each data packet it sent out, severely limiting its maximum throughput to the group.

The ACK implosion problem sparked off a wave of academic research in the mid-90s aimed at producing the perfect reliable multicast protocol — one that would work well with large numbers of receivers over the wide-area Internet. One approach was to use *acknowledgment trees*, as introduced in the RMTP protocols [58, 65]. ACK trees made up of receivers are used to ‘bubble’ up acknowledgments to the sender, with a single tree used for each sender within each group.

A more common solution to ACK implosion was the use of negative acknowledgments (NAKs). A receiver generates a negative acknowledgment when it realizes that it’s missing a packet, most commonly by examining the sequence numbers attached by the sender on each packet. In the straightforward implementation, the NAK is sent by the receiver straight back to the sender, which responds with a unicast retransmission of the missing packet.

While early solutions based on ACK trees and NAKs scaled to dozens of receivers, they faced a fundamental barrier to further scaling — they expected a single sender to handle retransmissions for large numbers of receivers; and, by implication, to buffer sent data, track its delivery at each receiver and garbage-collect it only when it was received by the entire group. The next generation of protocols was thus *receiver-based*, where receivers in the multicast group interacted with each other to obtain missing packets that

were delivered at some receivers but not at others.

Perhaps the most well-known of the receiver-based solutions is Scalable Reliable Multicast (SRM) [35], proposed in 1997 by Floyd et al. SRM uses sender sequencing and NAKs — however, the NAKs are not sent back to the sender, but are instead multicast to the entire group of receivers. To ensure that the network is not flooded by retransmissions from every receiver that gets a NAK request, SRM uses a form of probabilistic dampening, where receivers retransmit messages in response to NAKs with a certain probability.

SRM subsequently became the basis of the PGM protocol [82], a commercial initiative by a number of companies including Cisco, Microsoft and Tibco. PGM incorporates several techniques for locality awareness, restricting NAK requests to sub-trees of the routing topology. An important feature of both SRM and PGM is that they do not require the receivers to maintain explicit knowledge of each other — all recovery traffic travels over the IP Multicast group used for sending data.

Close on the heels of SRM came Bimodal Multicast (or pbcast) [20], another receiver-based protocol. Pbcast is significantly different from previous reliability protocols — it provides probabilistic guarantees of reliability via a *gossip* protocol. Gossip was first proposed by Demers et al. in 1988 [32] as a way of keeping distributed state consistent — nodes randomly choose other nodes in the system and ‘gossip’ about the state, reconciling inconsistencies. In pbcast, receivers randomly choose other receivers and gossip about the messages they received in the immediate past. If a pair of gossiping receivers discovers that one has received a message the other has not, the former sends the latter the missing message.

In the original description of pbcast, each receiver needs to know the membership of

the entire group, since the protocol requires it to randomly select another receiver from the group to gossip with. This limits the scalability of the protocol, despite its resilience to stale membership data. Lightweight probabilistic broadcast [34] improved upon the original by requiring each receiver to know only a limited subset of the entire group.

The use of packet-level Forward Error Correction in reliable multicast protocols was first proposed by Huitema in 1996 [47] and later implemented by Luigi Rizzo [74]. FEC-based protocols typically involve the sender transmitting redundant FEC packets to the multicast group, which could then be used by receivers to recover missing data packets. FEC allows receivers to proactively recover from packet loss without contacting either the sender of the original data or any other receiver. As a result, these protocols are insensitive to the number of receivers in the group; however, the efficacy of the FEC codes in recovering lost data is very sensitive to the patterns of loss observed at the receivers.

A variant of the FEC-based protocols involves having the sender use FEC only for retransmissions [67]. For example, if two receivers require two different packets to be retransmitted, the sender transmits a single XOR of the two packets and each receiver generates its missing data packet by combining the XOR and the other successfully received data packet.

## **2.2 High-Speed Long-Distance Transport**

TCP/IP is the reigning standard for unicast communication. It has deep and exclusive embeddings in commodity operating systems and networking APIs. As a result, TCP/IP is used by applications as the default option for reliable communication over a vast array of different networking media, including the long-haul optical networks interconnecting

datacenters.

However, TCP/IP does not work well on networks with high bandwidth delay product, a fact that has been established through analytical proofs [57,68] as well as practical experience. Consequently, the last decade has seen much research aimed at developing viable TCP/IP variants and alternatives for high-speed long-distance networks.

One approach involves the bottom-up redesign of the network to better support high-speed long-haul networks. For example, XCP [53] proposes modifying routers to provide explicit feedback to applications; the router periodically computes the spare capacity available, distributes it across current flows and then informs the flows of the appropriate window resizes. While XCP utilizes bandwidth on high-speed networks extremely well, it requires the network to be heavily modified; as a result, it has not seen significant deployment.

A different avenue of research aims at using delay as a congestion signal for TCP/IP, as opposed to loss. The key intuition is that an approaching network congestion event is manifested *a priori* by buffers filling up at routers, which translates into increases in the round-trip delay of the link. By using delay as a congestion signal, TCP/IP can cut back *before* congestion occurs, without first pushing the network into a loss pattern. Importantly, delay-based protocols are not susceptible to back-offs triggered by non-congestion causes of loss, such as misconfigured or malfunctioning hardware. Delay-based congestion control was first proposed in the TCP Vegas protocol in the mid-90s [22]. More recently, FAST TCP [90] has emerged as a modern delay-based alternative to commodity TCP/IP on high-speed networks. A similar vein of work tackles differentiating between loss caused by congestion and non-congestion events, to enable TCP/IP to react appropriately by cutting back on the window size only when congestion occurs [23,55,70].

Another approach involves replacing the conventional TCP/IP ‘Additive Increase Multiplicative Decrease’ (AIMD) curve with more sophisticated control curves. For example, the BIC [93] protocol’s window resizing mechanisms alleviate problems that commodity TCP/IP has with preserving fairness across flows with different window sizes and different RTTs. CUBIC [73] further improves on BIC’s fairness properties, factoring in the passage of wall-clock time while resizing the window to prevent the RTT of a flow from affecting its throughput utilization.

A large body of work in high-speed data transfer has emerged from the efforts of the e-science and grid computing communities, which were early adopters of high-bandwidth long-haul networks for shuttling large data-sets between supercomputing sites. Many of these solutions replace TCP/IP with application-level protocols that operate above UDP. For instance, SABUL [42] and Tsunami [63, 88] are application-level flow control protocols that use UDP for the data plane and TCP for control traffic. Similarly, RBUDP [46] is an aggressive ‘blast’ protocol that attempts to send data over UDP at the maximum constant rate the network will allow. An interesting alternative solution that does not require changing or eliminating commodity TCP/IP involves striping application-level flows across multiple parallel TCP/IP flows, an idea first described in Pockets [81].

Performance enhancing proxies were proposed in RFC 3135 [21] as a means to enhance TCP/IP performance on specialized networks — such as wireless networks or long-distance links — without changing the end-host protocol stack [25, 64, 66, 86].

## 2.3 Forward Error Correction

FEC is the simple idea of injecting redundancy into a channel to protect against data loss or corruption. Receivers can use the redundancy information to reconstruct lost data without requesting extra information from the sender. As a result, no feedback channel needs to exist from the receiver back to the sender, a property of FEC that made it indispensable in settings such as satellite and space communication.

### 2.3.1 Overview of FEC codes

All FEC schemes can be categorized as either **block** or **convolutional** codes. A block code operates on discrete chunks or blocks of data — from  $k$  input symbols, it generates  $n$  output symbols. The decoder can reconstruct the original  $k$  input symbols from any  $k$  of the total  $n$  output symbols. As a result, the block code is resilient to a maximum of  $n - k$  losses. Note that a symbol could be a single bit or a large set of bits — for example, a network packet or a disk block.

In contrast, a convolutional code operates on a stream of data. The function used by the encoder to generate  $n$  bits of output from  $m$  bits of input depends on the last  $k$  bits of input. The value  $k$  is called the constraint length and the value  $\frac{m}{n}$  is the rate of the code. The inability of the decoder to recover a symbol can propagate and affect other decoding operations downstream.

Concatenated codes involve wrapping one code within another; the inner code corrects most of the errors while the outer code ‘mops’ up the remaining errors. Both block and convolutional codes can be either *systematic* or *unsystematic* [76]. A systematic code includes the input symbols verbatim in the set of output symbols, while an unsys-

tematic code does not.

The most well-known block code is Reed-Solomon [92], which is used in applications as diverse as satellite communication and CD/DVD encoding. Traditional block codes like Reed Solomon are *optimal* — the receiver needs exactly  $k$  output symbols to recover the original  $k$  input symbols. A recently explored class of block codes are non-optimal codes, where the receiver requires slightly more than  $k$  output symbols to recover the original  $k$  input symbols. Typically, non-optimal codes represent a trade-off between the number of packets required to reconstruct the original and the speed and scalability of encoding and decoding. For example, Tornado codes [60] are orders of magnitude faster than Reed-Solomon and much more scalable in input size, but are non-optimal and require more data to be received at the decoder.

Tornado codes operate by creating a bipartite graph of nodes representing input symbols and output symbols, where each output symbol is connected to multiple input symbols and vice versa. The value of each output symbol is simply the XOR of all the input symbols it's linked to. The output symbols can then themselves be recursively used as input symbols, creating another layer of output symbols. The key to Tornado's scalability is the limited reliance of each output symbol on a handful of input symbols, whereas in Reed-Solomon each output symbol depends on every input symbol.

Tornado codes require both the sender and the receiver(s) to decide *a priori* on an encoding graph, and consequently on the rate of the encoding. An improvement came with the invention of *rateless* codes such as LT codes [59], where additional symbols could be generated by the encoder as required. Raptor codes [80] further improved coding efficiency by wrapping LT codes within an outer Tornado code — as a result, the LT code only needs to recover a certain fraction of its own input symbols (which happen to be the output symbols of the outer Tornado code).

The Tornado, Raptor and LT codes were part of a larger effort to create a *digital fountain* — a data source that would continuously transmit a fixed size input (for example, a file) in the form of encoded packets to large numbers of nodes, requiring each node to only receive as many packets as the original input contained. A node receives data from the source in much the same manner that a person fills a cup of water from a fountain — it does not matter which drops of water are received, only that enough drops have been collected to fill the cup.

### **2.3.2 FEC in Systems**

Packet-level FEC was first described for high-speed WAN networks as early as 1990 [79]. Subsequently, it was applied by researchers in the context of ATM networks [18]. Interest in packet-level FEC for IP networks was revived in 1996 [47] in the context of both reliable multicast and long-distance communication. Rizzo subsequently provided a working implementation of a software packet-level FEC engine [74, 75].

As mentioned before, the concept of a digital fountain [24] led to renewed interest in using FEC for large-scale applications such as Internet content distribution [24, 62]. FEC has also been used to provide quality-of-service in overlay networks, as described in the OverQos system [83].

## CHAPTER 3

### MAELSTROM

The emergence of commodity clusters and datacenters has enabled a new class of globally distributed high-performance applications that coordinate over vast geographical distances. For example, a financial firm’s New York City datacenter may receive real-time updates from a stock exchange in Switzerland, conduct financial transactions with banks in Asia, cache data in London for locality and mirror it to Kansas for disaster-tolerance.

To interconnect these bandwidth-hungry datacenters across the globe, organizations are increasingly deploying private ‘lambda’ networks. Raw bandwidth is ubiquitous and cheaply available in the form of existing ‘dark fiber’; however, running and maintaining high-quality *loss-free* networks over this fiber is difficult and expensive. Though high-capacity optical links are almost never congested, they drop packets for numerous reasons — dirty/degraded fiber [43], misconfigured/malfunctioning hardware [49, 50] and switching contention [56], for example — and in different patterns, ranging from singleton drops to extended bursts [45, 54]. Non-congestion loss has been observed on long-haul networks as well-maintained as Abilene/Internet2 and National LambdaRail [44, 45, 49, 50].

The inadequacy of commodity TCP/IP in high bandwidth-delay product networks is extensively documented [53, 57, 68]. TCP/IP has three major problems when used over such networks. First, TCP/IP suffers throughput collapse if the network is even slightly prone to packet loss. Conservative flow control mechanisms designed to deal with the systematic congestion of the commodity Internet react too sharply to ephemeral loss on over-provisioned links — a single packet in ten thousand is enough to reduce TCP/IP throughput to a third over a 50 ms gigabit link, and one in a thousand drops it by an

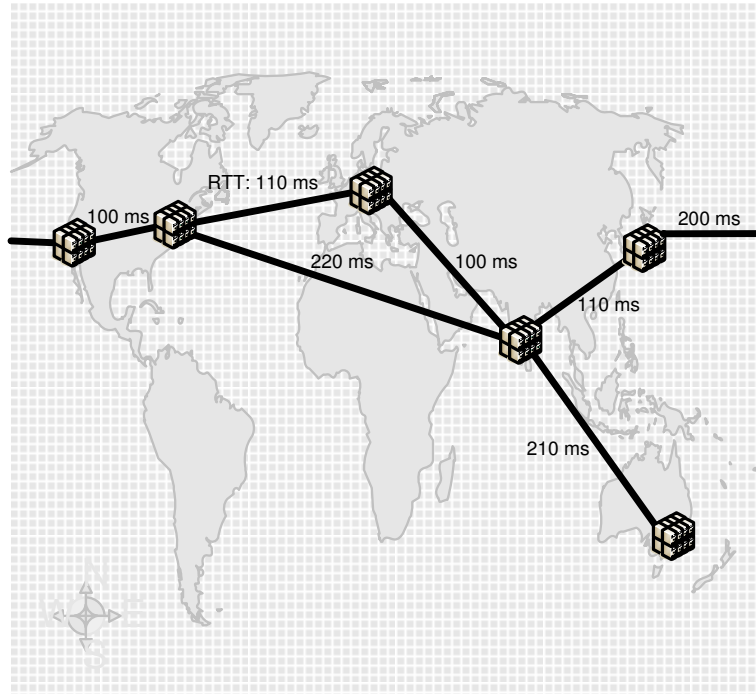


Figure 3.1: Example Lambda Network

order of magnitude.

Second, real-time or interactive applications are impacted by the reliance of reliability mechanisms on acknowledgments and retransmissions, limiting the latency of packet recovery to at least the Round Trip Time (RTT) of the link. If delivery is sequenced, as in TCP/IP, each lost packet acts as a virtual ‘road-block’ in the FIFO channel until it is recovered. Third, TCP/IP requires massive buffers at the communicating end-hosts to fully exploit the bandwidth of a long-distance high-speed link, even in the absence of packet loss.

Deploying new loss-resistant alternatives to TCP/IP is not feasible in corporate datacenters, where standardization is the key to low and predictable maintenance costs; neither is eliminating loss events on a network that could span thousands of miles. Accordingly, there is a need to *mask* loss on the link from the commodity protocols running at end-hosts, and to do so *rapidly* and *transparently*. Rapidly, because recovery delays

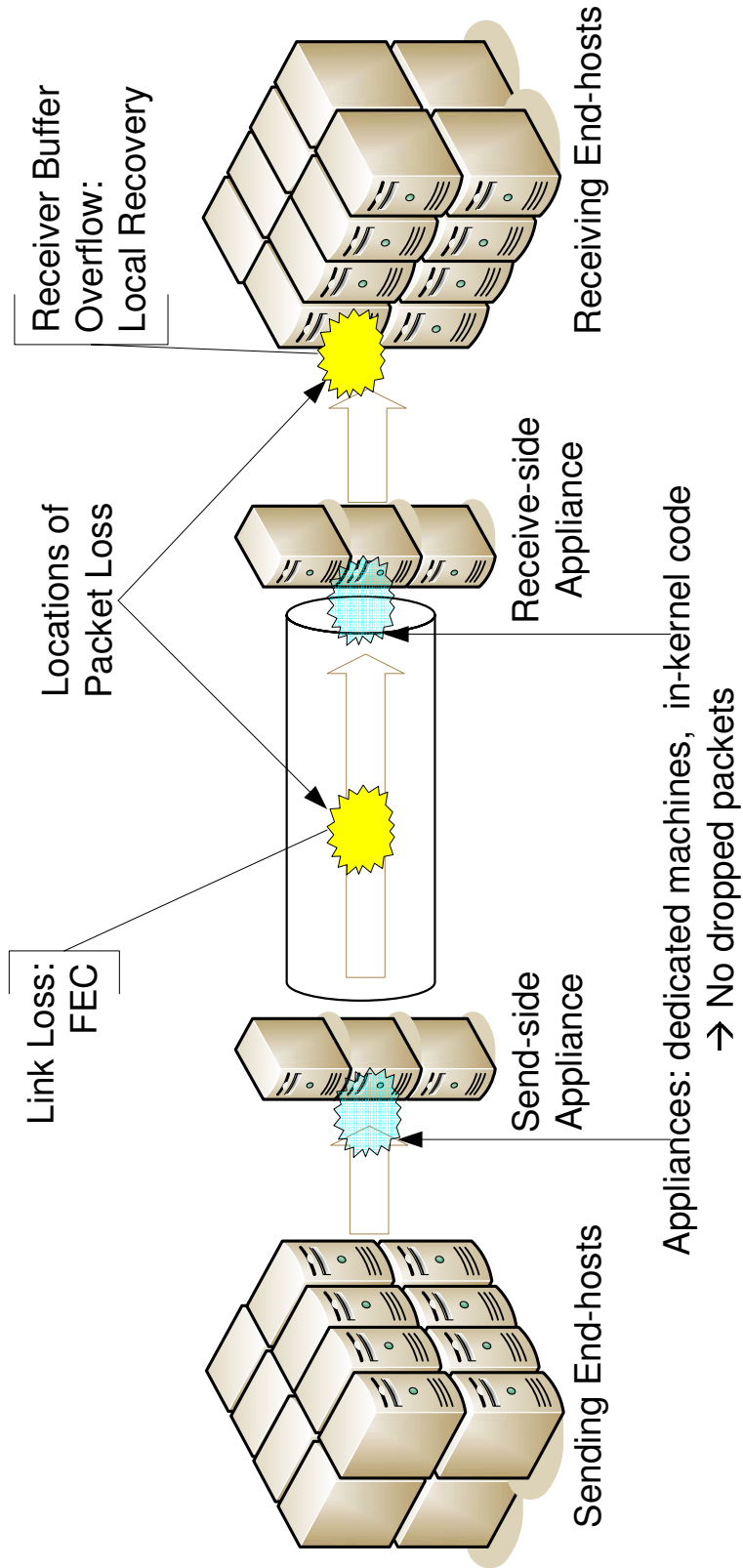


Figure 3.2: Maelstrom Communication Path

for lost packets translate into dramatic reductions in application-level throughput; and transparently, because applications and OS networking stacks in commodity datacenters cannot be rewritten from scratch.

Forward Error Correction (FEC) is a promising solution for reliability over long-haul links [74] — packet recovery latency is independent of the RTT of the link. While FEC codes have been used for decades within link-level hardware solutions, faster commodity processors have enabled packet-level FEC at end-hosts [47, 75]. End-to-end FEC is very attractive for inter-datacenter communication: it’s inexpensive, easy to deploy and customize, and does not require specialized equipment in the network linking the datacenters. However, end-host FEC has two major issues — First, it’s not transparent, requiring modification of the end-host application/OS. Second, it’s not necessarily rapid; FEC works best over high, stable traffic rates and performs poorly if the data rate in the channel is low and sporadic [15], as in a single end-to-end channel.

In this chapter, we present the Maelstrom Error Correction appliance — a rack of proxies residing between a datacenter and its WAN link (see Figure 3.2). Maelstrom encodes FEC packets over traffic flowing through it and routes them to a corresponding appliance at the destination datacenter, which decodes them and recovers lost data. Maelstrom is completely transparent — it does not require modification of end-host software and is agnostic to the network connecting the datacenters. Also, it eliminates the dependence of FEC recovery latency on the data rate in any single node-to-node channel by encoding over the *aggregated* traffic leaving the datacenter. Additionally, Maelstrom uses a new encoding scheme called *layered interleaving*, designed especially for time-sensitive packet recovery in the presence of bursty loss.

Maelstrom’s positioning as a network appliance reflects the physical infrastructure of modern datacenters — clean insertion points for proxy devices exist on the high-

speed lambda links that interconnect individual datacenters to each other. Maelstrom can operate as either a passive or active device on these links. Of the three problems of TCP/IP mentioned above, Maelstrom solves the first two — throughput collapse and real-time recovery delays — while operating as a passive device that does not intervene in the critical communication path. In active mode, Maelstrom eliminates the need for massive buffers at end-hosts as well, at the cost of adding a point of failure in the network path.

The contributions of this chapter are as follows:

- We explore end-to-end FEC for long-distance communication between datacenters, and argue that the rate sensitivity of FEC codes and the opacity of their implementations present major obstacles to their usage.
- We propose Maelstrom, a gateway appliance that transparently aggregates traffic and encodes over the resulting high-rate stream.
- We describe *layered interleaving*, a new FEC scheme used by Maelstrom where for constant encoding overhead the latency of packet recovery degrades gracefully as losses get burstier.
- We discuss implementation considerations. We built two versions of Maelstrom; one runs in user mode, while the other runs within the Linux kernel.
- We evaluate Maelstrom on Emulab [91] and show that it provides near lossless TCP/IP throughput and latency over lossy links, and recovers packets with latency independent of the RTT of the link and the rate in any single channel.

### 3.1 Loss Model

Packet loss typically occurs at two points in an end-to-end communication path between two datacenters, as shown in Figure 3.2 — in the wide-area network connecting them and at the receiving end-hosts. Loss in the lambda link can occur for many reasons, as stated previously: transient congestion, dirty or degraded fiber, malfunctioning or misconfigured equipment, low receiver power and burst switching contention are some reasons [43, 49, 50, 51, 56]. Loss can also occur at receiving end-hosts within the destination datacenter; these are usually cheap commodity machines prone to temporary overloads that cause packets to be dropped by the kernel in bursts [15] — this loss mode occurs with UDP-based traffic but not with TCP/IP, which advertises receiver windows to prevent end-host buffer overflows.

What are typical loss rates on long-distance optical networks? The answer to this question is surprisingly hard to determine, perhaps because such links are a relatively recent addition to the networking landscape and their ownership is still mostly restricted to commercial organizations disinclined to reveal such information. One source of information is TeraGrid [10], an optical network interconnecting major supercomputing sites in the US. TeraGrid has a monitoring framework within which ten sites periodically send each other 1 Gbps streams of UDP packets and measure the resulting loss rate [11]. Each site measures the loss rate to every other site once an hour, resulting in a total of 90 loss rate measurements collected across the network every hour. Figure 3.3 shows that between Nov 1, 2007 and Jan 25, 2007, 24% of all such measurements were over 0.01% and a surprising 14% of them were over 0.1%. After eliminating a single site (Indiana University) that dropped incoming packets steadily at a rate of 0.44%, 14% of the remainder were over 0.01% and 3% were over 0.1%.

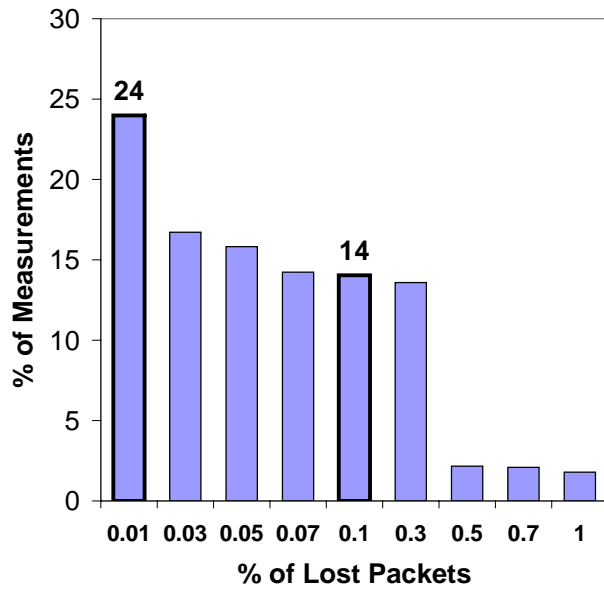


Figure 3.3: Loss Rates on TeraGrid

These numbers may look small in absolute terms, but they are sufficient to bring TCP/IP throughput crashing down on high-speed long-distance links. Conventional wisdom states that optical links do not drop packets; most carrier-grade optical equipment is designed to shut down beyond bit error rates of  $10^{-12}$  — one out of a trillion bits. However, the reliability of the lambda network is clearly not equal to the sum of its optical parts; in fact, it's less reliable by orders of magnitude. As a result, applications and protocols — such as TCP/IP — which expect extreme reliability from the high-speed network are instead subjected to unexpectedly high loss rates.

Of course, these numbers reflect the loss rate specifically experienced by UDP traffic on an end-to-end path and may not generalize to TCP packets. Also, we do not know if packets were dropped within the optical network or at intermediate devices within

either datacenter, though it's unlikely that they were dropped at the end-hosts; many of the measurements lost just one or two packets whereas kernel/NIC losses are known to be bursty [15]. Further, loss occurred on paths where levels of optical link utilization (determined by 20-second moving averages) were consistently lower than 20%, ruling out congestion as a possible cause, a conclusion supported by dialogue with the network administrators [89].

What are some possible causes for such high loss rates on TeraGrid? A likely hypothesis is *device clutter* — the critical communication path between nodes in different datacenters is littered with multiple electronic devices, each of which represents a potential point of failure. Another possibility is that such loss rates may be typical for any large-scale network where the cost of immediately detecting and fixing failures is prohibitively high. For example, we found through dialogue with the administrators that the steady loss rate experienced by the Indiana University site was due to a faulty line card, and the measurements showed that the error persisting over at least a three month period.

Other data-points for loss rates on high-speed long-haul networks are provided by the back-bone networks of Tier-1 ISPs. Global Crossing reports average loss rates between 0.01% and 0.03% on four of its six inter-regional long-haul links for the month of December 2007 [3]. Qwest reports loss rates of 0.01% and 0.02% in either direction on its trans-pacific link for the same month [8]. We expect privately managed lambdas to exhibit higher loss rates due to the inherent trade-off between fiber/equipment quality and cost [29], as well as the difficulty of performing routine maintenance on long-distance links. Consequently, we model end-to-end paths as dropping packets at rates of 0.01% to 1%, to capture a wide range of deployed networks.

## 3.2 Existing Reliability Options

TCP/IP is the default reliable communication option for contemporary networked applications, with deep, exclusive embeddings in commodity operating systems and networking APIs. Consequently, most applications requiring reliable communication over any form of network use TCP/IP.

As noted before, TCP/IP has three major problems when used over high-speed long-distance networks:

**1. Throughput Collapse in Lossy Networks:** TCP/IP is unable to distinguish between ephemeral loss modes — due to transient congestion, switching errors, or bad fiber — and persistent congestion. The loss of one packet out of ten thousand is sufficient to reduce TCP/IP throughput to a third of its lossless maximum; if one packet is lost out of a thousand, throughput collapses to a thirtieth of the maximum.

The root cause of throughput collapse is TCP/IP's fundamental reliance on loss as a signal of congestion. While recent approaches have sought to replace loss with delay as a congestion signal [90], or to specifically identify loss caused by non-congestion causes [70], older variants — prominently Reno — remain ubiquitously deployed.

**2. Recovery Delays for Real-Time Applications:** Conventional TCP/IP uses positive acknowledgments and retransmissions to ensure reliability — the sender buffers packets until their receipt is acknowledged by the receiver, and resends if an acknowledgment is not received within some time period. Hence, a lost packet is received in the form of a retransmission that arrives no earlier than 1.5 RTTs after the original send event. The sender has to buffer each packet until it's acknowledged, which takes 1 RTT in lossless operation, and it has to perform additional work to retransmit the packet if it does not

receive the acknowledgment. Also, any packets that arrive with higher sequence numbers than that of a lost packet must be queued while the receiver waits for the lost packet to arrive.

Consider a high-throughput financial banking application running in a datacenter in New York City, sending updates to a sister site in Switzerland. The RTT value between these two centers is typically 100 milliseconds; i.e., in the case of a lost packet, all packets received within the 150 milliseconds between the original packet send and the receipt of its retransmission have to be buffered at the receiver.

Notice that for this commonplace scenario, the loss of a single packet stops all traffic in the channel to the application for a seventh of a second; a sequence of such blocks can have devastating effect on a high-throughput system where every spare cycle counts. Further, in applications with many fine-grained components, a lost packet can potentially trigger a butterfly effect of missed deadlines along a distributed workflow. During high-activity periods — market crashes at stock exchanges, Christmas sales at online stores, winter storms at air-traffic control centers — overloaded networks and end-hosts can exhibit continuous packet loss, with each lost packet driving the system further and further out of sync with respect to its real-world deadlines.

**3. Massive Buffering Needs for High Throughput Applications:** TCP/IP uses fixed size buffers at receivers to prevent overflows; the sender never pushes more unacknowledged data into the network than the receiver is capable of holding. In other words, the size of the fluctuating window at the sender is bounded by the size of the buffer at the receiver. In high-speed long-distance networks, the quantity of in-flight unacknowledged data has to be extremely high for the flow to saturate the network. Since the size of the receiver window limits the sending envelope, it plays a major role in determining TCP/IP's throughput.

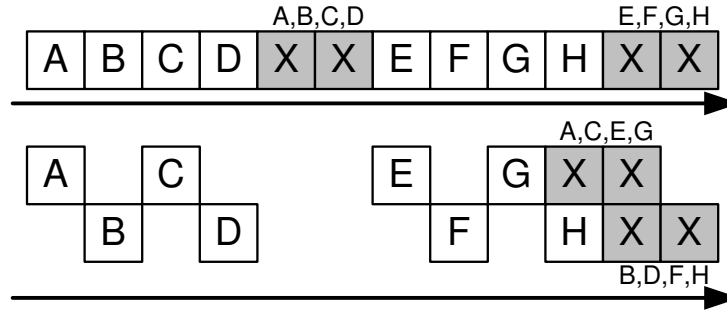


Figure 3.4: Interleaving with index 2: separate encoding for odd and even packets

The default receiver buffer sizes in many standard TCP/IP implementations are in the range of tens of kilobytes, and consequently inadequate receiver buffering is the first hurdle faced by most practical deployments. A natural solution is to increase the size of the receiver buffers; however, in many cases the receiving end-host may not have the spare memory capacity to buffer the entire bandwidth-delay product of the long-distance network. The need for larger buffers is orthogonal to the flow control mechanisms used within TCP/IP and impacts all variants equally.

### 3.2.1 The Case For (and Against) FEC

FEC encoders are typically parameterized with an  $(r, c)$  tuple — for each outgoing sequence of  $r$  data packets, a total of  $r + c$  data and error correction packets are sent over the channel. Significantly, redundancy information cannot be generated and sent until all  $r$  data packets are available for sending. Consequently, the latency of packet recovery is determined by the rate at which the sender transmits data. Generating error correction packets from less than  $r$  data packets at the sender is not a viable option — even though the data rate in this channel is low, the receiver and/or network could be operating at near full capacity with data from other senders.

FEC is also very susceptible to bursty losses [69]. *Interleaving* [67] is a standard

encoding technique used to combat bursty loss, where error correction packets are generated from alternate disjoint sub-streams of data rather than from consecutive packets. For example, with an interleave index of 3, the encoder would create correction packets separately from three disjoint sub-streams: the first containing data packets numbered  $(0, 3, 6 \dots (r-1) * 3)$ , the second with data packets numbered  $(1, 4, 7 \dots (r-1) * 3 + 1)$ , and the third with data packets numbered  $(2, 5, 8, \dots (r-1) * 3 + 2)$ . Interleaving adds burst tolerance to FEC but exacerbates its sensitivity to sending rate — with an interleave index of  $i$  and an encoding rate of  $(r, c)$ , the sender would have to wait for  $i * (r - 1) + 1$  packets before sending any redundancy information.

These two obstacles to using FEC in time-sensitive settings — rate sensitivity and burst susceptibility — are interlinked through the tuning knobs: an interleave of  $i$  and a rate of  $(r, c)$  provides tolerance to a burst of up to  $c * i$  consecutive packets. Consequently, the burst tolerance of an FEC code can be changed by modulating either the  $c$  or the  $i$  parameters. Increasing  $c$  enhances burst tolerance at the cost of network and encoding overhead, potentially worsening the packet loss experienced and reducing throughput. In contrast, increasing  $i$  trades off recovery latency for better burst tolerance without adding overhead — as mentioned, for higher values of  $i$ , the encoder has to wait for more data packets to be transmitted before it can send error correction packets.

Importantly, once the FEC encoding is parameterized with a rate and an interleave to tolerate a certain burst length  $B$  (for example,  $r = 5$ ,  $c = 2$  and  $i = 10$  to tolerate a burst of length  $2 * 10 = 20$ ), all losses occurring in bursts of size less than or equal to  $B$  are recovered with the same latency — and this latency depends on the  $i$  parameter. Ideally, we'd like to parameterize the encoding to tolerate a maximum burst length and then have recovery latency depend on the actual burstiness of the loss. At the same time, we would like the encoding to have a constant rate for network provisioning and

stability. Accordingly, an FEC scheme is required where latency of recovery degrades gracefully as losses get burstier, even as the encoding overhead stays constant.

### **3.3 Maelstrom Design and Implementation**

We describe the Maelstrom appliance as a single machine — later, we will show how more machines can be added to the appliance to balance encoding load and scale to multiple gigabits per second of traffic.

#### **3.3.1 Basic Mechanism**

The basic operation of Maelstrom is shown in Figure 3.5 — at the send-side datacenter, it snoops outgoing data packets en route to the destination datacenter, generating and injecting FEC repair packets into the stream in their wake. A repair packet consists of a ‘recipe’ list of data packet identifiers and FEC information generated from these packets; in the example in Figure 3.5, this information is a simple XOR. The size of the XOR is equal to the MTU of the datacenter network, and to avoid fragmentation of repair packets we require that the MTU of the long-haul network be set to a slightly larger value. This requirement is usually satisfied in practical deployments, since gigabit links very often use ‘Jumbo’ frames of up to 9000 bytes [48] while LAN networks have standard MTUs of 1500 bytes.

At the receiving datacenter, the appliance examines incoming repair packets and uses them to recover missing data packets. On recovery, the data packet is injected transparently into the stream to the receiving end-host. Recovered data packets will typically arrive out-of-order, but this behavior is expected by communication stacks

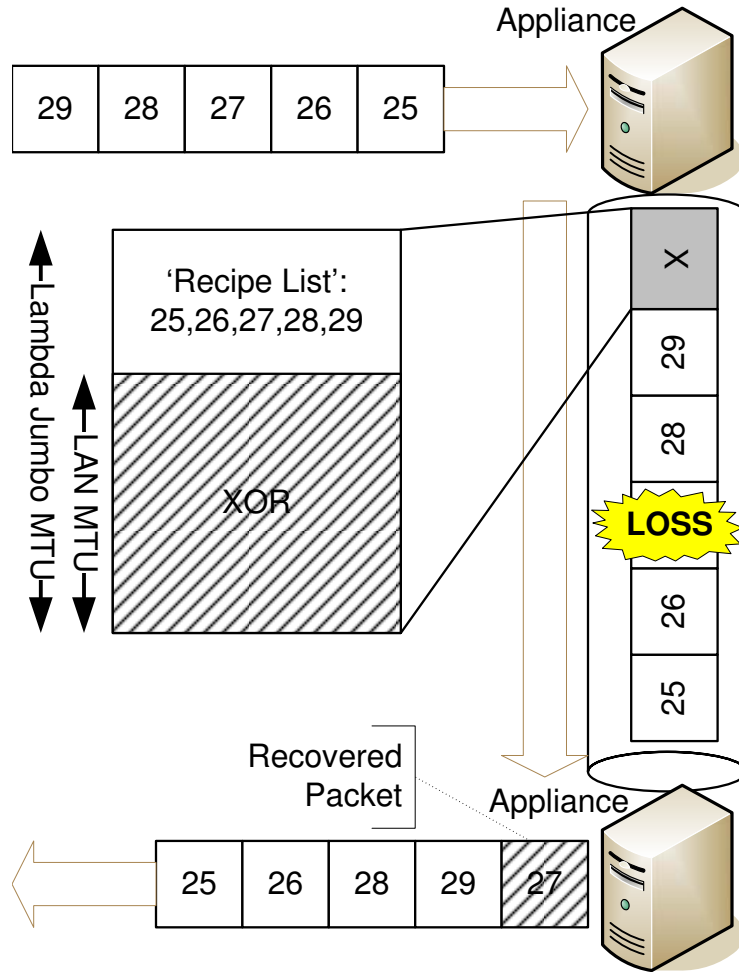


Figure 3.5: Basic Maelstrom mechanism: repair packets are injected into stream transparently

designed for the commodity Internet.

### 3.3.2 Flow Control

While relaying TCP/IP data, Maelstrom has two flow control modes: *end-to-end* and *split*. Figure 3.6 illustrates these two modes.

**End-to-end Mode:** With end-to-end flow control, the appliance treats TCP/IP packets as conventional IP packets and routes them through without modification, allowing



A) End-to-End Flow Control



B) Split Flow Control

Figure 3.6: Flow Control Options in Maelstrom

flow-control to proceed between the end-hosts. Importantly, TCP/IP’s semantics are not modified; when the sending end-host receives an acknowledgment, it can assume that the receiving end-host successfully received the message. In end-to-end mode, Maelstrom functions as a passive device, snooping outgoing and incoming traffic at the datacenter’s edge — its failure does not disrupt the flow of packets between the two datacenters.

**Split Mode:** In *split* mode, the send-side appliance acts as a TCP/IP endpoint, terminating connections and sending back ACKs immediately before relaying data on appliance-to-appliance flows. Split mode is extremely useful when end-hosts have limited buffering capacity, since it allows the receive-side appliance to buffer incoming data over the high-speed long-distance link. It also mitigates TCP/IP’s slow-start effects for short-lived flows. In split mode, Maelstrom has to operate as an active device, inserted into the critical communication path — its failure disconnects the communication path between the two datacenters.

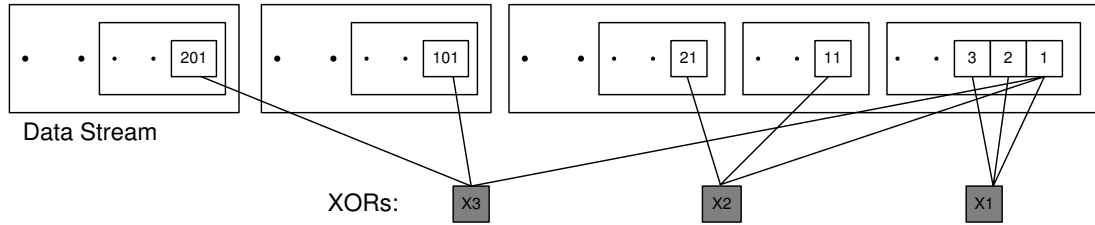


Figure 3.7: Layered Interleaving:  $(r = 3, c = 3), I = (1, 10, 100)$

**Is Maelstrom TCP-Friendly?** While Maelstrom respects end-to-end flow control connections (or splits them and implements its own proxy-to-proxy flow control as described above), it is not designed for routinely congested networks; the addition of FEC under TCP/IP flow control allows it to steal bandwidth from other competing flows running without FEC in the link, though maintaining fairness versus similarly FEC-enhanced flows [61]. However, friendliness with conventional TCP/IP flows is not a primary protocol design goal on over-provisioned multi-gigabit links, which are often dedicated to specific high-value applications. We see evidence for this assertion in the routine use of parallel flows [81] and UDP ‘blast’ protocols [46, 88] both in commercial deployments and by researchers seeking to transfer large amounts of data over high-capacity academic networks.

### 3.3.3 Layered Interleaving

In layered interleaving, an FEC protocol with rate  $(r, c)$  is produced by running  $c$  multiple instances of an  $(r, 1)$  FEC protocol simultaneously with increasing interleave indices  $I = (i_0, i_1, i_2, \dots, i_{c-1})$ . For example, if  $r = 8, c = 3$  and  $I = (i_0 = 1, i_1 = 10, i_2 = 100)$ , three instances of an  $(8, 1)$  protocol are executed: the first instance with interleave  $i_0 = 1$ , the second with interleave  $i_1 = 10$  and the third with interleave  $i_2 = 100$ . An  $(r, 1)$  FEC encoding is simply an XOR of the  $r$  data packets — hence, in layered interleaving each data packet is included in  $c$  XORs, each of which is generated at different interleaves

from the original data stream. Choosing interleaves appropriately (as we shall describe shortly) ensures that the  $c$  XORs containing a data packet do not have any other data packet in common. The resulting protocol effectively has a rate of  $(r, c)$ , with each XOR generated from  $r$  data packets and each data packet included in  $c$  XORs. Figure 3.7 illustrates layered interleaving for a  $(r = 3, c = 3)$  encoding with  $I = (1, 10, 100)$ .

As mentioned previously, standard FEC schemes can be made resistant to a certain loss burst length at the cost of increased recovery latency for all lost packets, including smaller bursts and singleton drops. In contrast, layered interleaving provides graceful degradation in the face of bursty loss for constant encoding overhead — singleton random losses are recovered as quickly as possible, by XORs generated with an interleave of 1, and each successive layer of XORs generated at a higher interleave catches larger bursts missed by the previous layer.

The implementation of this algorithm is simple and shown in Figure 3.8 — at the send-side proxy, a set of repair bins is maintained for each layer, with  $i$  bins for a layer with interleave  $i$ . A repair bin consists of a partially constructed repair packet: an XOR and the ‘recipe’ list of identifiers of data packets that compose the XOR. Each intercepted data packet is added to each layer — where adding to a layer simply means choosing a repair bin from the layer’s set, incrementally updating the XOR with the new data packet, and adding the data packet’s header to the recipe list. A counter is incremented as each data packet arrives at the appliance, and choosing the repair bin from the layer’s set is done by taking the modulo of the counter with the number of bins in each layer: for a layer with interleave 10, the  $x$ th intercepted packet is added to the  $(x \bmod 10)$ th bin. When a repair bin fills up — its recipe list contains  $r$  data packets — it ‘fires’: a repair packet is generated consisting of the XOR and the recipe list and is scheduled for sending, while the repair bin is re-initialized with an empty recipe list and

blank XOR.

At the receive-side proxy, incoming repair packets are processed as follows: if all the data packets contained in the repair’s recipe list have been received successfully, the repair packet is discarded. If the repair’s recipe list contains a single missing data packet, recovery can occur immediately by combining the XOR in the repair with the other successfully received data packets. If the repair contains multiple missing data packets, it cannot be used immediately for recovery — it is instead stored in a table that maps missing data packets to repair packets. Whenever a data packet is subsequently received or recovered, this table is checked to see if any XORs now have singleton losses due to the presence of the new packet and can be used for recovering other missing packets.

Importantly, XORs received from different layers interact to recover missing data packets, since an XOR received at a higher interleave can recover a packet that makes an earlier XOR at a lower interleave usable — hence, though layered interleaving is equivalent to  $c$  different  $(r, 1)$  instances in terms of overhead and design, its recovery power is much higher and comes close to standard  $(r, c)$  algorithms.

## Optimizations

**Staggered Start for Rate-Limiting** In the naive implementation of the layered interleaving algorithm, repair packets are transmitted as soon as repair bins fill and allow them to be constructed. Also, all the repair bins in a layer fill in quick succession; in Figure 3.8, the arrival of packets 36, 37, 38 and 39 will successively fill the four repair bins in layer 2. This behavior leads to a large number of repair packets being generated and sent within a short period of time, which results in undesirable overhead and traffic spikes; ideally, we would like to rate-limit transmissions of repair packets to one for

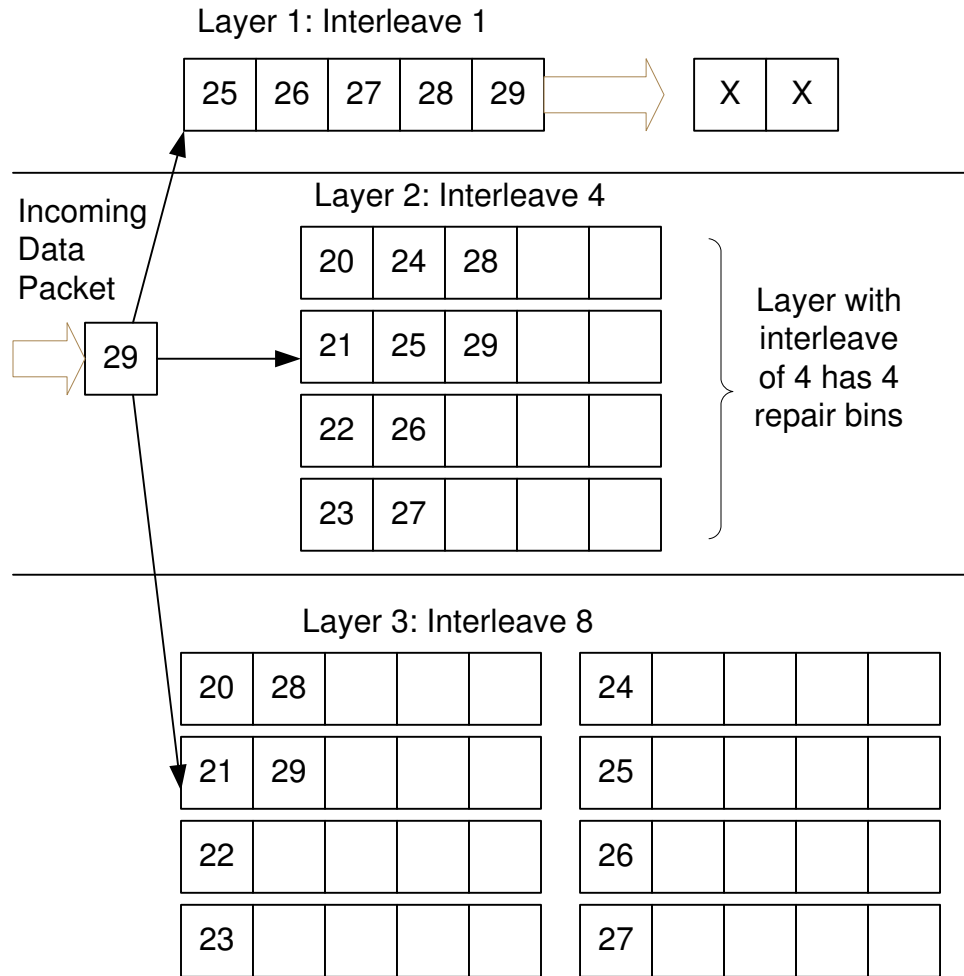


Figure 3.8: Layered Interleaving Implementation: ( $r = 5, c = 3$ ),  $I = (1, 4, 8)$

every  $r$  data packets.

This problem is fixed by ‘staggering’ the starting sizes of the bins, analogous to the starting positions of runners in a sprint; the very first time bin number  $x$  in a layer of interleave  $i$  fires, it does so at size  $x \bmod r$ . For example, in Figure 3.8, the first repair bin in the second layer with interleave 4 would fire at size 1, the second would fire at size 2, and so on. Hence, for the first  $i$  data packets added to a layer with interleave  $i$ , exactly  $i/r$  fire immediately with just one packet in them; for the next  $i$  data packets added, exactly  $i/r$  fire immediately with two data packets in them, and so on until  $r * i$  data packets have been added to the layer and all bins have fired exactly once. Subsequently, all bins

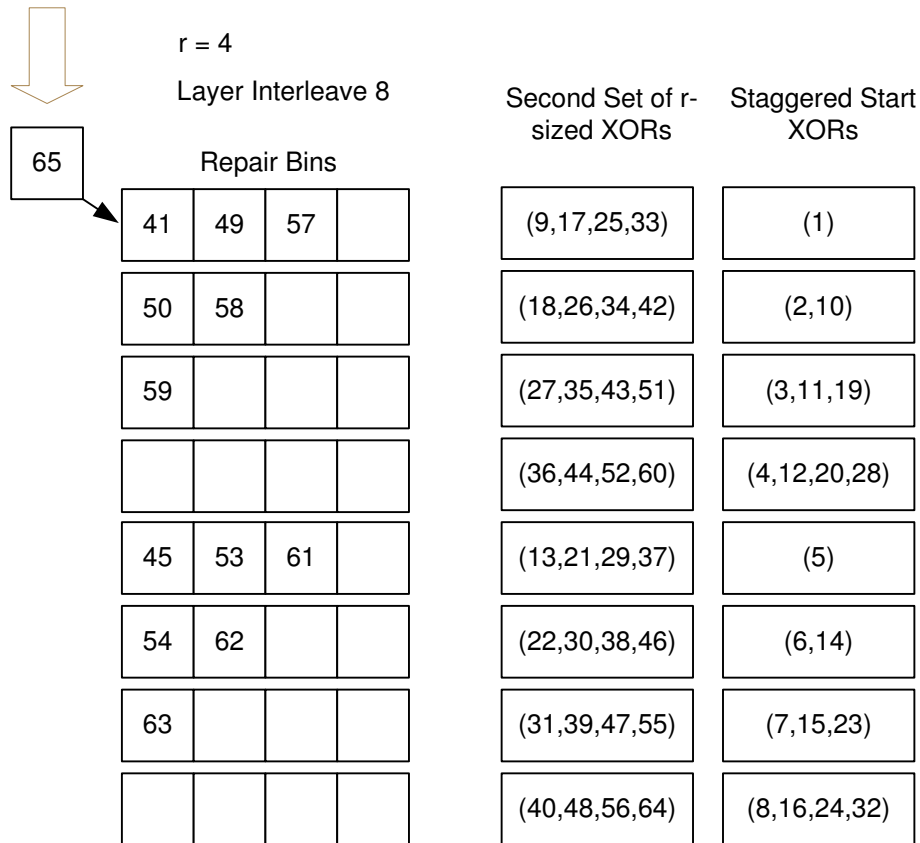


Figure 3.9: Staggered Start

fire at size  $r$ ; however, now that they have been staggered at the start, only  $i/r$  fire for any  $i$  data packets. The outlined scheme works when  $i$  is greater than or equal to  $r$ , as is usually the case. If  $i$  is smaller than  $r$ , the bin with index  $x$  fires at  $((x \bmod r) * r/i)$  — hence, for  $r = 4$  and  $i = 2$ , the initial firing sizes would be 2 for the first bin and 4 for the second bin. If  $r$  and  $i$  are not integral multiples of each other, the rate-limiting still works but is slightly less effective due to rounding errors.

**Delaying XORs** In the straightforward implementation, repair packets are transmitted as soon as they are generated. This results in the repair packet leaving immediately after the last data packet that was added to it, which lowers burst tolerance — if the repair packet was generated at interleave  $i$ , the resulting protocol can tolerate a burst of  $i$  lost data packets excluding the repair, but the burst could swallow both the repair and the

last data packet in it as they are not separated by the requisite interleave. The solution to this is simple — delay sending the repair packet generated by a repair bin until the next time a data packet is added to the now empty bin, which happens  $i$  packets later and introduces the required interleave between the repair packet and the last data packet included in it.

Notice that although transmitting the XOR immediately results in faster recovery, doing so also reduces the probability of a lost packet being recovered. This trade-off results in a minor control knob permitting us to balance speed against burst tolerance; our default configuration is to transmit the XOR immediately.

### 3.3.4 Back-of-the-Envelope Analysis

To start with, we note that no two repair packets generated at different interleaves  $i_1$  and  $i_2$  (such that  $i_1 < i_2$ ) will have more than one data packet in common as long as the Least Common Multiple (*LCM*) of the interleaves is greater than  $r * i_1$ ; pairings of repair bins in two different layers with interleaves  $i_1$  and  $i_2$  occur every  $LCM(i_1, i_2)$  packets. Thus, a good rule of thumb is to select interleaves that are relatively prime to maximize their *LCM*, and also ensure that the larger interleave is greater than  $r$ .

Let us assume that packets are dropped with uniform, independent probability  $p$ . Given a lost data packet, what is the probability that we can recover it? We can recover a data packet if at least one of the  $c$  XORs containing it is received correctly and ‘usable’, i.e., all the other data packets in it have also been received correctly, the probability of which is simply  $(1 - p)^{r-1}$ . The probability of a received XOR being unusable is the complement:  $(1 - (1 - p)^{r-1})$ .

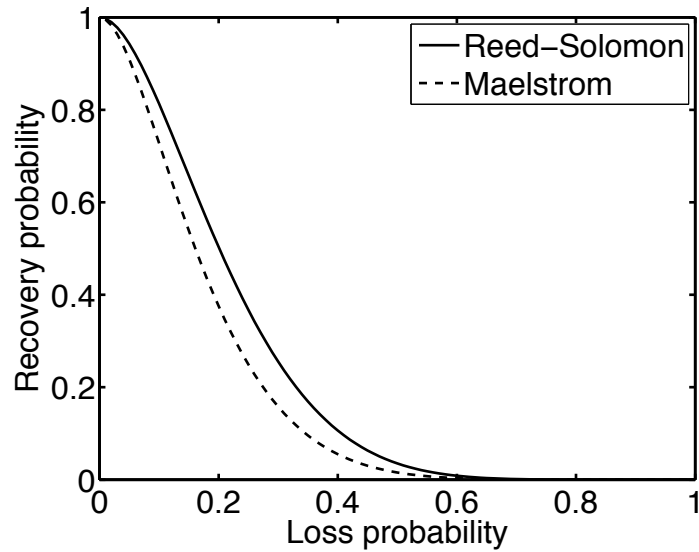


Figure 3.10: Comparison of Packet Recovery Probability:  $r=7, c=2$

Consequently, the probability  $x$  of a sent XOR being dropped or unusable is the sum of the probability that it was dropped and the probability that it was received and unusable:  $x = p + (1 - p)(1 - (1 - p)^{r-1}) = (1 - (1 - p)^r)$ .

Since it is easy to ensure that no two XORs share more than one data packet, the usability probabilities of different XORs are independent. The probability of all the  $c$  XORs being dropped or unusable is  $x^c$ ; hence, the probability of correctly receiving at least one usable XOR is  $1 - x^c$ . Consequently, the probability of recovering the lost data packet is  $1 - x^c$ , which expands to  $1 - (1 - (1 - p)^r)^c$ .

This closed-form formula only gives us a lower bound on the recovery probability, since the XOR usability formula does not factor in the probability of the other data packets in the XOR being dropped and *recovered*.

Now, we extend the analysis to bursty losses. If the lost data packet was part of a loss burst of size  $b$ , repair packets generated at interleaves less than  $b$  are dropped or useless with high probability, and we can discount them. The probability of recovering

the data packet is then  $1 - x^{c'}$ , where  $c'$  is the number of XORs generated at interleaves greater than  $b$ . The formulae derived for XOR usability still hold, since packet losses with more than  $b$  intervening packets between them have independent probability; there is only correlation within the bursts, not between bursts.

How does this compare to traditional  $(r, c)$  codes such as Reed-Solomon [92]? In Reed-Solomon,  $c$  repair packets are generated and sent for every  $r$  data packets, and the correct delivery of any  $r$  of the  $r + c$  packets transmitted is sufficient to reconstruct the original  $r$  data packets. Hence, given a lost data packet, we can recover it if at least  $r$  packets are received correctly in the encoding set of  $r + c$  data and repair packets that the lost packet belongs to. Thus, the probability of recovering a lost packet is equivalent to the probability of losing  $c - 1$  or less packets from the total  $r + c$  packets. Since the number of other lost packets in the XOR is a random variable  $Y$  and has a binomial distribution with parameters  $(r + c - 1)$  and  $p$ , the probability  $P(Y \leq c - 1)$  is the summation  $\sum_{z \leq c-1} P(Y = z)$ . In Figure 3.10, we plot the recovery probability curves for Layered Interleaving and Reed-Solomon against uniformly random loss rate, for  $(r = 7, c = 2)$  — note that the curves are very close to each other, especially in the loss range of interest between 0% and 10%.

### 3.3.5 Local Recovery for Receiver Loss

In the absence of intelligent flow control mechanisms like TCP/IP's receiver-window advertisements, inexpensive datacenter end-hosts can be easily overwhelmed and drop packets during traffic spikes or CPU-intensive maintenance tasks like garbage collection. Reliable application-level protocols layered over UDP — for reliable multicast [15] or high speed data transfer [46], for example — would ordinarily go back to the sender to

retrieve the lost packet, even though it was dropped at the receiver after covering the entire geographical distance.

The Maelstrom proxy acts as a local packet cache, storing incoming packets for a short period of time and providing hooks that allow protocols to first query the cache to locate missing packets before sending retransmission requests back to the sender. Future versions of Maelstrom could potentially use knowledge of protocol internals to transparently intervene; for example, by intercepting and satisfying retransmission requests sent by the receiver in a NAK-based protocol, or by resending packets when acknowledgments are not observed within a certain time period in an ACK-based protocol.

### **3.3.6 Implementation Details**

We initially implemented and evaluated Maelstrom as a user-space proxy. Performance turned out to be limited by copying and context-switching overheads, and we subsequently reimplemented the system as a module that runs within the Linux 2.6.20 kernel. At an encoding rate of (8, 3), the experimental prototype of the kernel version reaches output speeds close to 1 gigabit per second of combined data and FEC traffic, limited only by the capacity of the outbound network card.

Of course, lambda networks are already reaching speeds of 40-100 gigabits, and higher speeds are a certainty down the road. To handle multi-gigabit loads, we envision Maelstrom as a small rack-style cluster of blade-servers, each acting as an individual proxy. Traffic would be distributed over such a rack by partitioning the address space of the remote datacenter and routing different segments of the space through distinct Maelstrom appliance pairs. In future work, we plan to experiment with such configurations, which would also permit us to explore fault-tolerance issues (if a Maelstrom

blade fails, for example), and to support load-balancing schemes that might vary the IP address space partitioning dynamically to spread the encoding load over multiple machines. For this paper, however, we present the implementation and performance of a single-machine appliance.

The kernel implementation is a module for Linux 2.6.20 with hooks into the kernel packet filter [6]. Maelstrom proxies work in pairs, one on each side of the long haul link. Each proxy acts both as an ingress and egress router at the same time since they handle duplex traffic in the following manner:

- The egress router captures IP packets and creates redundant FEC packets. The original IP packets are routed through unaltered as they would have been originally; the redundant packets are then forwarded to the remote ingress router via a UDP channel.
- The ingress router captures and stores IP packets coming from the direction of the egress router. Upon receipt of a redundant packet, an IP packet is recovered if there is an opportunity to do so. Redundant packets that can be used at a later time are stored. If the redundant packet is useless it is immediately discarded. Upon recovery the IP packet is sent through a raw socket to its intended destination.

Using FEC requires that each data packet have a unique identifier that the receiver can use to keep track of received data packets and to identify missing data packets in a repair packet. If we had access to end-host stacks, we could have added a header to each packet with a unique sequence number [75]; however, we intercept traffic transparently and need to route it without modification or addition, for performance reasons. Consequently, we identify IP packets by a tuple consisting of the source and destination IP address, IP identification field, size of the IP header plus data, and a checksum over the

IP data payload. The checksum over the payload is necessary since the IP identification field is only 16 bits long and a single pair of end-hosts communicating at high speeds will use the same identifier for different data packets within a fairly short interval unless the checksum is added to differentiate between them. Note that non-unique identifiers result in garbled recovery by Maelstrom, an event which will be caught by higher level checksums designed to deal with transmission errors on commodity networks and hence does not have significant consequences unless it occurs frequently.

The kernel version of Maelstrom can generate up to a Gigabit per second of data and FEC traffic, with the input data rate depending on the encoding rate. In our experiments, we were able to saturate the outgoing card at rates as high as (8, 4), with CPU overload occurring at (8, 5) where each incoming data packet had to be XORed 5 times.

### **3.3.7 Buffering Requirements**

At the receive-side proxy, incoming data packets are buffered so that they can be used in conjunction with XORs to recover missing data packets. Also, any received XOR that is missing more than one data packet is stored temporarily, in case all but one of the missing packets are received later or recovered through other XORs, allowing the recovery of the remaining missing packet from this XOR. In practice we stored data and XOR packets in double buffered red black trees — for 1500 byte packets and 1024 entries this occupies around 3 MB of memory.

At the send-side, the repair bins in the layered interleaving scheme store incrementally computed XORs and lists of data packet headers, without the data packet payloads, resulting in low storage overheads for each layer that rise linearly with the value of the interleave. The memory footprint for a long-running proxy was around 10 MB in our

experiments.

### **3.3.8 Other Performance Enhancing Roles**

Maelstrom appliances can optionally aggregate small sub-kilobyte packets from different flows into larger ones for better communication efficiency over the long-distance link. Additionally, in split flow control mode they can perform send-side buffering of in-flight data for multi-gigabyte flows that exceed the sending end-host's buffering capacity. Also, Maelstrom appliances can act as multicast forwarding nodes: appliances send multicast packets to each other across the long-distance link, and use IP Multicast [30] to spread them within their datacenters. Lastly, appliances can take on other existing roles in the datacenter, acting as security and VPN gateways and as conventional performance enhancing proxies (PEPs) [21].

## **3.4 Evaluation**

We evaluated Maelstrom on the Emulab testbed at Utah [91]. For all the experiments, we used a ‘dumbbell’ topology of two clusters of nodes connected via routing nodes with a high-latency link in between them, designed to emulate the setup in Figure 3.2, and ran the proxy code on the routers. Figures 3.12 and 3.13 show the performance of the kernel version at Gigabit speeds; the remainder of the graphs show the performance of the user-space version at slower speeds. To emulate the MTU difference between the long-haul link and the datacenter network (see Section 3.3.1) we set an MTU of 1200 bytes on the network connecting the end-hosts to the proxy and an MTU of 1500 bytes on the long-haul link between proxies; the only exception is Figure 3.12, where we

maintained equal MTUs of 1500 bytes on both links. Further, all the experiments are done with Maelstrom using end-to-end flow control (see Figure 3.6), except for 3.13, which illustrates the performance of split mode flow control.

### 3.4.1 Throughput Metrics

Figures 3.11 and 3.12 show that commodity TCP/IP throughput collapses in the presence of non-congestion loss, and that Maelstrom successfully masks loss and prevents this collapse from occurring. Figure 3.11 shows the performance of the user-space version on a 100 Mbps link and Figure 3.12 shows the kernel version on a 1 Gbps link. The experiment in each case involves running iperf [84] flows from one node to another across the long-distance link with and without intermediary Maelstrom proxies and measuring obtained throughput while varying loss rate (top graph on each figure) and one-way link latency (bottom graph). The error bars on the graphs on top are standard errors of the throughput over ten runs; between each run, we flush TCP/IP's cache of tuning parameters to allow for repeatable results. The clients in the experiment are running TCP/IP Reno on a Linux 2.6.20 that performs autotuning. The Maelstrom parameters used are  $r = 8, c = 3, I = (1, 20, 40)$ .

The user-space version involved running a single 10 second iperf flow from one node to another with and without Maelstrom running on the routers and measuring throughput while varying the random loss rate on the link and the one-way latency. To test the kernel version at gigabit speeds, we ran eight parallel iperf flows from one node to another for 120 seconds. The curves obtained from the two versions are almost identical; we present both to show that the kernel version successfully scales up the performance of the user-space version to hundreds of megabits of traffic per second.

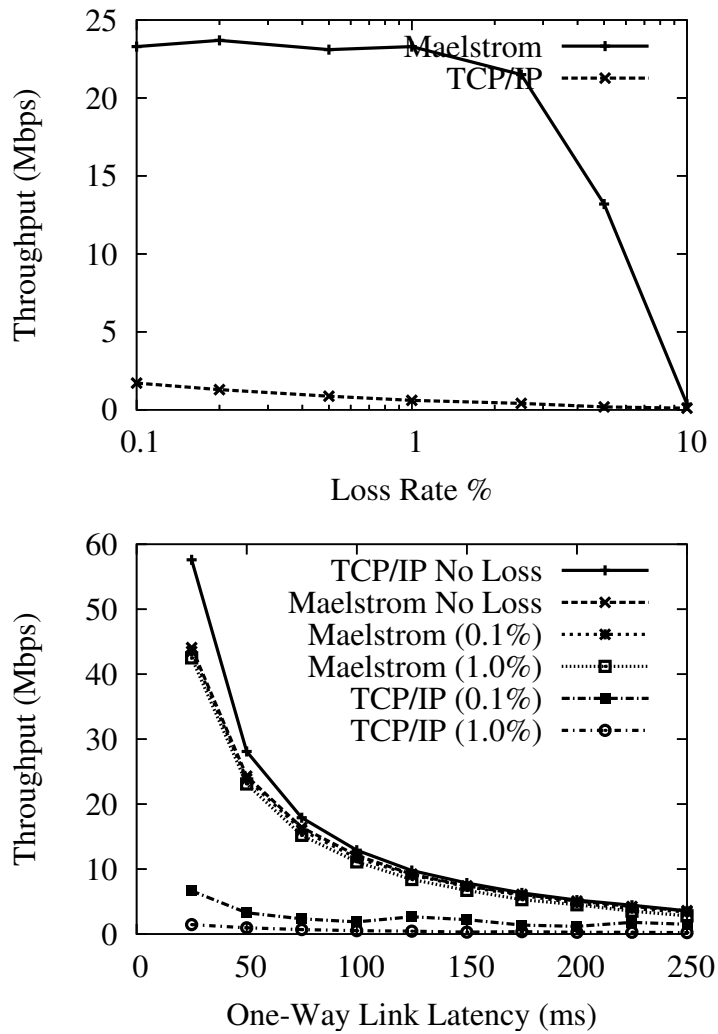


Figure 3.11: User-Space Throughput against Loss Rate (Top) and One-Way Link Latency (Bottom)

In Figures 3.11 (Top) and 3.12 (Top), we show how TCP/IP performance degrades on a 50ms link as the loss rate is increased from 0.01% to 10%. Maelstrom masks loss up to 2% without significant throughput degradation, with the kernel version achieving two orders of magnitude higher throughput than conventional TCP/IP at 1% loss.

The graphs on the bottom side of Figures 3.11 and 3.12 show TCP/IP throughput declining on a link of increasing length when subjected to uniform loss rates of 0.1% and 1%. The top line in the graphs is the performance of TCP/IP without loss and

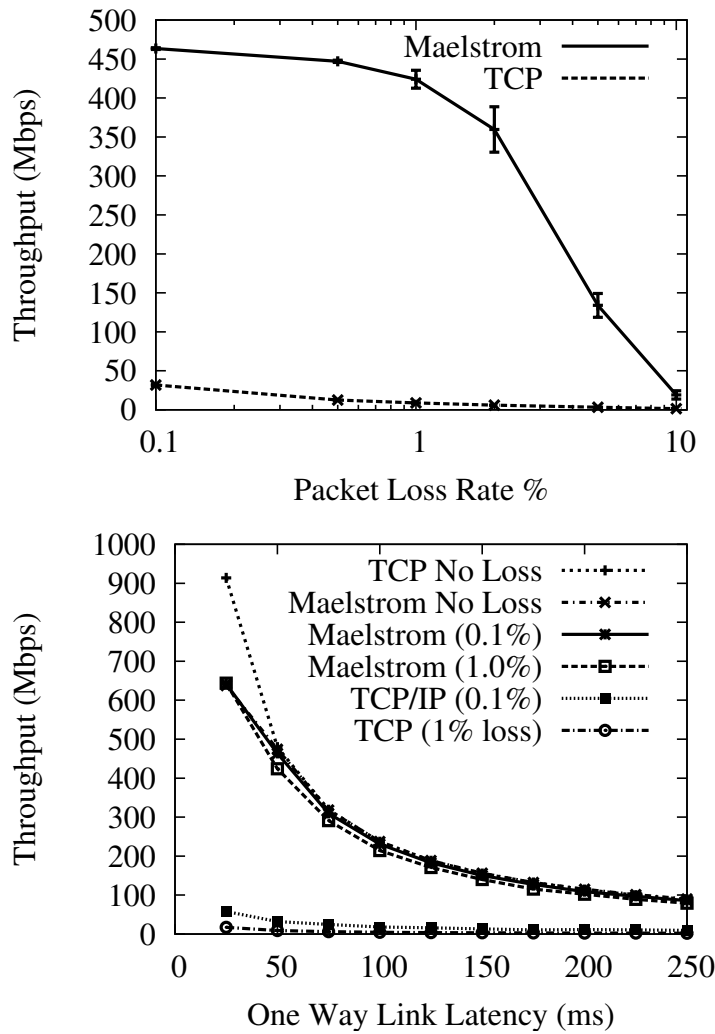


Figure 3.12: Kernel Throughput against Loss Rate (Top) and One-Way Link Latency (Bottom).

provides an upper bound for performance on the link. In both user-space and kernel versions, Maelstrom masks packet loss and tracks the lossless line closely, lagging only when the link latency is low and TCP/IP's throughput is very high.

To allow TCP/IP to attain very high speeds on the gigabit link, we had to set the MTU of the entire path to be the maximum 1500 bytes, which meant that the long-haul link had the same MTU as the inter-cluster link. This resulted in the fragmentation of repair packets sent over UDP on the long-haul link into two IP packet fragments. Since

the loss of a single fragment resulted in the loss of the repair, we observed a higher loss rate for repairs than for data packets. Consequently, we expect performance to be better on a network where the MTU of the long-haul link is truly larger than the MTU within each cluster.

Even with zero loss, TCP/IP throughput in Figure 3.12 (Bottom) declines with link latency; this is due to the cap on throughput placed by the buffering available at the receiving end-hosts. The preceding experiments were done with Maelstrom in end-to-end flow control mode, where it is oblivious to TCP/IP and does not split connections, and is consequently sensitive to the size of the receiver buffer. Figure 3.13 shows the performance of split mode flow control, where Maelstrom breaks a single TCP/IP connection into three hops (see Figure 3.6) and buffers data. As expected, split-mode flow control eliminates the requirement for large buffers at the receiving end-hosts. Throughput is essentially insensitive to one-way link latency, with a slight drop due to buffering overhead on the Maelstrom boxes.

### 3.4.2 Latency Metrics

To measure the latency effects of TCP/IP and Maelstrom, we ran a 0.1 Mbps stream between two nodes over a 100 Mbps link with 50 ms one-way latency, and simultaneously ran a 10 Mbps flow alongside on the same link to simulate a real-time stream combined with other inter-cluster traffic. Figure 3.14 (Top) shows the average delivery latency of 1KB application-level packets in the 0.1 Mbps stream, as loss rates go up.

Figure 3.14 (Bottom) shows the same scenario with a constant uniformly random loss rate of 0.1% and varying one-way latency. Maelstrom's delivery latency is almost exactly equal to the one-way latency on the link, whereas TCP/IP takes more than twice

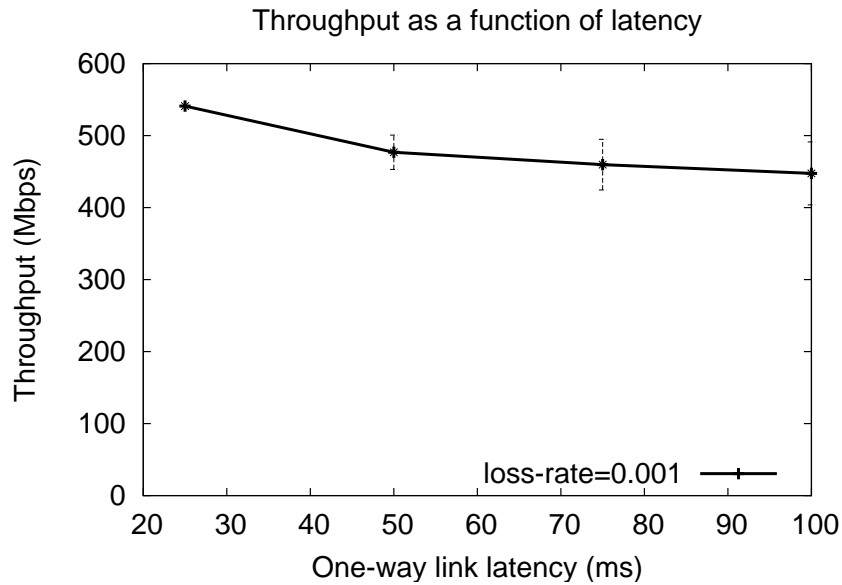


Figure 3.13: Throughput of Split-Mode Buffering Flow Control against One-Way Link Latency.

as long once one-way latencies go past 100 ms.

Figure 3.15 plots delivery latency against message identifier. A key point is that we are plotting the delivery latency of all packets, not just lost ones. The spikes in latency are triggered by losses that lead to packets piling up both at the receiver and the sender. TCP/IP delays correctly received packets at the receiver while waiting for missing packets sequenced earlier by the sender. It also delays packets at the sender when it cuts down on the sending window size in response to the loss events. The delays caused by these two mechanisms are illustrated in Figure 3.15, where single packet losses cause spikes in delivery latency that last for hundreds of packets. The Maelstrom configuration used is  $r = 7, c = 2, I = (1, 10)$ .

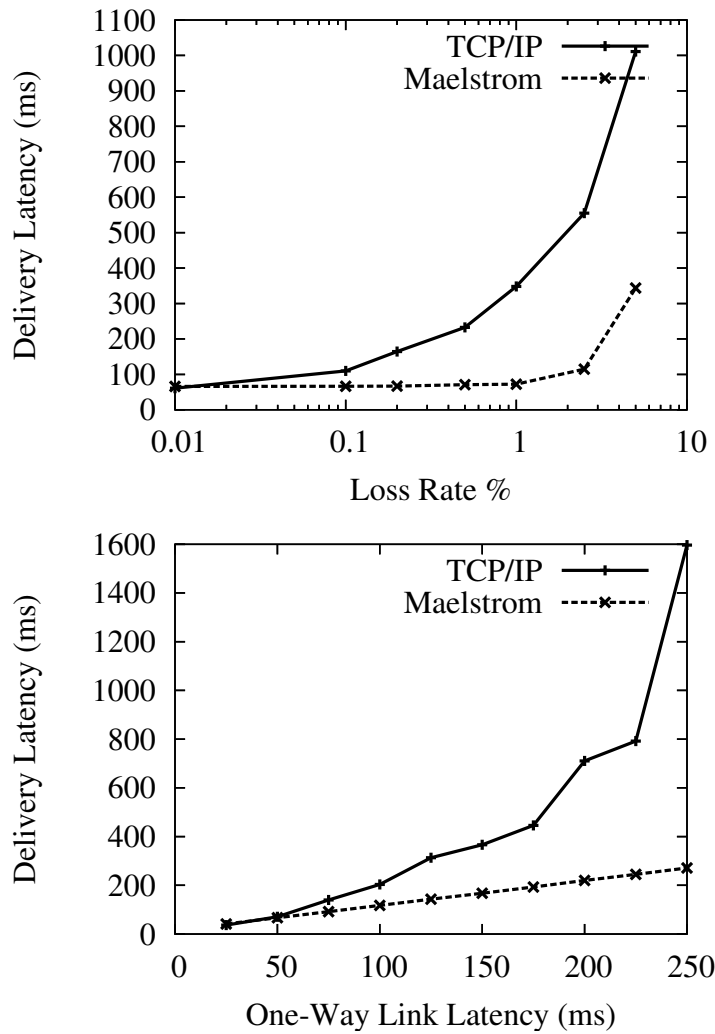


Figure 3.14: Per-Packet One-Way Delivery Latency against Loss Rate (Top) and Link Latency (Bottom)

### 3.4.3 Layered Interleaving and Bursty Loss

Thus far we have shown how Maelstrom effectively hides loss from TCP/IP for packets dropped with uniform randomness. Now, we examine the performance of the layered interleaving algorithm, showing how different parameterizations handle bursty loss patterns. We use a loss model where packets are dropped in bursts of fixed length, allowing us to study the impact of burst length on performance. The link has a one-way latency of 50 ms and a loss rate of 0.1% (except in Figure 3.16, where it is varied), and a 10

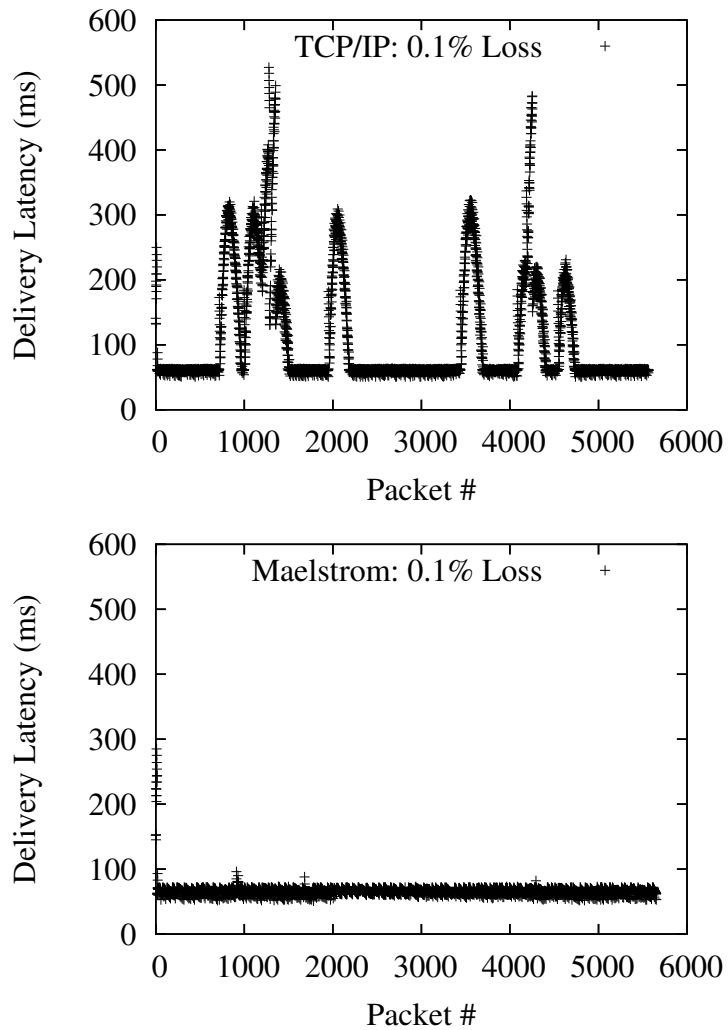


Figure 3.15: Packet delivery latencies

Mbps flow of udp packets is sent over it.

In Figure 3.16 we show that our observation in Section 3.3.4 is correct for high loss rates — if the interleaves are relatively prime, performance improves substantially when loss rates are high and losses are bursty. The graph plots the percentage of lost packets successfully recovered on the y-axis against an x-axis of loss rates on a log scale. The Maelstrom configuration used is  $r = 8, c = 3$  with interleaves of  $(1, 10, 20)$  and  $(1, 11, 19)$ .

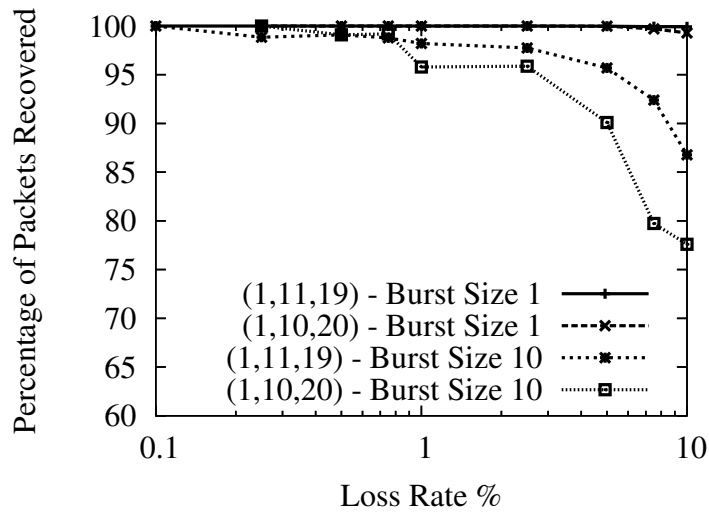


Figure 3.16: Relatively prime interleaves offer better performance

In Figure 3.17, we show the ability of layered interleaving to provide gracefully degrading performance in the face of bursty loss. On the top, we plot the percentage of lost packets successfully recovered against the length of loss bursts for two different sets of interleaves, and in the bottom graph we plot the average latency at which the packets were recovered. Recovery latency is defined as the difference between the eventual delivery time of the recovered packet and the one-way latency of the link (we confirmed that the Emulab link had almost no jitter on correctly delivered packets, making the one-way latency an accurate estimate of expected lossless delivery time). As expected, increasing the interleaves results in much higher recovery percentages at large burst sizes, but comes at the cost of higher recovery latency. For example, a (1, 19, 41) set of interleaves catches almost all packets in an extended burst of 25 packets at an average latency of around 45 milliseconds, while repairing all random singleton losses within 2-3 milliseconds. The graphs also show recovery latency rising gracefully with the increase in loss burst length: the longer the burst, the longer it takes to recover the lost packets. The Maelstrom configuration used is  $r = 8, c = 3$  with interleaves of (1, 11, 19) and (1, 19, 41).

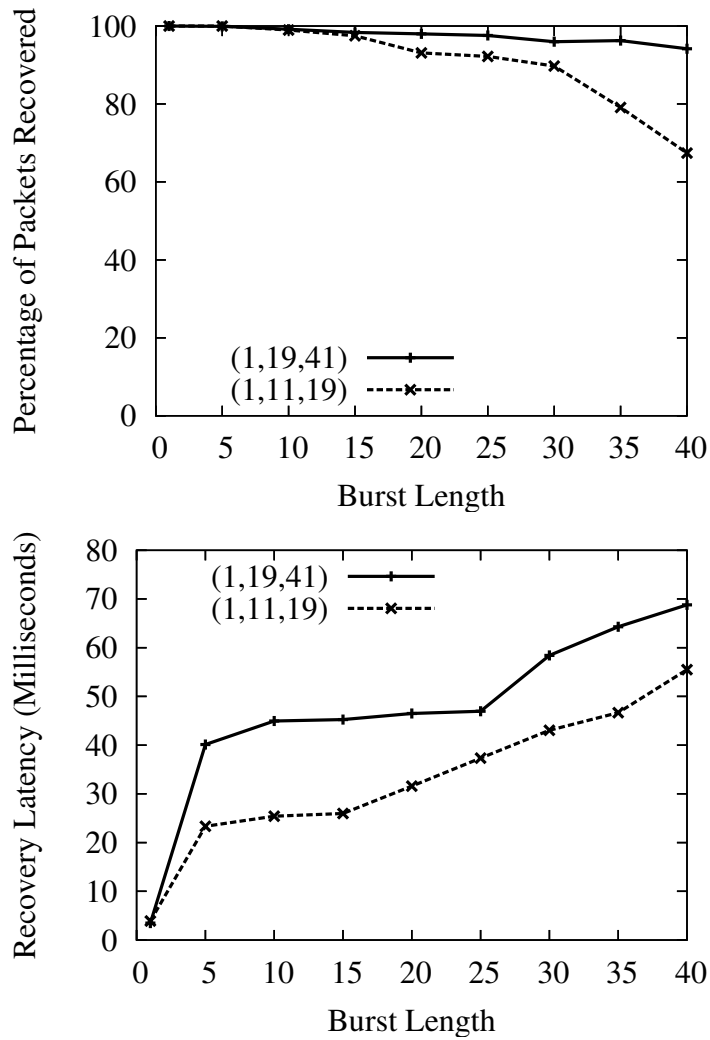


Figure 3.17: Layered Interleaving Recovery Percentage and Latency

Figure 3.18 shows histograms of recovery latencies for the two interleave configurations under different loss burst lengths. The histograms confirm the trends described above: packet recoveries take longer from top to bottom as we increase loss burst length, and from left to right as we increase the interleave values.

Figure 3.19 illustrates the difference between a traditional FEC code and layered interleaving by plotting a 50-element moving average of recovery latencies for both codes. The channel is configured to lose singleton packets randomly at a loss rate of 0.1% and additionally lose long bursts of 30 packets at occasional intervals. Both codes are config-

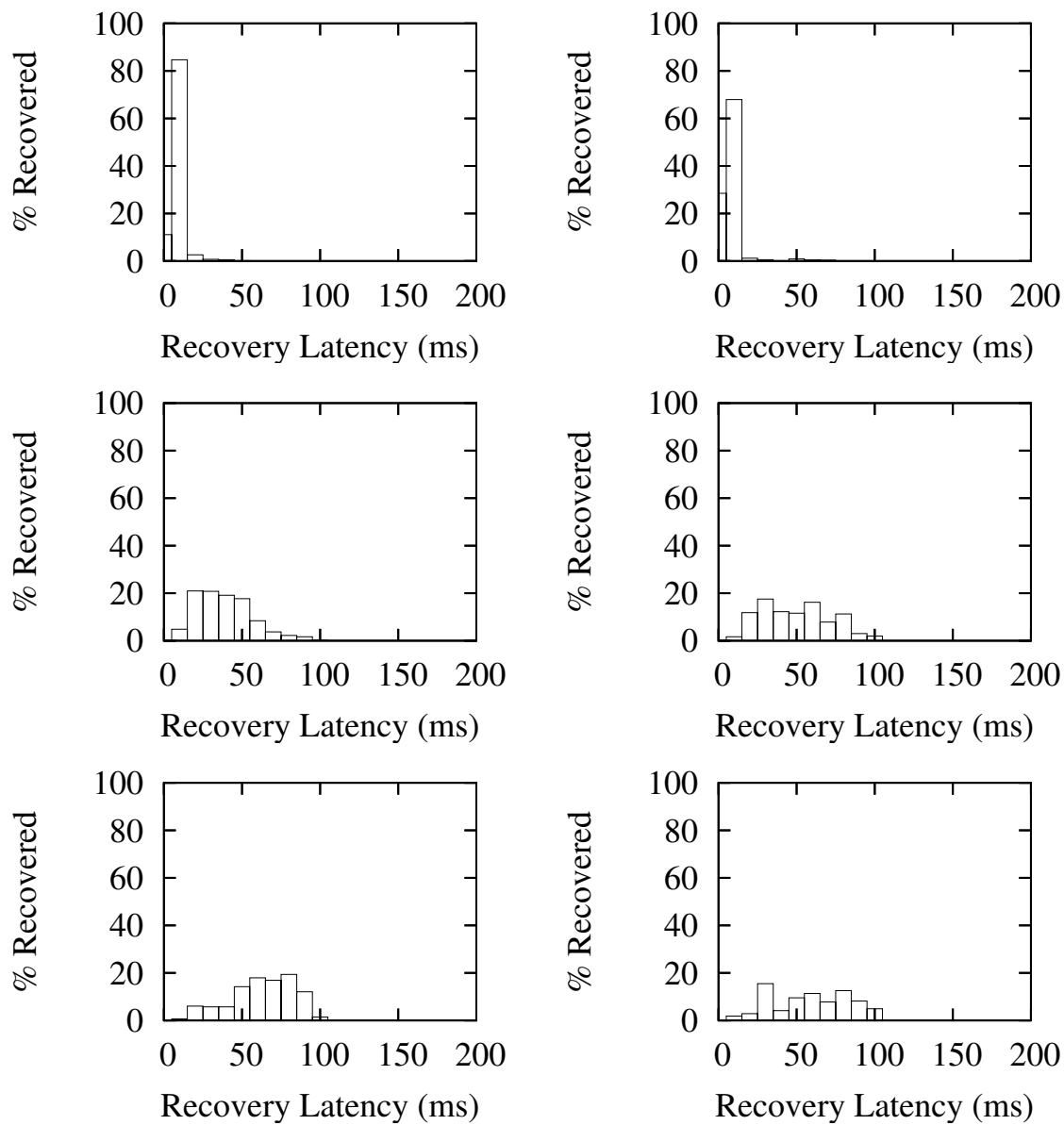


Figure 3.18: Latency Histograms for  $I=(1,11,19)$  (Left) and  $I=(1,19,41)$  (Right) — Burst Sizes 1 (Top), 20 (Middle) and 40 (Bottom)

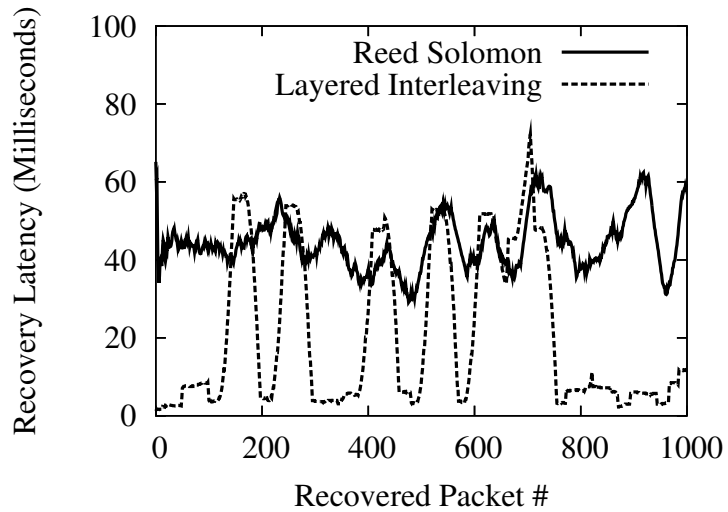


Figure 3.19: Reed-Solomon versus Layered Interleaving

ured with  $r = 8, c = 2$  and recover all lost packets — Reed-Solomon uses an interleave of 20 and layered interleaving uses interleaves of (1, 40) and consequently both have a maximum tolerable burst length of 40 packets. We use a publicly available implementation of a Reed-Solomon code based on Vandermonde matrices, described in [74]; the code is plugged into Maelstrom instead of layered interleaving, showing that we can use new encodings within the same framework seamlessly. The Reed-Solomon code recovers all lost packets with roughly the same latency whereas layered interleaving recovers singleton losses almost immediately and exhibits latency spikes whenever the longer loss burst occurs.

## CHAPTER 4

### RICOCHET

The last decade has seen the migration of time-critical applications to commodity clusters. Application domains ranging from computational finance to air-traffic control and military communication have been driven by scalability and cost concerns to abandon traditional real-time environments for COTS datacenters. In the process, they give up conservative - and arguably unnecessary - guarantees of real-time performance for the promise of massive scalability and multiple nines of timely availability, all at a fraction of the running cost. Delivering on this promise within expanding and increasingly complex datacenters is a non-trivial task, and a wealth of commercial technology has emerged to support clustered applications.

At the heart of commercial datacenter software is *reliable multicast* — used by publish-subscribe and data distribution layers [9, 12] to spread data through clusters at high speeds, by clustered application servers [1, 4, 5] to communicate state, updates and heartbeats between server instances, and by distributed caching infrastructures [2, 7] to rapidly update cached data. The multicast technology used in contemporary industrial products is derivative of protocols developed by academic researchers over the last two decades, aimed at scaling metrics like throughput or latency across dimensions as varied as group size [20, 34], node and network heterogeneity [26], or geographical and routing distance [35, 58]. However, these protocols were primarily designed to extend the reach of multicast to massive networks; they are not optimized for the failure modes of datacenters and may be unstable, inefficient and ineffective when retrofitted to clustered settings. Crucially, they are not designed to cope with the unique scalability demands of time-critical fault-tolerant applications.

We posit that a vital dimension of scalability for clustered applications is the *number*

*of multicast channels* in the system. We define a multicast channel as the pairing of a group of receivers and a single sender to that group. A single receiver can join many different groups; each group of receivers can consist of many separate channels, one for each sender transmitting data to it. We say that two groups overlap if they have common receivers that belong to both groups.

All the uses of multicast mentioned above induce large numbers of overlapping groups, each of which can have many senders transmitting to it. For example, a computational finance calculator that uses a topic-based pub-sub system to subscribe to a fraction of the equities on the stock market will end up as a receiver in many multicast groups. Multiple such applications within a datacenter - each subscribing to different sets of equities - can result in arbitrary patterns of group overlap. Similarly, data caching or replication at fine granularity can result in a single node hosting many data items. Replication driven by high-level objectives such as locality, load-balancing or fault-tolerance can lead to distinct overlapping replica sets - and hence, multicast groups - for each item.

In this chapter, we propose Ricochet, a time-critical reliable multicast protocol designed to perform well in the multicast patterns induced by clustered applications. Ricochet uses IP Multicast [30] to transmit data and recovers lost packets using *Lateral Error Correction* (LEC), a novel error correction mechanism in which XOR repair packets are probabilistically exchanged *between receivers and combined across overlapping multicast groups*. The latency of loss recovery in LEC depends inversely on the aggregate rate of data in the system, rather than the rate in any one channel. It performs equally well in any arbitrary configuration of channels, allowing Ricochet to scale to massive numbers of groups and senders while retaining the best characteristics of state-of-the-art multicast technology: even distribution of responsibility among receivers, insensitivity

to group size, stable proactive overhead and graceful degradation of performance in the face of increasing loss rates.

The contributions of this chapter are the following:

- We argue that a critical dimension of scalability for multicast in clustered settings is the number of channels in the system.
- We show that existing reliable multicast protocols have recovery latency characteristics that are inversely dependent on the data rate in a channel, and do not perform well when each receiver is in many low-rate multicast channel.
- We introduce receiver-based Forward Error Correction, a new reliability mechanism that involves multicast receivers generating and exchanging XORs to recover missing packets. Receiver-based FEC provides recovery latency independent of the number of senders to a single group.
- We extend receiver-based FEC to Lateral Error Correction by intelligently combining repair traffic across group boundaries. LEC provides recovery latency independent of the number of groups in the system.
- We describe the design and implementation of Ricochet, a reliable multicast protocol that uses LEC to achieve massive scalability in the number of channels in the system.
- We extensively evaluate the Ricochet implementation on a 64-node cluster, showing that it performs well with different loss rates, tolerates bursty loss patterns, and is relatively insensitive to grouping and sending patterns — providing recovery characteristics that degrade gracefully with the number of channels in the system, as well as other conventional dimensions of scalability.

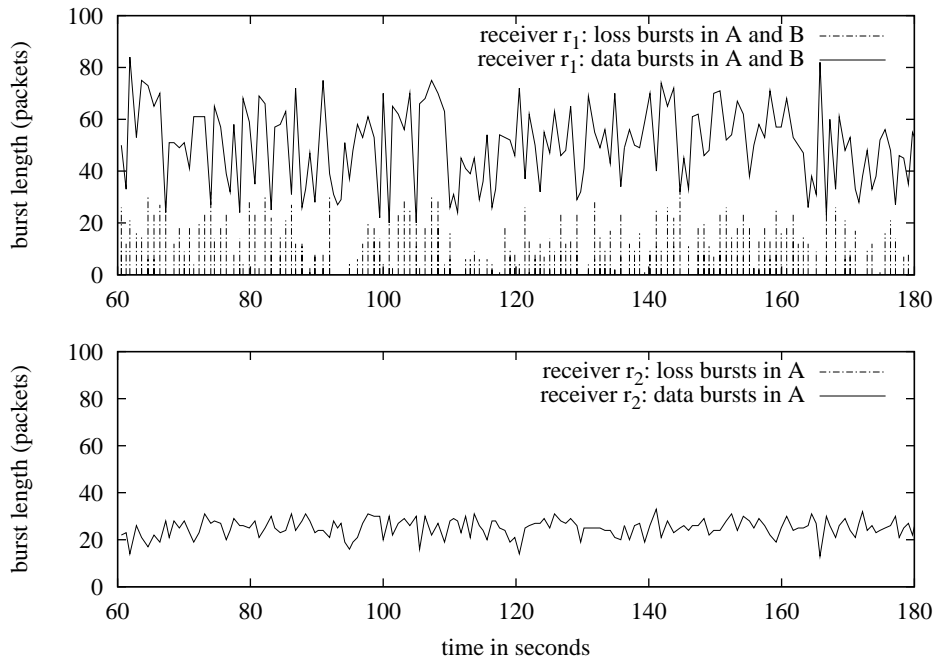
## 4.1 System Model

We consider patterns of multicast usage where each node is in many different groups of small to medium size (10 to 50 nodes). Following the IP Multicast model, a group is defined as a set of receivers for multicast data, and senders do not have to belong to the group to send to it. We expect each node to receive data from a large set of distinct senders, across all the groups it belongs to.

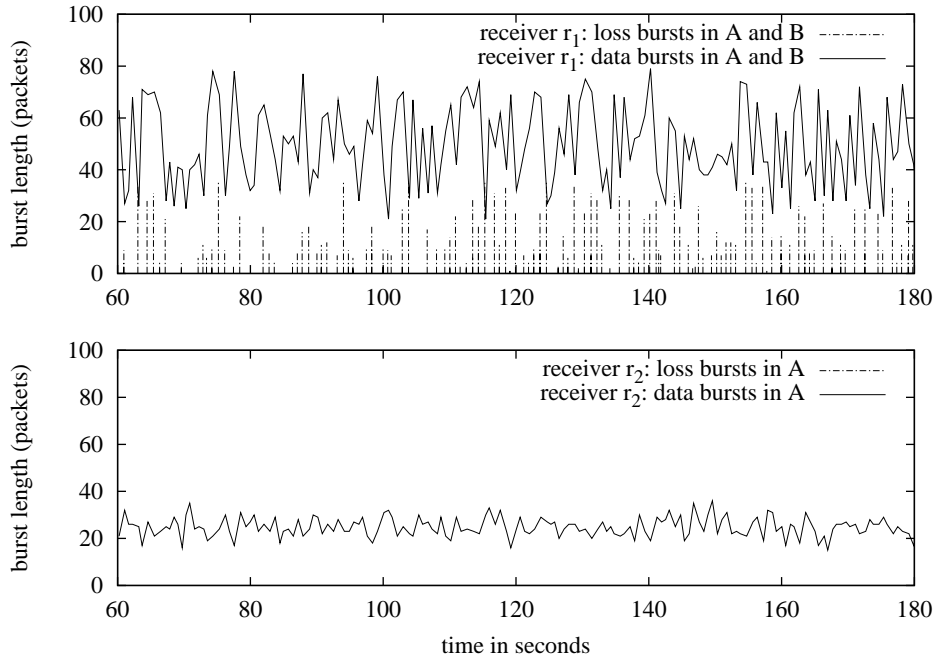
**Where does Loss occur in a Datacenter?** Datacenter networks have flat routing structures with no more than two or three hops on any end-to-end path. They are typically over-provisioned and of high quality, and packet loss in the network is almost non-existent. In contrast, datacenter end-hosts are inexpensive and easily overloaded; even with high-capacity network interfaces, the commodity OS often drops packets due to buffer overflows caused by traffic spikes or high-priority threads occupying the CPU. Hence, our loss model is one of short packet bursts dropped at the end-host receivers at varying loss rates.

Figure 4.1 strongly indicates that loss in a datacenter is (a) bursty and (b) independent across end-hosts. In this experiment, a receiver  $r_1$  joins two multicast groups  $A$  and  $B$ , and another receiver  $r_2$  in the same switching segment joins only group  $A$ . From a sender located multiple switches away on the network, we send per-second data bursts of around 25 1KB packets to group  $A$  and simultaneously send a burst of 0-50 packets to group  $B$ , and measure packet loss at both receivers. We ran this experiment on two networks: a 64-node cluster at Cornell with 1.3 Ghz receivers and the Emulab testbed at Utah with 2 Ghz receivers, all nodes running Linux 2.6.12.

The top graphs in Figure 4.1 show the traffic bursts and loss bursts at receiver  $r_1$ , and the bottom graphs show the same information for  $r_2$ . We can see that  $r_1$  gets overloaded



**(a) Cornell 64-node Cluster**



**(b) Utah Emulab Testbed**

Figure 4.1: Datacenter Loss is bursty and uncorrelated across nodes: receiver  $r_1$  (Top) joins groups A and B and exhibits bursty loss, whereas receiver  $r_2$  (Bottom) joins only group A and experiences zero loss.

and drops packets in bursts of size 1-30 packets, whereas  $r_2$  does not drop any packets — importantly, around 30% of the packets dropped by  $r_1$  are in group  $A$ , which is common to both receivers. Hence, loss is both bursty and independent across nodes. Together, these graphs indicate strongly that loss occurs due to buffer overflows at receiver  $r_1$ .

The example in Figure 4.1 is simplistic - each incoming burst of traffic arrives at the receiver within a small number of milliseconds - but conveys a powerful message: it is very easy to trigger significant bursty loss at datacenter end-hosts. The receivers in these experiments were running empty and draining packets continuously out of the kernel, with zero contention for the CPU or the network, whereas the settings of interest to us involve time-critical, possibly CPU-intensive applications running on top of the communication stack.

Further, we expect multi-group settings to intrinsically exhibit bursty incoming traffic of the kind emulated in this experiment — each node in the system receives data from multiple senders in multiple groups and it is likely that the inter-arrival time of data packets at a node will vary widely, even if the traffic rate at one sender or group is steady. In some cases, burstiness of traffic could also occur due to time-critical application behavior - for example, imagine an update in the value of a stock quote triggering off activity in several system components, which then multicast information to a replicated central datastore. If we assume that each time-critical component processes the update within a few hundred microseconds, and that inter-node socket-to-socket latency is around fifty microseconds (an actual number from our experimental cluster), the central datastore could easily see a sub-millisecond burst of traffic. In this case, the componentized structure of the application resulted in bursty traffic; in other scenarios, the application domain could be intrinsically prone to bursty input. For example, a financial calculator tracking a set of hundred equities with correlated movements might expect to

receive a burst of a hundred packets in multiple groups almost instantaneously.

## 4.2 The Design of a Time-Critical Multicast Primitive

In recent years, multicast research has focused almost exclusively on application-level routing mechanisms, or overlay networks ([27] is one example), designed to operate in the wide-area without any existing router support. The need for overlay multicast stems from the lack of IP Multicast coverage in the modern Internet, which in turn reflects concerns of administration complexity, scalability, and the risk of multicast ‘storms’ caused by misbehaving nodes. However, the homogeneity and comparatively limited size of datacenter networks pose few scalability and administration challenges to IP Multicast, making it a viable and attractive option in such settings. In this paper, we restrict ourselves to a more traditional definition of ‘reliable multicast’, as a reliability layer over IP Multicast. Given that the selection of datacenter hardware is typically influenced by commercial constraints, we believe that any viable solution for this context must be able to run on any mix of existing commodity routers and OS software; hence, we focus exclusively on application-level mechanisms, ruling out schemes which require router modification, such as PGM [38].

### 4.2.1 The Timeliness of (Scalable) Reliable Multicast Protocols

Reliable multicast protocols typically consist of three logical phases: *transmission* of the packet, *discovery* of packet loss, and *recovery* from it. Recovery is a fairly fast operation; once a node knows it is missing a packet, recovering it involves retrieving the packet from some other node. However, in most existing scalable multicast protocols,

the time taken to discover packet loss dominates recovery latency heavily in the kind of settings we are interested in. The key insight is that *the discovery latency of reliable multicast protocols is usually inversely dependent on data rate*: for existing protocols, the rate of outgoing data at a single sender in a single group. Existing schemes for reliability in multicast can be roughly divided into the following categories:

**ACK/timeout:** RMTP [58], RMTP-II [65]. In this approach, receivers send back ACKs (acknowledgements) to the sender of the multicast. This is the trivial extension of unicast reliability to multicast, and is intrinsically unscalable due to ACK implosion; for each sent message, the sender has to process an ACK from every receiver in the group [58]. One work-around is to use ACK aggregation, which allows such solutions to scale in the number of receivers but requires the construction of a tree for every sender to a group. Also, any aggregative mechanism introduces latency, leading to larger timeouts at the sender and delaying loss discovery; hence, ACK trees are unsuitable in time-critical settings.

**Gossip-Based:** Bimodal Multicast [20], Ipbcast [34]. Receivers periodically gossip histories of received packets with each other. Upon receiving a digest, a receiver compares the contents with its own packet history, sending any packets that are missing from the gossiped history and requesting transmission of any packets missing from its own history. Gossip-based schemes offer scalability in the number of receivers per group, and extreme resilience by diffusing the responsibility of ensuring reliability for each packet over the entire set of receivers. However, they are not designed for time-critical settings: discovery latency is equal to the time period between gossip exchanges (a significant number of milliseconds - 100ms in Bimodal Multicast [20]), and recovery involves a further one or two-phase interaction as the affected node obtains the packet from its gossip contact.

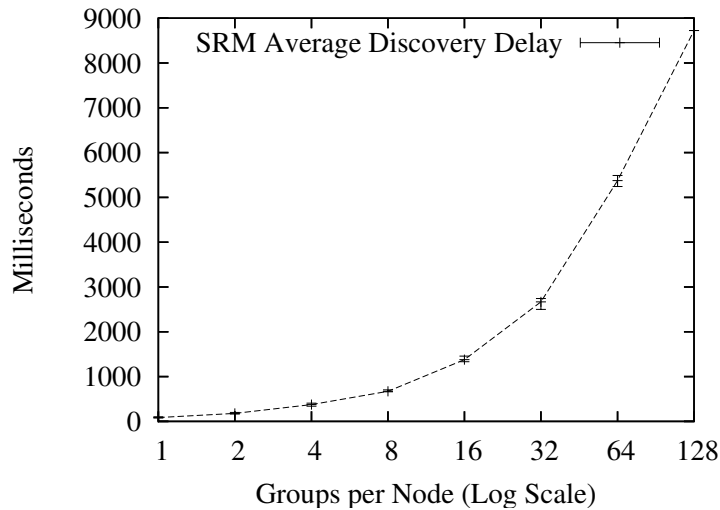


Figure 4.2: SRM’s Discovery Latency vs. Groups per Node, on a 64-node cluster, with groups of 10 nodes each. Error bars are min and max over 10 runs.

**NAK/Sender-based Sequencing:** SRM [35]. Senders number outgoing multicasts, and receivers discover packet loss when a subsequent message arrives. Loss discovery latency is thus proportional to the inter-send time at any single sender to a single group - a receiver can’t discover a loss in a group until it receives the next packet from the same sender to that group - and consequently depends on the sender’s data transmission rate to the group. To illustrate this point, we measured the performance of SRM as we increased the number of groups each node belonged in, keeping the throughput in the system constant by reducing the data rate within each group - as Figure 4.2 shows, discovery latency of lost packets degrades linearly as each node’s bandwidth is increasingly fragmented and each group’s rate goes down, increasing the time between two consecutive sends by a sender to the same group. Once discovery occurs in SRM, lost packet recovery is initiated by the receiver, which uses IP multicast (with a suitable TTL value); the sender (or some other receiver), responds with a retransmission, also using IP multicast.

**Sender-based FEC [47, 67]:** Forward Error Correction schemes involve multicasting

redundant error correction information along with data packets, so that receivers can recover lost packets without contacting the sender or any other node. FEC mechanisms involve generating  $c$  repair packets for every  $r$  data packets, such that any  $r$  of the combined set of  $r+c$  data and repair packets is sufficient to recover the original  $r$  data packets; we term this  $(r, c)$  parameter the *rate-of-fire*. FEC mechanisms have the benefit of *tunability*, providing a coherent relationship between overhead and timeliness - the more the number of repair packets generated, the higher the probability of recovering lost packets from the FEC data. Further, FEC based protocols are very stable under stress, since recovery does not induce large degrees of extra traffic. As in NAK protocols, the timeliness of FEC recovery depends on the data transmission rate of a single sender in a single group; the sender can send a repair packet to a group only after sending out  $r$  data packets to that group. Fast, efficient encodings such as Tornado codes [24] make sender-based FEC a very attractive option in multicast applications involving a single, dedicated sender; for example, software distribution or Internet radio.

The key idea behind the Ricochet protocol is *receiver-based Forward Error Correction*. In conventional FEC-based protocols, the multicast *sender* generates FEC packets from *outgoing* data. In Ricochet, the *receivers* in the multicast group generate FEC packets from *incoming* data. These receiver-generated FEC packets are then exchanged between randomly chosen receivers to enable rapid recovery from packet loss. Receiver-based FEC combines the tunability and timeliness of conventional FEC with the scalability of gossip-based protocols — the recovery latency of lost packets is independent of the number of senders to a single group. However, the recovery latency still depends on the aggregate data rate in a single group.

To scale to large numbers of groups, Ricochet uses a variant of receiver-based FEC called Lateral Error Correction (LEC), where receivers exchange repair packets across

group boundaries. In LEC, a FEC packet sent by one receiver to another can be composed from data packets received in multiple groups that both receivers belong to. As a result, the packet recovery latency of Ricochet effectively depends on the aggregate data rate at a receiver across all groups rather than the rate within any single group. In the next two sections, we will first describe the Ricochet implementation of single-group receiver-based FEC and then discuss its extension to multiple groups through the LEC algorithm. The single-group version of Ricochet was published in NCA 2005 [17] as the Slingshot protocol, and the final multi-group version that uses LEC was published in NSDI 2007 [15].

### 4.3 Receiver-based Forward Error Correction in a Single Group

In this section, we will assume that a separate instance of Ricochet is run within each group in the system; hence, the protocol views the system as a single isolated group and has no awareness of other groups. As implied by its name, receiver-based FEC runs at the receivers in a multicast group and does not require any modification to the sender, which transmits data using IP Multicast. The basic operation of the protocol involves generating XORs from incoming data at receivers and exchanging them with other randomly chosen receivers.

Ricochet operates using two different packet types: *data packets* - the actual data multicast within a group - and *repair packets*, which contain recovery information for multiple data packets. Figure 4.3 shows the structure of these two packet types. Each data packet header contains a packet identifier - a *(sender, group, sequence number)* tuple that identifies it uniquely. In the single-group version, the group identifier is naturally identical for all packets received by an instance of the protocol.

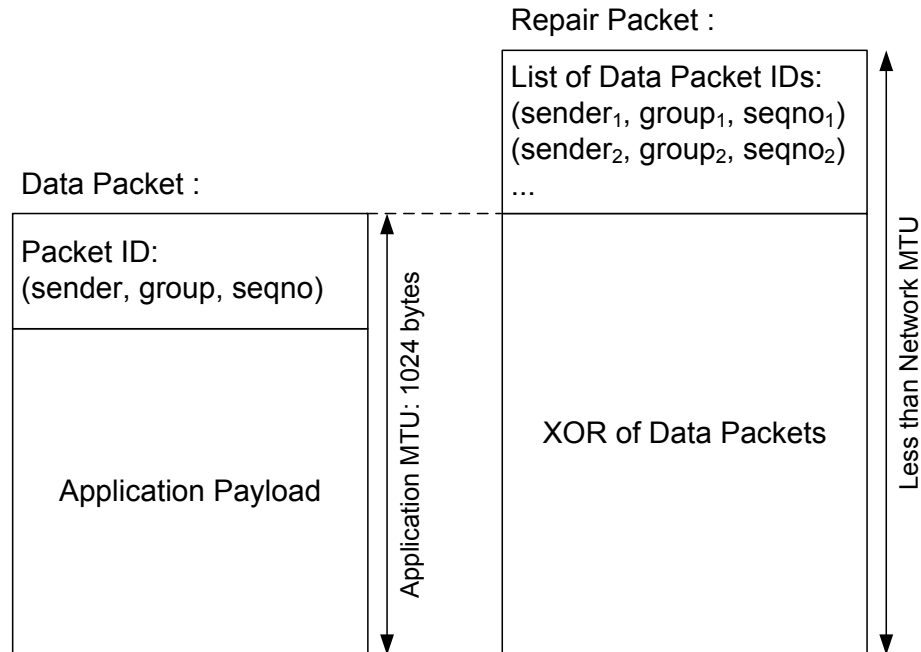


Figure 4.3: Ricochet Packet Structure

A repair packet contains an XOR of multiple data packets, along with a list of their identifiers - we say that the repair packet is *composed* from these data packets, and that the data packets are *included* in the repair packet. An XOR repair composed from  $r$  data packets allows recovery of one of them, if all the other  $r-1$  data packets are available; the missing data packet is obtained by simply computing the XOR of the repair's payload with the other data packets. To ensure that both data and repair packets are sent in the network without fragmentation, data packets are limited to a size slightly less than the Maximum Transmission Unit (MTU) of the network; this ensures that a repair packet has space for the XOR, which is equal to the size of a single data packet, as well as a list of message identifiers describing the data packets included in it.

The mechanism of receiver-based Forward Error Correction within a single group is simple: for every  $r$  data packets that a node receives, it generates an XOR and transmits it via unicast to  $c$  randomly chosen receivers in the group. We call the tuple  $(r, c)$  the *rate-of-fire* of the group — it determines both the network overhead incurred by the protocol

as well as the resulting packet recovery characteristics. In order to select  $c$  random targets from the group for each repair packet, a Ricochet node maintains a ‘view’ of the group’s membership — essentially, a list of other nodes in the group. The view needs to only contain a random subset of the nodes in the actual group; we describe the membership mechanism in detail later in this chapter.

When a data packet arrives at a Ricochet node,

- it is sent to a special data structure called the *repair bin*. The repair bin consists of an incrementally computed XOR and a list of data packet identifiers — as each data packet is inserted into the bin, a new XOR is computed and the identifier list is updated. When  $r$  data packets have been added to the repair bin, it ‘fires’ by creating a new repair packet and sending it to  $c$  randomly selected targets in the group. The state of the repair bin is then reset to a blank XOR buffer and an empty identifier list.
- its identifier is added to the *packet tracker*, a module that maintains two pieces of information for each sender in the group: the highest data packet sequence number known to exist as well as a list of sequence numbers for packets known to exist but not yet received at this node. The latter is stored as a list of intervals for efficiency; for example, if the packets with sequence numbers 5, 6, 7 and 8 are missing, the list has a single entry for the interval [5 – 8]. The incoming data packet’s identifier is sent twice to the packet tracker through separate methods — once to notify the tracker of its existence and again to notify the tracker of its receipt. Entries are added to the missing packet list whenever an identifier is sent to the packet tracker with a sequence number that exceeds the highest known sequence number by more than 1, indicating a gap in the sequence of known packets.
- it is stored in the *data buffer*. In order to recovery missing packets from the XORs

contained in repair packets, a Ricochet node needs to have the other data packets included in the XOR. The data buffer stores recently arrived data packets in case they are required for the recovery of a missing packet from an XOR.

When a repair packet arrives at a Ricochet node,

- the packet tracker is notified of the existence of all the data packets included in the XOR. It updates the highest known sequence number and the list of missing packets appropriately.
- the packet tracker is consulted to check how many of the data packets included in the XOR are missing. If none of the packets are missing, the repair packet can be discarded. If more than one is missing, the repair packet cannot be immediately used to recover lost data, and it is stored in the *repair buffer* for potential future use. If exactly one data packet is missing, it can be recovered using the XOR in the repair packet. All the other data packets included in the XOR are extracted from the data buffer and combined with it to regenerate the missing packet.

### 4.3.1 Analysis of Receiver-based FEC

We present a simplistic analysis to predict how the probability of a lost packet being recovered depends on the rate-of-fire parameter, the size of the group, the partial view size, and the probability of packet loss. We assume that packets are dropped independently at end node buffers and that the datacenter network does not drop packets; this is consistent with the loss model described in 4.1. A major simplifying assumption is that packets are dropped with a random independent probability  $p$  rather than in bursts. Given a group size  $N_G$ , a partial view size  $N_v$  and a rate-of-fire parameter  $(r, c)$ , we want

---

**Algorithm 1** Single Group Receiver-based FEC algorithm.

---

1: **Code of receiver**  $p_i$ :

2: Initialisation:

3:    $DataBuffer \leftarrow \emptyset$  {Contains received Data Packets}

4:    $RepairBin \leftarrow \emptyset$  {Data packets to create next Repair Packet}

5:    $LastSeqNo[] \leftarrow \emptyset$  {Last known data packet from each sender}

6:    $KnownLost \leftarrow \emptyset$  {List of Message IDs thought to be lost}

7: **upon** r-multicast( $dp$ ) **do**

8:   send ( $dp$ ) to all processes in  $v$ .

9: **upon** reception of data packet  $dp$  from sender  $p_j$  **do**

10:   deliver  $dp$  to the application

11:    $DataBuffer \leftarrow DataBuffer \cup \{dp\}$

12:    $RepairBin \leftarrow RepairBin \cup \{dp\}$

13:   markLost( $dp.id$ )

14:    $KnownLost \leftarrow KnownLost - \{dp.id\}$

15:   composeRepairPac()

{Recovery}

16: **upon** reception of repair packet  $rp$  from sender  $p_k$  **do**

17:   **for all** message id  $id \in rp.MsgIdList$  **do**

18:     markLost( $id$ )

19:     **if**  $|\{id|id \in KnownLost \wedge id \in rp.MsgIdList\}| = 1$  **then**

20:       Recover data packet  $dp$  corresponding to this  $id$

21:       deliver  $dp$  to the application

22:        $DataBuffer \leftarrow DataBuffer \cup \{dp\}$

23:        $KnownLost \leftarrow KnownLost - \{dp.id\}$

{Marks unreceived messages preceding passed-in  $id$  as lost}

24: **procedure** markLost( $id$ )

25:   **for**  $i = lastseqno[id.sender] + 1$  to  $id.seqno$  **do**

26:      $KnownLost \leftarrow KnownLost \cup \{(id.sender, i)\}$

27:    $lastseqno[id.sender] \leftarrow id.seqno$

{Constructs FEC repair packet}

28: **procedure** composeRepairPac

29:   **if**  $|RepairBin| \geq r$  **then** {(r, c) denotes the rate-of-fire}

30:      $DestinationSet \leftarrow$  select  $c$  processes  $q$  s.t.  $q \in v$

31:     generate repair packet  $rp$

32:     send  $rp$  to all  $q$  in  $DestinationSet$

33:      $RepairBin \leftarrow \emptyset$

---

to predict the probability of recovering a given data packet  $dp$  lost at node  $n$ . We make the assumption that the partial view at a node is a uniformly chosen subset of the whole group membership. In this analysis we assume that the node never includes itself in its view.

Now, let  $X$  be a random variable signifying the number of nodes in the system which have  $n$  in their partial views. Since each partial view is a uniformly picked subset of the group membership, the probability of  $n$  being included in a view other than its own is  $p_v = \frac{\binom{N_G-2}{N_v-1}}{\binom{N_G-1}{N_v}}$ . Thus,  $X$  has a binomial distribution with parameters  $N_G - 1$  and  $p_v$ .

Given that the number of nodes including  $n$  in their views is a particular value  $i$ , let  $Y$  be a random variable denoting the number of repair packets originating at these nodes that include the data packet  $dp$  and are targeted at  $n$ . The upper bound on the total number of such packets is  $i$ , for the case that each of the  $i$  nodes receives  $dp$  without loss and sends out a repair packet to  $n$ . At any of the  $i$  nodes, the probability of  $n$  being selected as one of the  $c$  destinations is  $p_c = \frac{\binom{N_v-1}{c-1}}{\binom{N_v}{c}}$ . If we also consider the probability  $p$  of the sender of the repair packet dropping  $dp$ , then  $Y$  has a binomial distribution with parameters  $i$  and  $p_c(1 - p)$ .

If we set the number of repair packets which include  $dp$  and are targeted at  $n$  to a value  $j$ , then the number of such packets received without loss by  $n$  is represented by a random variable  $Z$  that has a binomial distribution with parameters  $j$  and  $1 - p$ . Let us set  $Z$  to the value  $k$ . Now, we need to compute the probability  $p_k$  of  $dp$  being recovered if  $n$  receives  $k$  repair packets containing it. We can recover  $dp$  if, for at least one of the incoming repair packets containing it,  $n$  has all the other  $r - 1$  data packets included in that repair packet. We derive inclusive upper and lower bounds  $p_{kL}$  and  $p_{kU}$  for  $p_k$ , where  $k \geq 1$ ; it is equal to zero when  $k = 0$ . The lower bound corresponds to the case where all  $k$  repair packets have the same contents, and the upper bound is given

by the case where all  $k$  repair packets are pair-wise disjoint; i.e they include completely different data packets. Hence,  $p_k$  is bounded by:

$$p_{kL} = (1 - p)^{r-1} \leq p_k \leq p_{kU} = 1 - (1 - (1 - p)^{r-1})^k$$

Hence, the final probability  $P_{n,dp}$  of a data packet  $dp$  lost at node  $n$  being recovered successfully is bounded by:

$$\begin{aligned} & \sum_{i=0}^{N_G} P(X = i) \sum_{j=0}^i P(Y = j) \sum_{k=0}^j P(Z = k) p_{kL} \\ & \leq P_{n,dp} \\ & \leq \sum_{i=0}^{N_G} P(X = i) \sum_{j=0}^i P(Y = j) \sum_{k=0}^j P(Z = k) p_{kU} \end{aligned}$$

This analysis shows that the percentage of packets recovered by receiver-based FEC has a coherent relationship with the  $(r, c)$  parameter as well as the loss rate in the system.

#### 4.4 Lateral Error Correction for Multiple Groups

Receiver-based FEC within a single group involves the exchange of repair packets between the receivers of a single group. Lateral Error Correction is an extension of this mechanism to multiple groups. Each Ricochet node runs an LEC engine that decides on the composition and destinations of repair packets, creating them from incoming data across multiple groups. The operating principle behind LEC is the notion that repair packets sent by a node to another node can be composed from data in any of the multi-cast groups that are common to them. This allows recovery of lost packets at the receiver of the repair packet to occur with latency that depends inversely on the aggregate rate of data in all these groups, rather than the rate of data in any single group. Figure 4.4

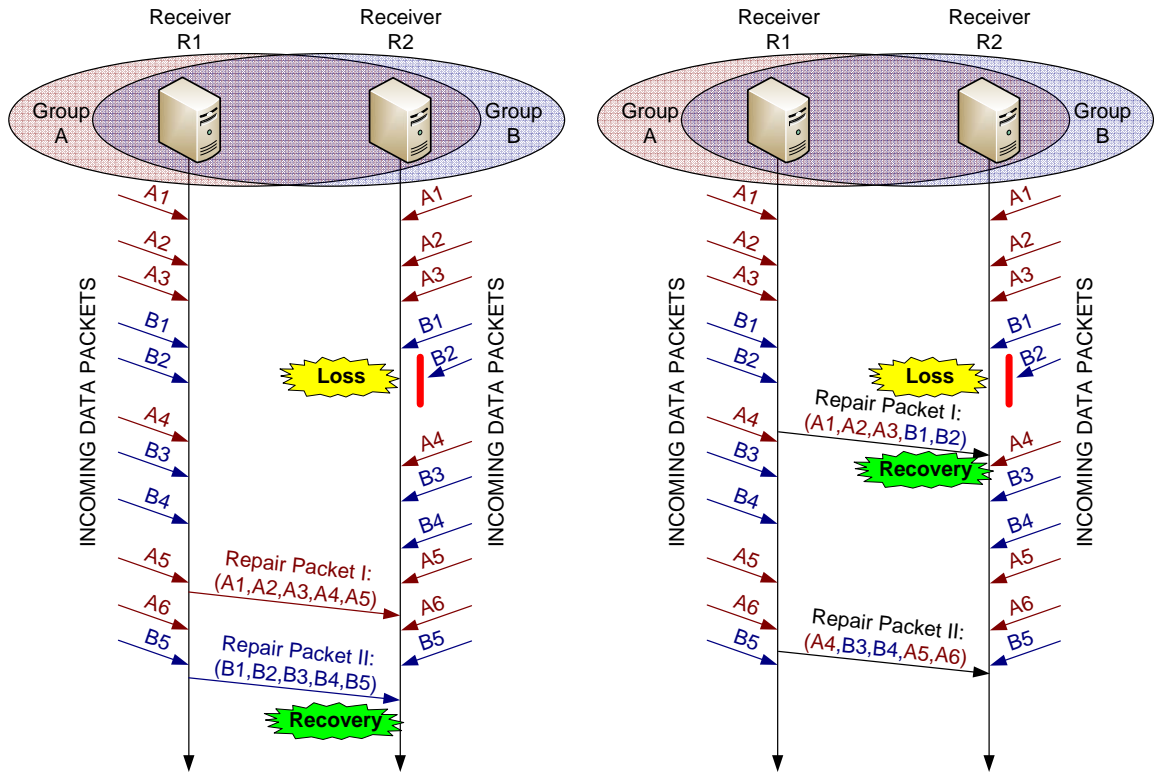


Figure 4.4: LEC in 2 Groups: Receiver  $n_1$  can send repairs to  $n_2$  that combine data from both groups  $A$  and  $B$ .

illustrates this idea:  $n_1$  has groups  $A$  and  $B$  in common with  $n_2$ , and hence it can generate and dispatch repair packets that contain data from both these groups.  $n_1$  needs to wait only until it receives 5 data packets in either  $A$  or  $B$  before it sends a repair packet, allowing faster recovery of lost packets at  $n_2$ .

While combining data from different groups in outgoing repair packets drives down recovery time, it tampers with the coherent tunability that single group receiver-based FEC provides. The *rate-of-fire* parameter in receiver-based FEC provides a clear, coherent relationship between overhead and recovery percentage; for every  $r$  data packets,  $c$  repair packets are generated in the system, resulting in some computable probability of recovering from packet loss. The challenge for LEC is to combine repair traffic for

multiple groups while retaining per-group overhead and recovery percentages, so that each individual group can maintain its own rate-of-fire. To do so, we abstract out the essential properties of receiver-based FEC that we wish to maintain:

1. **Coherent, Tunable Per-Group Overhead:** For every data packet that a node receives in a group with rate-of-fire  $(r, c)$ , it sends out *an average of  $c$  repair packets* including that data packet to other nodes in the group.
2. **Randomness:** Destination nodes for repair packets are picked *randomly*, with no node receiving more or less repairs than any other node, on average.

LEC supports overlapping groups with the same  $r$  component and different  $c$  values in their rate-of-fire parameter. In LEC, the rate-of-fire parameter is translated into the following guarantee: For every data packet  $d$  that a node receives in a group with rate-of-fire  $(r, c)$ , it selects an average of  $c$  nodes from the group randomly and sends each of these nodes exactly one repair packet that includes  $d$ . In other words, the node sends an average of  $c$  repair packets containing  $d$  to the group. In the following section, we describe the algorithm that LEC uses to compose and dispatch repair packets while maintaining this guarantee.

#### 4.4.1 Algorithm Overview

Ricochet is a symmetric protocol - exactly the same LEC algorithm and supporting code runs at every node - and hence, we can describe its operation from the vantage point of a single node,  $n_1$ .

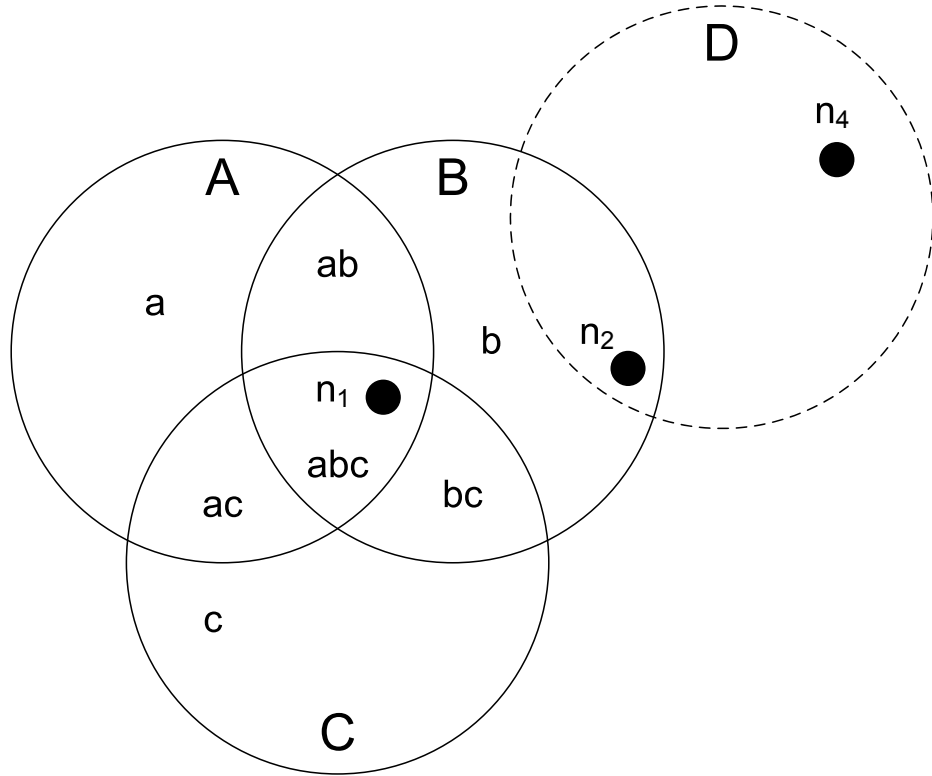
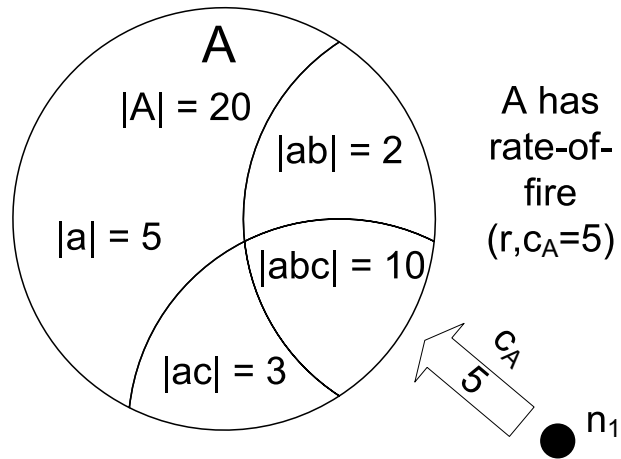


Figure 4.5:  $n_1$  belongs to groups  $A, B, C$ : it divides them into disjoint regions  $abc, ab, ac, bc, a, b, c$

### Regions

The LEC engine running at  $n_1$  divides  $n_1$ 's neighborhood - the set of nodes it shares one or more multicast groups with - into *regions*, and uses this information to construct and disseminate repair packets. Regions are simply the disjoint intersections of all the groups that  $n_1$  belongs to. Figure 4.5 shows the regions in a hypothetical system, where  $n_1$  is in three groups,  $A, B$  and  $C$ . We denote groups by upper-case letters and regions by the concatenation of the group names in lowercase; i.e.,  $abc$  is a region formed by the intersection of  $A, B$  and  $C$ . In our example, the neighborhood set of  $n_1$  is carved into seven regions:  $abc, ac, ab, bc, a, b$  and  $c$ , essentially the power set of the set of groups involved. Readers may be alarmed that this transformation results in an exponential number of regions, but this is not the case; we are only concerned with non-empty



$$c_A^x = |x|/|A| * c_A$$

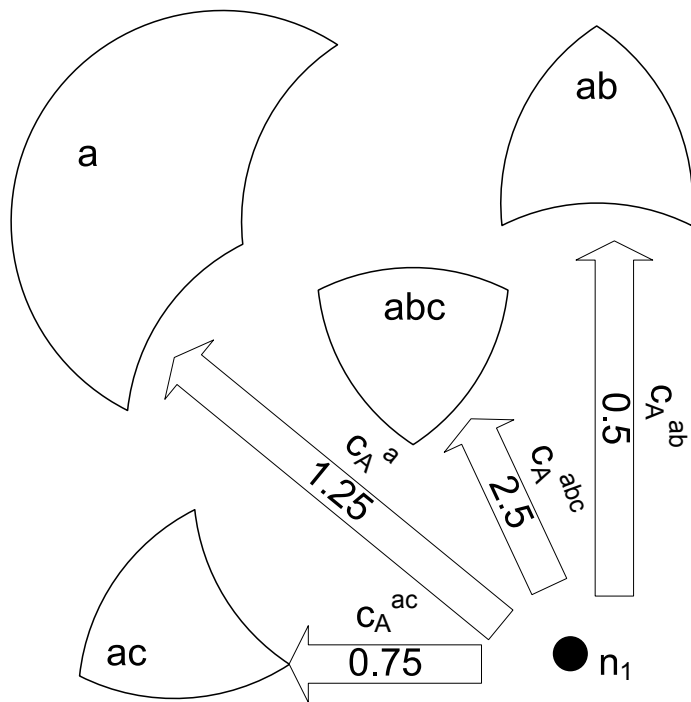


Figure 4.6:  $n_1$  selects proportionally sized chunks of  $c_A$  from the regions of **A**

intersections, the cardinality of which is bounded by the number of nodes in the system, as each node belongs to exactly one intersection (see Section 4.4.1). Note that  $n_1$  does not belong to group  $D$  and is oblivious to it; it observes  $n_2$  as belonging to region  $b$ , rather than  $bd$ , and is not aware of  $n_4$ 's existence.

### Selecting targets from regions, not groups

Instead of selecting targets for repairs randomly from the entire group, LEC selects targets randomly from *each region*. The number of targets selected from a region is set such that:

1. It is proportional to the size of the region.
2. The total number of targets selected, across regions, is equal to the  $c$  value of the group.

Hence, for a given group  $A$  with rate-of-fire  $(r, c_A)$ , the number of targets selected by LEC in a particular region, say  $abc$ , is equal to  $c_A * \frac{|abc|}{|A|}$ , where  $|x|$  is the number of nodes in the region or group  $x$ . We denote the number of targets selected by LEC in region  $abc$  for packets in group  $A$  as  $c_A^{abc}$ . Figure 4.6 shows  $n_1$  selecting targets for repairs from the regions of  $A$ .

Note that LEC may pick a different number of targets from a region for packets in a different group; for example,  $c_A^{abc}$  differs from  $c_B^{abc}$ . Selecting targets in this manner also preserves randomness of selection; if we rephrase the task of target selection as a sampling problem, where a random sample of size  $c$  has to be selected from the group, selecting targets from regions corresponds to *stratified sampling* [28], a technique from statistical theory.

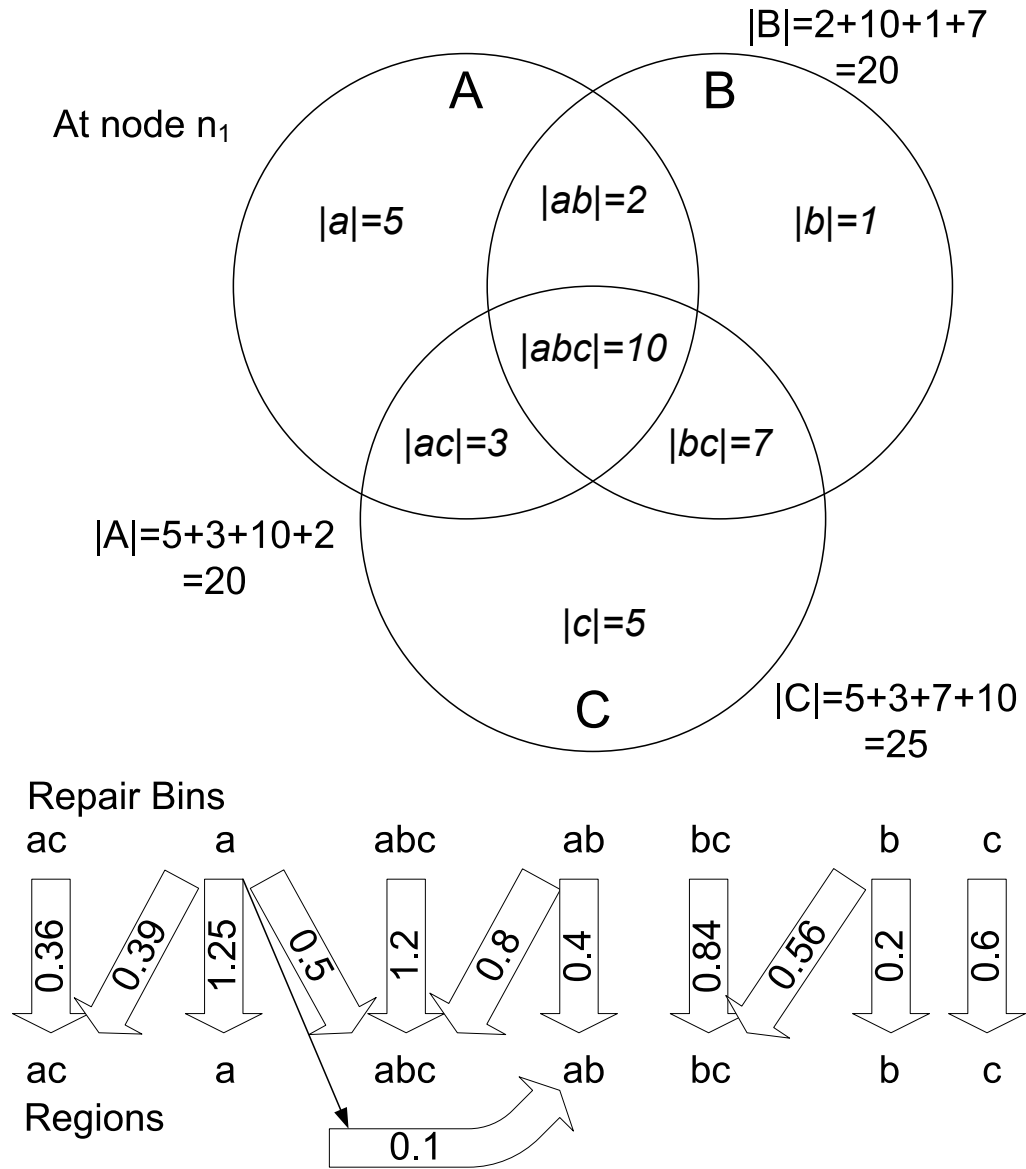


Figure 4.7: Mappings between repair bins and regions: the repair bin for  $ab$  selects 0.4 targets from region  $ab$  and 0.8 from  $abc$  for every repair packet. Here,  $c_A = 5$ ,  $c_B = 4$ , and  $c_C = 3$ .

## Why select targets from regions?

Selecting targets from regions instead of groups allows LEC to construct repair packets from multiple groups; since we know that all nodes in region  $ab$  are interested in data from groups  $A$  and  $B$ , we can create composite repair packets from incoming data packets in both groups and send them to nodes in that region.

Receiver-based FEC in a single group was implemented using repair bins, as described in the previous section. Recall that a repair bin collects incoming data within a group at a receiver — when it accumulates  $r$  data packets, a repair packet is generated from its contents and sent to  $c$  randomly selected nodes in the group, after which the bin is cleared. Extending the repair bin construct to regions seems simple; a bin can be maintained for each region, collecting data packets received in any of the groups composing that region. When the bin fills up to size  $r$ , it can generate a repair packet containing data from all these groups, and send it to targets selected from within the region.

Using per-region repair bins raises an interesting question: if we construct a composite repair packet from data in groups  $A$ ,  $B$ , and  $C$ , how many targets should we select from region  $abc$  for this repair packet -  $c_A^{abc}$ ,  $c_B^{abc}$ , or  $c_C^{abc}$ ? One possible solution is to pick the maximum of these values. If  $c_A^{abc} \geq c_B^{abc} \geq c_C^{abc}$ , then we would select  $c_A^{abc}$ . However, a data packet in group  $B$ , when added to the repair bin for the region  $abc$  would be sent to an average of  $c_A^{abc}$  targets in the region; resulting in more repair packets containing that data packet sent to the region than required ( $c_B^{abc}$ ), which results in more repair packets sent to the entire group. Hence, more overhead is expended per data packet in group  $B$  than required by its  $(r, c_B)$  value; a similar argument holds for data packets in group  $C$  as well.

---

**Algorithm 2** LEC Algorithm for Setting Up Repair Bins

---

- 1: **Code at node**  $n_i$ :
  
  - 2: **upon** Change in Group Membership **do**
  - 3:   **while** L not empty *{L is the list of regions}*  
    **do**
  - 4:     Select and remove the region  $R_i = abc\dots z$  from  $L$  with highest number of groups involved (break ties in any order)
  - 5:     Set  $R_t = R_i$
  - 6:     **while**  $R_t \neq \epsilon$  **do**
  - 7:       set  $c_{min}$  to  $\min(c_A^{R_t}, c_B^{R_t} \dots)$ , where  $\{A, B, \dots\}$  is the set of groups forming  $R_t$
  - 8:       Set number of targets selected by  $R_i$ 's repair bin from region  $R_t$  to  $c_{min}$
  - 9:       Remove  $G$  from  $R_t$ , for all groups  $G$  where  $c_G^{R_t} = c_{min}$
  - 10:      For each remaining group  $G'$  in  $R_t$ , set  $c_{G'}^{R_t} = c_{G'}^{R_t} - c_{min}$
- 

Instead, we choose the *minimum* of values; this, as expected, results in a lower level of overhead for groups  $A$  and  $B$  than required, resulting in a lower fraction of packets recovered from LEC. To rectify this we send the additional compensating repair packets to the region  $abc$  from the repair bins for regions  $a$  and  $b$ . The repair bin for region  $a$  would select  $c_A^{abc} - c_C^{abc}$  destinations, on an average, for every repair packet it generates; this is in addition to the  $c_A^a$  destinations it selects from region  $a$ .

A more sophisticated version of the above strategy involves iteratively obtaining the required repair packets from regions involving the remaining groups; for instance, we would have the repair bin for  $ab$  select the minimum of  $c_A^{abc}$  and  $c_B^{abc}$  - which happens to be  $c_B^{abc}$  - from  $abc$ , and then have the repair bin for  $a$  select the remainder value,  $c_A^{abc} - c_B^{abc}$ , from  $abc$ . Algorithm 2 illustrates the final approach adopted by LEC, and Figure 4.7 shows the output of this algorithm for an example scenario. A repair bin selects a non-integral number of nodes from an intersection by alternating between its floor and ceiling probabilistically, in order to maintain the average at that number.

## Complexity

The algorithm described above is run every time nodes join or leave any of the multicast groups that  $n_1$  is part of. The algorithm has complexity  $O(I \cdot d)$ , where  $I$  is the number of populated regions (i.e, with one or more nodes in them), and  $d$  is the maximum number of groups that form a region. Note that  $I$  at  $n_1$  is bounded from above by the cardinality of the set of nodes that share a multicast group with  $n_1$ , since regions are disjoint and each node exists in exactly one of them.  $d$  is bounded by the number of groups that  $n_1$  belongs to.

## 4.5 Details of the Ricochet Implementation

Our initial implementation of Ricochet was in Java, as described in [15], and was intended solely as a research prototype to illustrate the effectiveness of the LEC idea. Subsequently, we have re-implemented Ricochet in C++ with the aim of building a production quality protocol stack that is highly modular and easy to use. The newer version - which we call Ricochet++ - is a work-in-progress; while it already includes all the functionality of the original Java version, we are in the process of adding a large and diverse feature set to it, including transparent socket-based interfaces, on-the-fly adaptivity and gray-box flow control. In the remainder of this thesis, we will primarily present the initial Java version along with pointers to changes in the newer C++ version where appropriate.

### 4.5.1 Implementation of Repair Bins

The repair bin implementation holds an XOR and a list of data packets, and supports an *add* operation that takes in a data packet and includes it in the internal state. The repair bin is associated with a particular region, receiving all data packets incoming in any of the groups forming that region. It has a list of regions from which it selects targets for repair packets; each of these regions is associated with a value, which is the average number of targets which must be selected from that region for an outgoing repair packet. In most cases, as shown in Figure 4.7, the value associated with a region is not an integer; as mentioned before, the repair bin alternates between the floor and the ceiling of the value to maintain the average at the value itself. For example, in Figure 4.7, the repair bin for *abc* has to select 1.2 targets from *abc*, on average; hence, it generates a random number between 0 and 1 for each outgoing repair packet, selecting 1 node if the random number is more than 0.2, and 2 nodes otherwise.

### 4.5.2 Staggering for Bursty Loss

A crucial algorithmic optimization in Ricochet is *staggering* - also known as interleaving [67] - which provides resilience to bursty loss. Given a sequence of data packets to encode, a stagger of 2 would entail constructing one repair packet from the 1st, 3rd, 5th... packets, and another repair packet from the 2nd, 4th, 6th... packets. The stagger value defines the number of repairs simultaneously being constructed, as well as the distance in the sequence between two data packets included in the same repair packet. Consequently, a stagger of  $i$  allows us to tolerate a loss burst of size  $i$  while resulting in a proportional slowdown in recovery latency, since we now have to wait for  $O(i * r)$  data packets before despatching repair packets.

In conventional sender-based FEC, staggering is not a very attractive option, providing tolerance to very small bursts at the cost of multiplying the already prohibitive loss discovery latency. However, LEC recovers packets so quickly that we can tolerate a slowdown of a factor of ten without leaving the tens of milliseconds range; additionally, a small stagger at the sender allows us to tolerate very large bursts of lost packets at the receiver, especially since the burst is dissipated among multiple groups and senders. Ricochet implements a stagger of  $i$  by the simple expedient of duplicating each logical repair bin into  $i$  instances; when a data packet is added to the logical repair bin, it is actually added to a particular instance of the repair bin, chosen in round-robin fashion. Instances of a duplicated repair bin behave exactly as single repair bins do, generating repair packets and sending them to regions when they get filled up.

### 4.5.3 Multi-Group Views

Each Ricochet node has a *multi-group view*, which contains membership information about other nodes in the system that share one or more multicast groups with it. In traditional group communication literature, a *view* is simply a list of members in a single group [87]; in contrast, a Ricochet node's multi-group view divides the groups that it belongs to into a number of regions, and contains a list of members lying in each region. Ricochet uses the multi-group view at a node to determine the sizes of regions and groups, to set up repair bins using the LEC algorithm. Also, the per-region lists in the multi-view are used to select destinations for repair packets. The multi-group view at  $n_1$  - and consequently the group and intersection sizes - does not include  $n_1$  itself.

#### 4.5.4 Membership and Failure Detection

Ricochet can plug into any existing membership and failure detection infrastructure, as long as it is provided with reasonably up-to-date views of per-group membership by some external service. In the initial implementation described in [15], we used simple centralized versions of Group Membership (GMS) and Failure Detection (FD) services which executed on high-end server machines; in the later C++ version we replaced these with distributed gossip-based protocols.

In the centralized version, the GMS and the FD services are independent entities that act in concert to provide nodes with group views. The FD service periodically pings machines to check if they are up and running; if it fails to receive a response to multiple pings over a period of time (with both the number of pings and the time period being configurable parameters) the FD reports the node as failed to the GMS. When the GMS receives such a notification from the FD, or alternatively receives a join/leave to a group from a node, it sends an update to all nodes in the affected group(s).

Crucially, the GMS is not aware of regions; it maintains conventional per-group lists of nodes, and sends per-group updates when membership changes. For example, if node  $n_{55}$  joins group  $A$ , the update sent by the GMS to every node in  $A$  would be a 3-tuple: (*Join*,  $A$ ,  $n_{55}$ ). Individual nodes process these updates to construct multi-group views relative to their own membership.

Since the GMS does not maintain region data, it has to scale only in the number of groups in the system; this can be easily done by partitioning the centralized service on group id and running each partition on a different server. For instance, one machine is responsible for groups  $A$  and  $B$ , another for  $C$  and  $D$ , and so on. Similarly, the FD can be partitioned on a topological criterion; one machine on each rack is responsible for

monitoring other nodes on the rack by pinging them periodically. For fault-tolerance, each partition of the GMS can be replicated on multiple machines using a strongly consistent protocol like Paxos. The FD can have a hierarchical structure to recover from failures; a smaller set of machines ping the per-rack failure detectors, and each other in a chain.

While such a semi-centralized solution for group membership is appropriate and sufficient for most datacenter environments, it is still preferable for mission-critical applications to have a distributed protocol with no single point of failure or overload. Accordingly, the latest iteration of Ricochet runs over a gossip-based failure detection protocol identical to the one described by van Renesse [85]. In addition to gossiping basic I-am-alive messages, the failure detector allows each node to insert arbitrary buffers to distribute to other nodes. Consequently, the GMS is built over this failure detector by having each node insert the set of groups it belongs to. As the failure detection layer gossips information around the system regarding the membership set of each node, the GMS layer collects this information and organizes it into multi-group views that can be used by Ricochet.

Importantly, Ricochet itself does not need consistent membership and degrades gracefully with the degree of inconsistency in the views; if a failed node is included in a view, performance will dip fractionally in all the groups it belongs to as the repairs sent to it by other nodes are wasted. Since connectivity and membership are typically stable in datacenters, we expect views to be only marginally out-of-sync with the real membership of the group.

### 4.5.5 Performance

Since Ricochet creates LEC information from each incoming data packet, the critical communication path that a data packet follows within the protocol is vital in determining eventual recovery times and the maximum sustainable throughput. XORs are computed in each repair bin incrementally, as packets are added to the bin. A crucial optimization used is pre-computation of the number of destinations that the repair bin sends out a repair to, across all the regions that it sends repairs to: Instead of constructing a repair and deciding on the number of destinations once the bin fills up, the repair bin precomputes this number and constructs the repair only if the number is greater than 0. When the bin overflows and clears itself, the expected number of destinations for the next repair packet is generated. This restricts the average number of two-input XORs per data packet to  $c$  (from the rate-of-fire) in the worst case — which occurs when no single repair bin selects more than 1 destination, and hence each outgoing repair packet is a unique XOR.

### 4.5.6 Buffering and Loss Control

LEC - like any other form of FEC - works best when losses are not in concentrated bursts. Ricochet maintains an application-level buffer with the aim of minimizing in-kernel losses, serviced by a separate thread that continuously drains packets from the kernel. If memory at end-hosts is constrained and the application-level buffer is bounded, we use customized packet-drop policies to handle overflows: a randomly selected packet from the buffer is dropped and the new packet is accommodated instead. In practice, this results in a sequence of almost random losses from the buffer, which are easy to recover using FEC traffic. Whether the application-level buffer is bounded

or not, it ensures that packet losses in the kernel are reduced to short bursts that occur only during periods of overload or CPU contention. We evaluate Ricochet against loss bursts of up to 100 packets, though in practice we expect the kind of loss pattern shown in 4.1, where few bursts are greater than 20-30 packets, even with highly concentrated traffic spikes.

#### **4.5.7 NAK Layer for 100% Recovery**

Ricochet recovers a high percentage of lost packets via the proactive LEC traffic; for certain applications, this probabilistic guarantee of packet recovery is sufficient and even desirable in cases where data ‘expires’ and there is no utility in recovering it after a certain number of milliseconds. However, the majority of applications require 100% recovery of lost data, and Ricochet uses a reactive NAK layer to provide this guarantee. If a receiver does not recover a packet through LEC traffic within a timeout period after discovery of loss, it sends an explicit NAK to the sender and requests a retransmission. While this NAK layer does result in extra reactive repair traffic, two factors separate it from traditional NAK mechanisms: firstly, recovery can potentially occur very quickly - within a few hundred milliseconds - since for almost all lost packets discovery of loss takes place within milliseconds through LEC traffic. Secondly, the NAK layer is meant solely as a backup mechanism for LEC and responsible for recovering a very small percentage of total loss, and hence the extra overhead is minimal.

### **4.5.8 Optimizations**

As mentioned previously, Ricochet maintains a buffer of unusable repair packets that enable it to utilize incoming repair packets better. If one repair packet is missing exactly one more data packet than another repair packet, and both are missing at least one data packet, Ricochet obtains the extra data packet by XORing the two repair packets. Additionally, the list of unusable repair packets is checked intermittently to see if recent data packet recoveries and receptions have made any old repair packets usable.

### **4.5.9 Message Ordering**

As presented, Ricochet provides multicast reliability but does not deliver messages in the same order at all receivers. We are primarily concerned with building an extremely rapid multicast primitive that can be used by applications that require unordered reliable delivery as well as layered under ordering protocols with stronger delivery properties. For instance, Ricochet can be used as a reliable transport by any of the existing mechanisms for total ordering [31] — in separate work [14], we describe one such technique that predicts out-of-order delivery in datacenters to optimize ordering delays.

## **4.6 Evaluation**

We evaluated our Java implementation of Ricochet on a 64-node cluster, comprising of four racks of 16 nodes each, interconnected via two levels of switches. Each node has a single 1.3 GHz CPU with 512 Mb RAM, runs Linux 2.6.12 and has two 100 Mbps network interfaces, one for control and the other for experimental traffic. Typical

socket-to-socket latency within the cluster is around 50 microseconds. In the following experiments, for a given loss rate  $L$ , three different loss models are used:

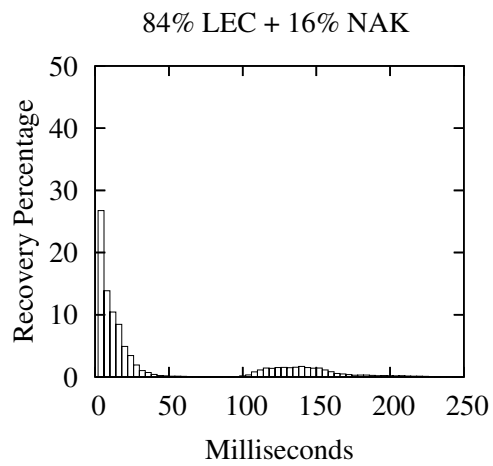
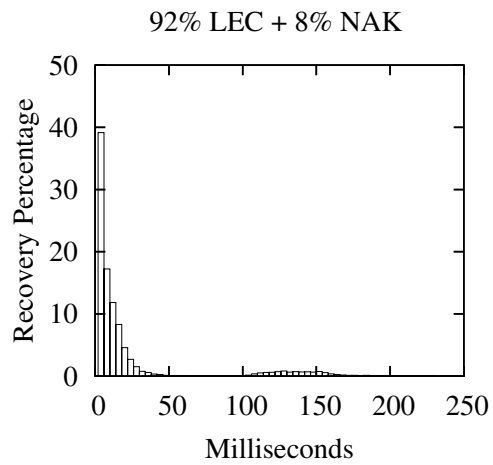
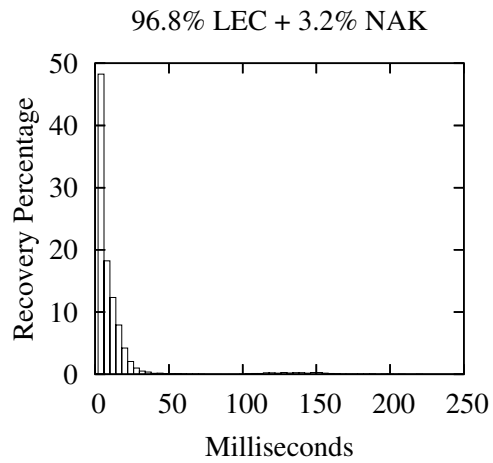
- **uniform** - also known as the Bernoulli model [94] - refers to dropping packets with uniform probability equal to the loss rate  $L$ .
- **bursty** involves dropping packets in equal bursts of length  $b$ . The probability of starting a loss burst is set so that each burst is of exactly  $b$  packets and the loss rate is maintained at  $L$ . This is not a realistic model but allows us to precisely measure performance relative to specific burst lengths.
- **markov** drops packets using a simple 2-state markov chain, where each node alternates between a lossy and a lossless state, and the probabilities are set so that the average length of a loss burst is  $m$  and the loss rate is  $L$ , as described in [94].

In experiments with multiple groups, nodes are assigned to groups at random, and the following formula is used to relate the variables in the grouping pattern:  $n * d = g * s$ , where  $n$  is the number of nodes in the system (64 in most of the experiments),  $d$  is the degree of membership, i.e. the number of groups each node joins,  $g$  is the total number of groups in the system, and  $s$  is the average size of each group. For example, in a 16-node setting where each node joins 512 groups and each group is of size 8,  $g$  is set to  $\frac{16 * 512}{8} \approx 1024$ . Each node is then assigned to 512 randomly picked groups out of 1024. Hence, the grouping patterns for each experiment is completely represented by a  $(n, d, s)$  tuple.

For every run, we set the sending rate at a node such that the total system rate of incoming messages is 64000 packets per second, or 1000 packets per node per second. Data packets are 1K bytes in size. Each point in the following graphs - other than Figure 4.8, which shows distributions for single runs - is an average of 5 runs. A run lasts 30 seconds and produces  $\approx 2$  million receive events in the system.

### 4.6.1 Distribution of Recoveries in Ricochet

First, we provide a snapshot of what typical packet recovery timelines look like in Ricochet. Earlier, we made the assertion that Ricochet discovers the loss of almost all packets very quickly through LEC traffic, recovers a majority of these instantly and recovers the remainder using an optional NAK layer. In Figure 4.8, we show the histogram of packet recovery latencies for a 16-node run with degree of membership  $d = 128$  and group size  $s = 10$ . We use a simplistic NAK layer that starts unicasting NAKs to the original sender of the multicast 100 milliseconds after discovery of loss, and retries at 50 millisecond intervals. Figure 4.8 shows three scenarios: under uniform loss rates of 10%, 15%, and 20%, different fractions of packet loss are recovered through LEC and the remainder via reactive NAKs. These graphs illustrate the meaning of the LEC recovery percentage: if this number is high, more packets are recovered very quickly without extra traffic in the initial segment of the graphs, and less reactive overhead is induced by the NAK layer. Importantly, even with a recovery percentage as low as 84% in Figure 4.8(bottom), we are able to recover all packets within 250 milliseconds with a crude NAK layer due to early LEC-based discovery of loss. For the remaining experiments, we will switch the NAK layer off and focus solely on LEC performance; also, since we found this distribution of recovery latencies to be fairly representative, we present only the percentage of lost packets recovered using LEC and the average latency of these recoveries. Experiment Setup: ( $n = 16, d = 128, s = 10$ ), Loss Model: Uniform, [10%, 15%, 20%].



**Top: 10% Loss Rate**

**Middle: 15% Loss Rate**

**Bottom: 20% Loss Rate**

Figure 4.8: Distribution of Recoveries: LEC + NAK for varying degrees of loss

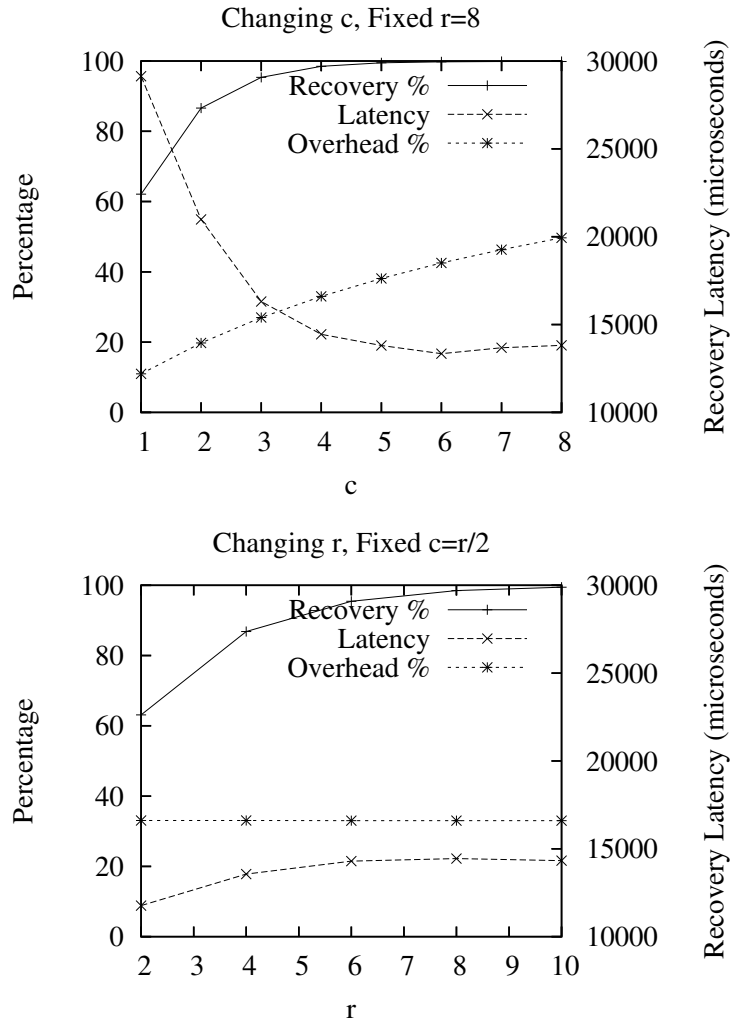


Figure 4.9: Tuning LEC : tradeoff points available between recovery %, overhead % (left y-axis) and avg recovery latency (right y-axis) by changing the rate-of-fire ( $r, c$ ).

### 4.6.2 Tunability of LEC in multiple groups

Figure 4.9 shows that the rate-of-fire parameter ( $r, c$ ) provides a knob to tune LEC’s recovery characteristics. In Figure 4.9 (Top), we can see that increasing the  $c$  value for constant  $r = 8$  increases the recovery percentage and lowers recovery latency by expending more overhead - measured as the percentage of repair packets to all packets. In Figure 4.9 (Bottom), we see the impact of increasing  $r$ , keeping the ratio of  $c$  to  $r$  - and consequently, the overhead - constant. For the rest of the experiments, we set the

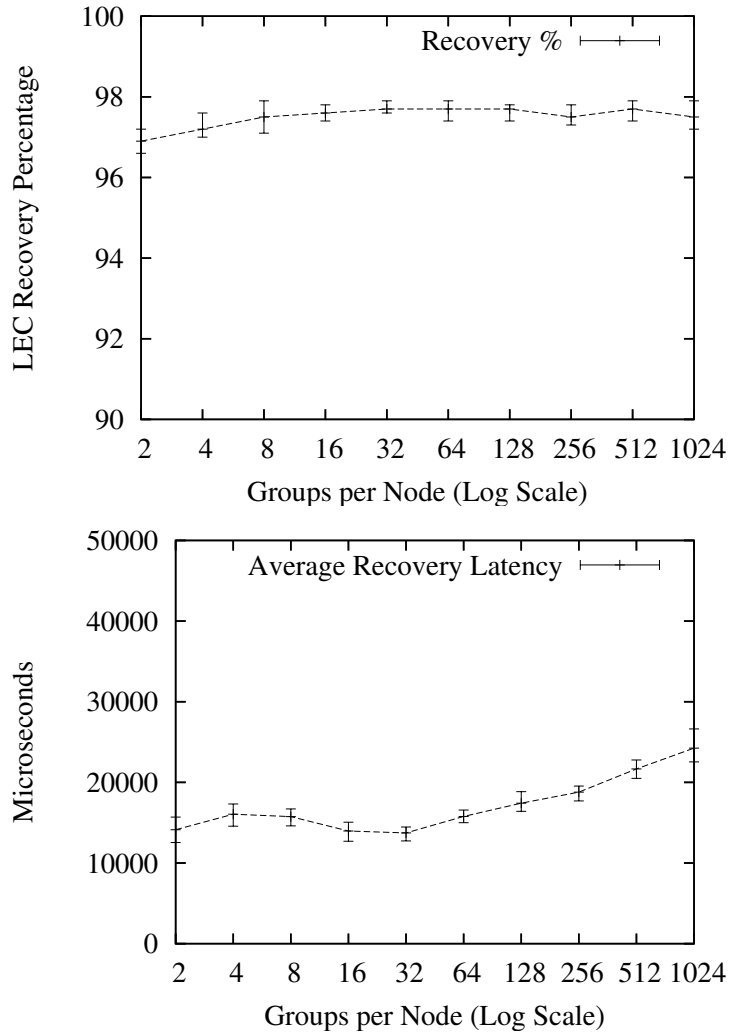


Figure 4.10: Scalability in Groups

rate-of-fire at ( $r = 8, c = 5$ ). Experiment Setup: ( $n = 64, d = 128, s = 10$ ), Loss Model: Uniform, 1%.

### 4.6.3 Scalability

Next, we examine the scalability of Ricochet to large numbers of groups. Figure 4.10 shows that increasing the degree of membership for each node from 2 to 1024 has almost no effect on the percentage of packets recovered via LEC, and causes a slow increase in

average recovery latency. The x-axis in these graphs is log-scale, and hence a straight line increase is actually logarithmic with respect to the number of groups and represents excellent scalability. The increase in recovery latency towards the right side of the graph is due to Ricochet having to deal internally with the representation of large numbers of groups; we examine this phenomenon later in this section.

For a comparison point, we refer readers back to SRM's discovery latency in Figure 4.2: in 128 groups, SRM discovery took place at 9 seconds. In our experiments, SRM recovery took place roughly 4 seconds after discovery in all cases. While fine-tuning the SRM implementation for clustered settings should eliminate that 4 second gap between discovery and recovery, at 128 groups Ricochet surpasses SRM's best possible recovery performance of 5 seconds by between 2 and 3 orders of magnitude.

Though Ricochet's recovery characteristics scale well in the number of groups, it is important that the computational overhead imposed by the protocol on nodes stays manageable, given that time-critical applications are expected to run over it. Figure 4.11 shows the scalability of an important metric: the time taken to process a single data packet. The straight line increase against a log x-axis shows that per-packet processing time increases logarithmically with the number of groups - doubling the number of groups results in a constant increase in processing time. The increase in processing time towards the latter half of the graph is due to the increase in the number of repair bins with the number of groups. While we considered 1024 groups adequate scalability, Ricochet can potentially scale to more groups with further optimization, such as creating bins only for occupied regions. In the current implementation, per-packet processing time goes from 160 microseconds for 2 groups to 300 microseconds for 1024, supporting throughput exceeding a thousand packets per second. Figure 4.11 also shows the average number of XORs per incoming data packet. As stated in section 4.5.2, the number of

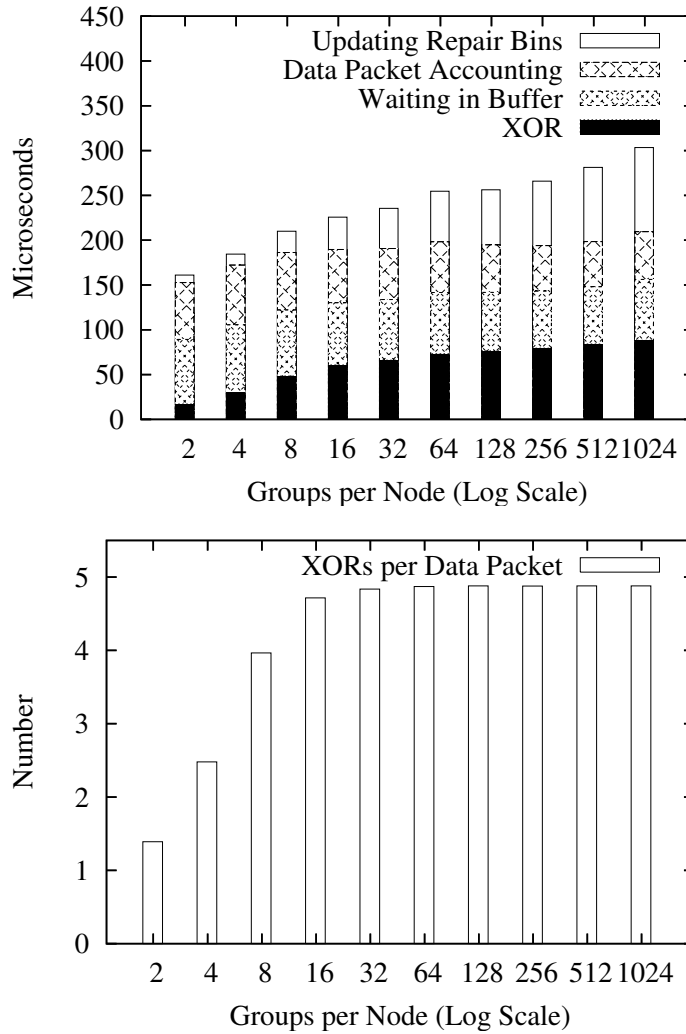


Figure 4.11: CPU time (Top) and XORs (Bottom) per data packet

XORs stays under 5 - the value of  $c$  from the rate-of-fire  $(r, c)$ . Experiment Setup:  $(n = 64, d = *, s = 10)$ , Loss Model: Uniform, 1%.

#### 4.6.4 Loss Rate and LEC Effectiveness

Figure 4.12 shows the impact of the Loss Rate on LEC recovery characteristics, under the three loss models. Both LEC recovery percentages and latencies degrade gracefully: with an unrealistically high loss rate of 25%, Ricochet still recovers 40% of lost packets

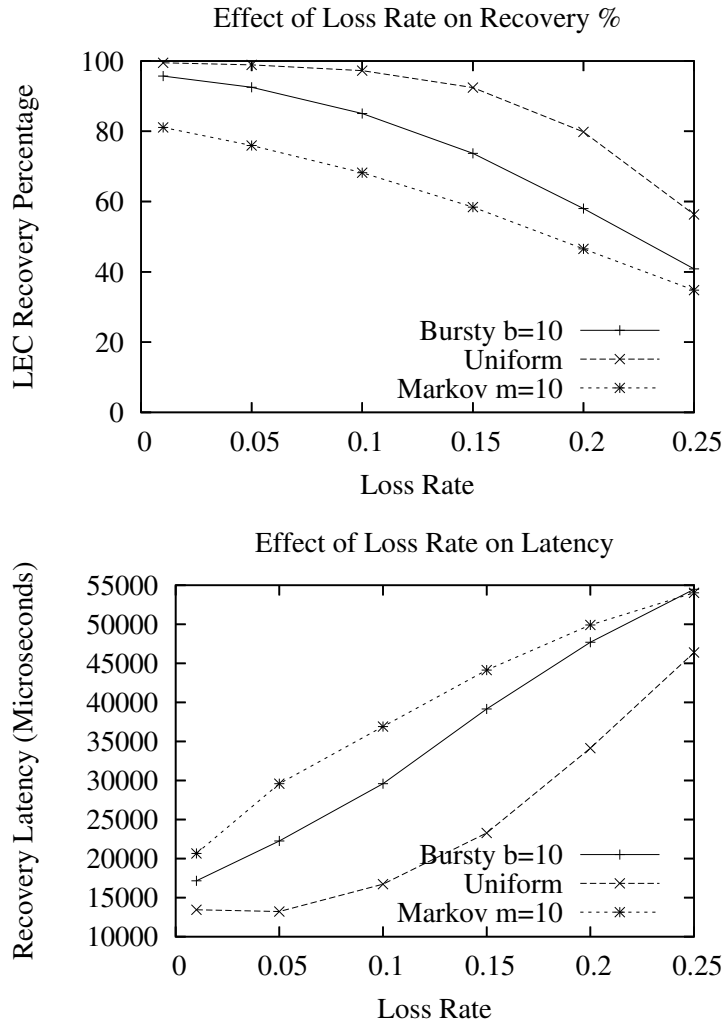


Figure 4.12: Impact of Loss Rate on LEC

at an average of 60 milliseconds. For uniform and bursty loss models, recovery percentage stays above 90% with a 5% loss rate; markov does not fare as well, even at 1% loss rate, primarily because it induces bursts much longer than its average of 10 - the max burst in this setting averages at 50 packets. Experiment Setup: ( $n = 64, d = 128, s = 10$ ), Loss Model: \*.

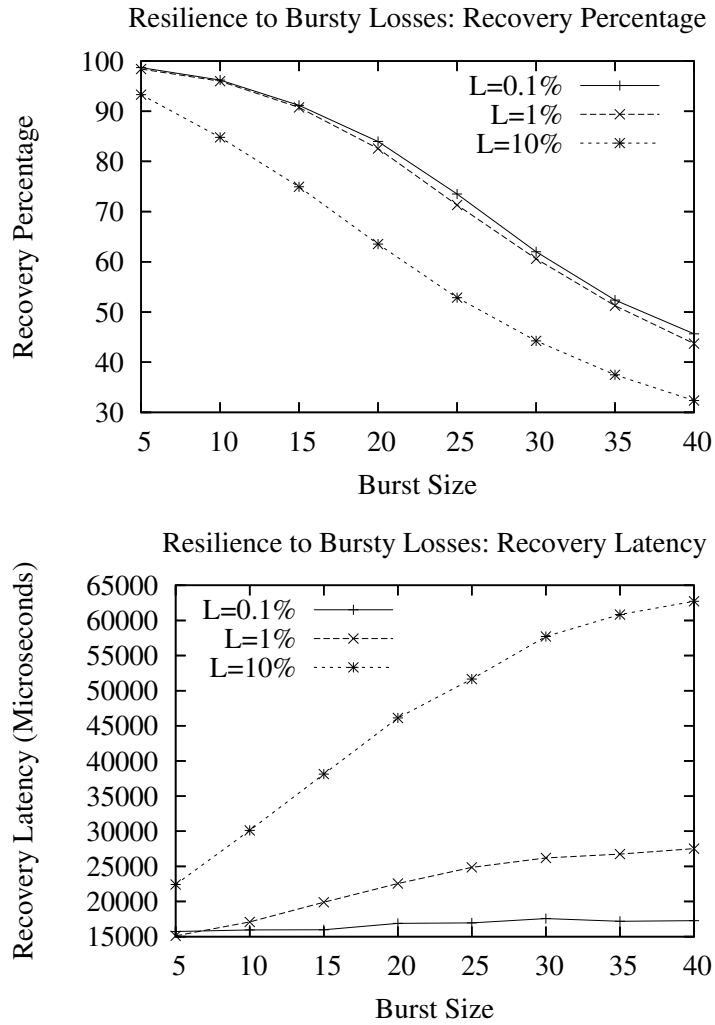


Figure 4.13: Resilience to Burstiness

### 4.6.5 Resilience to Bursty Losses

As we noted before, a major criticism of FEC schemes is their fragility in the face of bursty packet loss. Figure 4.13 shows that Ricochet is naturally resilient to small loss bursts, without the stagger optimization - however, as the burst size increases, the percentage of packets recovered using LEC degrades substantially. Experiment Setup: ( $n = 64, d = 128, s = 10$ ), Loss Model: Bursty.

However, switching on the stagger optimization described in Section 4.5.2 increases

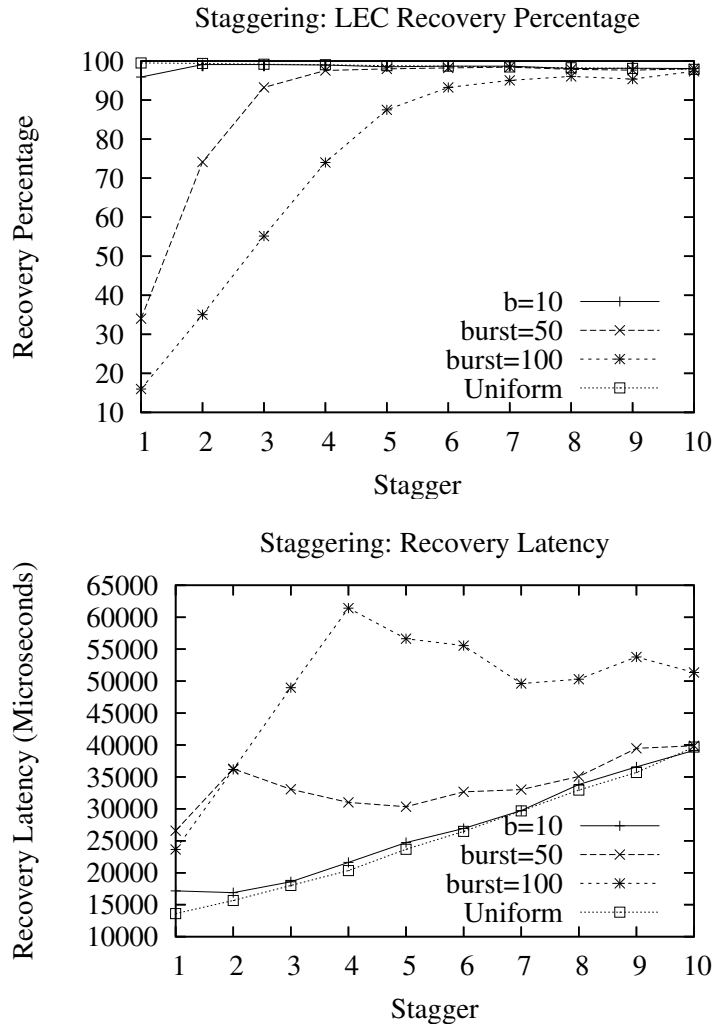


Figure 4.14: Staggering allows Ricochet to recover from long bursts of loss.

Ricochet’s resilience to burstiness tremendously, without impacting recovery latency much. Figure 4.14 shows that setting an appropriate stagger value allows Ricochet to handle large bursts of loss: for a burst size as large as 100, a stagger of 6 enables recovery of more than 90% lost packets at an average latency of around 50 milliseconds. Experiment Setup: ( $n = 64, d = 128, s = 10$ ), Loss Model: Bursty, 1%.

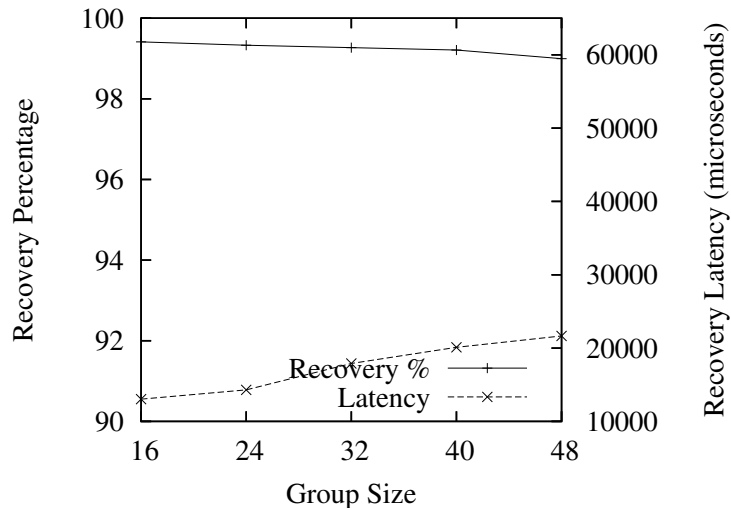


Figure 4.15: Effect of Group Size

#### 4.6.6 Effect of Group and System Size

What happens to LEC performance when the average group size in the cluster is large compared to the total number of nodes? Figure 4.15 shows that recovery percentages are almost unaffected, staying above 99% in this scenario, but recovery latency is impacted by more than a factor of 2 as we triple group size from 16 to 48 in a 64-node setting. Note that this measures the impact of the size of the group relative to the entire system; receiver-based FEC has been shown to scale well in a single isolated group to hundreds of nodes [17]. Experiment Setup: ( $n = 64, d = 128, s = *$ ), Loss Model: Uniform, 1%.

While we could not evaluate to system sizes beyond 64 nodes, Ricochet should be oblivious to the size of the entire system, since each node is only concerned with the groups it belongs to. We ran 4 instances of Ricochet on each node to obtain an emulated 256 node system with each instance in 128 groups, and the resulting recovery percentage of 98% - albeit with a degraded average recovery latency of nearly 200 milliseconds due to network and CPU contention - confirmed our intuition of the protocol's fundamental insensitivity to system size.

## CHAPTER 5

### FUTURE WORK AND CONCLUSION

The datacenter is the focal point of a computing revolution. On the one hand, cloud computing has transitioned overnight from yet another buzzword to a very concrete vision for the future — Amazon (EC2) and Google have already rolled out offerings that open their datacenters to software written by end-users. On the other hand, initiatives at Microsoft and Sun are aimed at producing next-generation hardware platforms for building better physical datacenters.

Despite the trend towards cloud computing, systems support for building scalable applications continues to lag. Modern datacenters glue together commodity operating systems designed for conventional servers with millions of lines of middleware code. A user of Amazon’s EC2 service can remotely operate a ‘virtual’ datacenter with hundreds of machines — but what software does she run on this datacenter?

The next-generation datacenter needs *abstractions*. Developers need tools for building large-scale applications that can run seamlessly on virtualized datacenters while optimizing for a host of concerns, old and new — responsiveness, high availability, power efficiency, security and data privacy.

#### **5.1 Rethinking Transport for the Datacenter**

The Ricochet++ framework is currently in development and provides a superset of the functionality of the original Ricochet protocol. The overarching goal for Ricochet++ is *adaptiveness*. As datacenters get larger and more complex, the ability to selectively switch between protocols — and to dynamically parameterize them — becomes critical for high-performance applications. With the advent of utility computing platforms,

applications are expected to run within highly virtualized containers, contending for processor and network resources. Protocol stacks for such settings will need to be highly adaptive, running different protocols in response to different conditions — for example, alternating between sender-based and receiver-based FEC based on which nodes are more heavily loaded; or using NAKs at high data rates and ACKs at low data rates.

We believe that the key to adaptiveness lies in two important properties: *modularity* provides the *mechanism* of adaptation, while *gray-box design* allows applications to mandate the *policy* of adaptation.

**Modularity:** The Ricochet++ framework is designed to be extremely modular, consisting of pluggable components. These modules interact with each other via a high-speed eventing system that mitigates locking and threading inefficiencies. In addition, Ricochet++ includes a reference-count based memory management subsystem that allows multiple modules to share a single read-only copy of a packet safely and efficiently. Modular design allows applications to switch between different protocols and run protocols in parallel — for example, running an ACK-based flow controller alongside a NAK-based reliability layer.

**Gray-box Exposure:** Network stacks have traditionally been closed ‘black boxes’; the most notable example is TCP/IP, which provides almost no explicit feedback to the end application. Datacenter applications are usually high-performance systems written by intermediate or expert developers. Consequently, we believe that providing fine-grained feedback to applications on network conditions and protocol performance can have enormous advantages. For example, a replicated data store can use information about slow receivers to eject replicas from the group, or to shift load away from the problem receivers to healthy nodes.

## 5.2 The Three P's — Power, Privacy, Parallelism

The next-generation datacenter needs to be energy-efficient, exploit new hardware designs such as multi-core chips, and act in close orchestration with global 'lambda' networks of peer datacenters.

**Lambda Networks:** Lambda Networks provide rich and interesting opportunities — to provide resilience against natural disasters and terrorist attacks by mirroring data to far-away datacenters, to save power and money by locating primary copies of data in these remote datacenters, and ultimately, to create truly global applications that can run seamlessly across geographies. The combination of powerful computing clusters and massive high-speed links opens the door to ubiquitous data access, anytime and anywhere. A potential killer app for lambda networks is IPTV distribution, with datacenters located in different cities interconnected via lambda links, and fiber-to-home providing optical last hops. Such a design raises many research questions — for example, the usage of end-to-end FEC to increase last hop range/receivers, or the implementation of scalable channel switching and video-on-demand.

**Green Systems:** Designs that treat power efficiency as a first-class goal are the need of the hour, both in light of organizational facility budgets and long-term environmental considerations. One approach is to place data and computation intelligently within and across datacenters. For example, the author is collaborating with filesystem researchers to create KyotoFS, a filesystem for highly cache-able workloads that organizes data in a distributed multi-disk log within a datacenter; since all writes go to the head of the log and reads can be serviced by in-memory caches, the disks in the middle and tail of the log can be switched off to save power. A parallel direction of research involves reducing the power consumption of multi-core systems; the availability of power-saving modes

in newer chips allows systems and protocols higher up in the stack to explore trade-offs between performance and energy.

**Datacenter Architecture:** The slowdown in CPU frequency scaling and the emergence of multi-core machines creates new problems and opportunities for datacenters. Current datacenter systems are designed for settings where the unit of scale is an individual processor with dedicated IO bus-lines; in the future, such systems will have walk a tight-rope between intra-machine resource contention and inter-machine communication inefficiencies. Datacenters will evolve into clusters of clusters, requiring parallel scale-up techniques within many-core machines and distributed scale-out techniques between them. Meanwhile, concerns such as power, ISP non-neutrality and privacy are redefining the role of datacenters. A design where the datacenter hosts a control plane but all sensitive data is stored in the client cloud tackles such concerns; but implementing real applications like social networks or e-mail in this fashion is a non-trivial challenge.

To summarize, scale-out architectures have reached an inflexion point in their evolution where they face multiple challenges — real-time coordination across distant datacenters as well as performance, power, privacy and parallelism within datacenters. A promising research agenda for the future includes the design and implementation of systems that explicitly target these goals and effectively use high-bandwidth networks to achieve them.

## BIBLIOGRAPHY

- [1] BEA WebLogic. <http://www.bea.com/products/weblogic/server/index.shtml>, 2008.
- [2] Gemstone Gemfire. <http://www.gemstone.com/products/gemfire/>, 2008.
- [3] Global Crossing Current Network Performance. [http://www.globalcrossing.com/network/network\\_performance\\_current.aspx](http://www.globalcrossing.com/network/network_performance_current.aspx), 2008.
- [4] IBM Websphere. <http://www.ibm.com/websphere/>, 2008.
- [5] JBoss. <http://www.jboss.com>, 2008.
- [6] Netfilter: Firewalling, NAT and Packet Mangling for Linux. <http://www.netfilter.org/>, 2008.
- [7] Oracle Coherence. <http://www.oracle.com/technology/products/coherence/index.html>, 2008.
- [8] Qwest IP Network Statistics. [http://stat.qwest.net/statqwest/statistics\\_tp.jsp](http://stat.qwest.net/statqwest/statistics_tp.jsp), 2008.
- [9] Real-Time Innovations Data Distribution Service. [http://www.rti.com/products/data\\_distribution/index.html](http://www.rti.com/products/data_distribution/index.html), 2008.
- [10] Teragrid. [www.teragrid.org](http://www.teragrid.org), 2008.
- [11] Teragrid UDP Performance. [network.teragrid.org/tgperf/udp/](http://network.teragrid.org/tgperf/udp/), 2008.
- [12] Tibco Rendezvous. <http://www.tibco.com/software/messaging/rendezvous/default.jsp>, 2008.
- [13] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, 1995.
- [14] Mahesh Balakrishnan, Ken Birman, and Amar Phanishayee. PLATO: Predictive Latency-Aware Total Ordering. In *SRDS 2006: IEEE International Symposium on Reliable Distributed Systems*, Leeds, UK, 2006.

- [15] Mahesh Balakrishnan, Ken Birman, Amar Phanishayee, and Stefan Pleisch. Ricochet: Lateral Error Correction for Time-Critical Multicast. In *NSDI 2007: Fourth Usenix Symposium on Networked Systems Design and Implementation*, Boston, MA, 2007.
- [16] Mahesh Balakrishnan, Tudor Marian, Ken Birman, Hakim Weatherspoon, and Einar Vollset. Maelstrom: Transparent Error Correction for Lambda Networks. In *NSDI 2008: Fifth Usenix Symposium on Networked Systems Design and Implementation*, San Francisco, CA, 2008.
- [17] Mahesh Balakrishnan, Stefan Pleisch, and Ken Birman. Slingshot: Time-Critical Multicast for Clustered Applications. In *NCA 2005: 4th IEEE International Symposium on Network Computing and Applications*, Boston, MA, 2005.
- [18] Ernst Biersack. Performance Evaluation of Forward Error Correction in ATM Networks. In *ACM SIGCOMM*, Baltimore, Maryland, 1992.
- [19] Kenneth P. Birman. *Reliable Distributed Systems*. Springer Verlag, 2005.
- [20] Kenneth P. Birman, Mark Hayden, Ozgur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems (TOCS)*, 17(2):41–88, 1999.
- [21] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. *Internet RFC3135*, June 2001.
- [22] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 1995.
- [23] S. Bregni, D. Caratti, and F. Martignon. Enhanced Loss Differentiation Algorithms for Use in TCP Sources over Heterogeneous Wireless Networks. In *IEEE GLOBECOM 2003*, San Francisco, CA, 2003.
- [24] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A Digital Fountain Approach to Reliable Distribution of Bulk Data. In *ACM SIGCOMM*, Vancouver, Canada, 1998.
- [25] Rajiv Chakravorty, Sachin Katti, Ian Pratt, and Jon Crowcroft. Flow Aggregation for Enhanced TCP over Wide Area Wireless. In *IEEE INFOCOM*, San Francisco, CA, 2003.

- [26] Yatin Chawathe, Steven McCanne, and Eric A. Brewer. RMX: Reliable Multicast for Heterogeneous Networks. In *IEEE INFOCOM*, Tel Aviv, Israel, 2000.
- [27] Yang Chu, Sanjay Rao, Srinivasan Seshan, and Hui Zhang. Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture. In *ACM SIGCOMM*, San Diego, CA, 2001.
- [28] William G. Cochran. *Sampling Techniques, 3rd Edition*. John Wiley, 1977.
- [29] Doug Comer, Vice President of Research and Tom Boures, Senior Engineer. Cisco Systems, Inc. *Private Communication.*, October 2007.
- [30] Stephen E. Deering and David R. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computers Systems (TOCS)*, 8(2):85–110, 1990.
- [31] Xavier Défago, André Schiper, and Péter Urbán. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Computing Surveys*, 36(4):372–421, December 2004.
- [32] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *PODC 1987: Sixth Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada, 1987.
- [33] Bill Devlin, Jim Gray, Bill Laing, and George Spix. Scalability Terminology: Farms, Clones, Partitions, Packs, RACS and RAPS. *Microsoft Research Technical Report MSR-TR-99-85*, 1999.
- [34] P. Th. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight Probabilistic Broadcast. *ACM Transactions on Computer Systems (TOCS)*, 21(4):341–374, 2003.
- [35] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking (TON)*, 5(6):784–803, 1997.
- [36] Armando Fox and Eric A. Brewer. Harvest, Yield and Scalable Tolerant Systems. In *HotOS VII: 7th Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, 1999.

- [37] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-Based Scalable Network Services. In *SOSP 1997: Sixteenth ACM Symposium on Operating Systems Principles*, Saint Malo, France, 1997.
- [38] J. Gemmel, T. Montgomery, T. Speakman, N. Bhaskar, and J. Crowcroft. The PGM Reliable Multicast Protocol. *IEEE Network*, 17(1):16–22, 2003.
- [39] G. Gilder. The Information Factories. *Wired Magazine* 14.10.
- [40] Steven D. Gribble, Matt Welsh, Eric A. Brewer, and David E. Culler. The MultiSpace: An Evolutionary Platform for Infrastructural Services. In *USENIX Annual Technical Conference*, Monterey, CA, 1999.
- [41] Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. H. Katz, Z. M. Mao, S. Ross, B. Zhao, and Robert C. Holte. The Ninja Architecture for Robust Internet-Scale Systems and Services. *Computer Networks*, 35(4):473–497, 2001.
- [42] Yunhong Gu and Robert L. Grossman. SABUL: A Transport Protocol for Grid Computing. *Journal of Grid Computing*, 1(4):377–386, 2003.
- [43] R. Habel, K. Roberts, A. Solheim, and J. Harley. Optical Domain Performance Monitoring. In *OFC 2000: The Optical Fiber Communication Conference*, Baltimore, MD, 2000.
- [44] Thomas J. Hacker, Brian D. Athey, and Brian D. Noble. The End-to-End Performance Effects of Parallel TCP Sockets on a Lossy Wide-Area Network. In *IPDPS 2002: International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, 2002.
- [45] Thomas J. Hacker, Brian D. Noble, and Brian D. Athey. The Effects of Systemic Packet Loss on Aggregate TCP Flows. In *Supercomputing '02: ACM/IEEE Conference on Supercomputing*, Baltimore, MD, 2002.
- [46] Eric He, Jason Leigh, Oliver Yu, and Thomas A. DeFanti. Reliable Blast UDP: Predictable High Performance Bulk Data Transfer. In *Cluster 2002: IEEE International Conference on Cluster Computing*, Chicago, IL, 2002.
- [47] Christian Huitema. The Case for Packet Level FEC. In *Fifth International Workshop on Protocols for High-Speed Networks*, Sophia Antipolis, France, 1997.
- [48] Justin Hurwitz and Wu-chun Feng. Initial end-to-end performance evaluation of

- 10-Gigabit Ethernet. In *Hot Interconnects 2003: 11th Symposium on High Performance Interconnects*, Stanford, CA, 2003.
- [49] Internet2. End-to-end performance initiative: Hey! where did my performance go? - rate limiting rears its ugly head. <http://e2epi.internet2.edu/case-studies/UMich/index.html>.
- [50] Internet2. End-to-end performance initiative: When 99% isn't quite enough - educause bad connection. <http://e2epi.internet2.edu/case-studies/EDUCAUSE/index.html>.
- [51] L.B. James, A.W. Moore, M. Glick, and J. Bulpin. Physical Layer Impact upon Packet Errors. In *PAM 2006: Passive and Active Measurement Workshop*, Adelaide, Australia, 2006.
- [52] Minwen Ji, Alistair Veitch, and John Wilkes. Seneca: Remote Mirroring Done Write. In *Usenix Technical Conference*, June 2003.
- [53] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *ACM SIGCOMM*, Pittsburgh, PA, 2002.
- [54] D. C. Kilper, R. Bach, D. J. Blumenthal, D. Einstein, T. Landolsi, L. Ostar, M. Preiss, and A. E. Willner. Optical Performance Monitoring. *Journal of Lightwave Technology*, 22(1):294–304, 2004.
- [55] T. Kim, S. Lu, and V. Bharghavan. Improving Congestion Control Performance Through Loss Differentiation. In *ICCCN 1999: The 8th International Conference on Computer Communications and Networks*, Boston, MA, 1998.
- [56] Andreas Kimsas, Harald Øverby, Steinar Bjornstad, and Vegard L. Tuft. A Cross Layer Study of Packet Loss in All-Optical Networks. In *AICT/ICIW*, Guadelope, French Caribbean, 2006.
- [57] T.V. Lakshman and U. Madhow. The Performance of TCP/IP for Networks with High Bandwidth-Delay Products and Random Loss. *IEEE/ACM Transactions on Networking (TON)*, 5(3):336–350, 1997.
- [58] John C. Lin and Sanjoy Paul. RMTP: A Reliable Multicast Transport Protocol. In *IEEE INFOCOM*, San Francisco, CA, 1996.
- [59] Michael Luby. LT codes. In *FOCS 2002: The 43rd Annual IEEE Symposium on Foundations of Computer Science*, Vancouver, BC, 2002.

- [60] Michael Luby, Michael Mitzenmacher, Mohammad Amin Shokrollahi, and Daniel A. Spielman. Efficient Erasure Correcting Codes. *IEEE Transactions on Information Theory*, 47(2):569–584, 2001.
- [61] H. Lundqvist and G. Karlsson. TCP with End-to-End Forward Error Correction. In *IZS 2004: International Zurich Seminar on Communications*, Zurich, Switzerland, 2004.
- [62] Petar Maymounkov and David Mazieres. Rateless Codes and Big Downloads. In *IPTPS 2003*, Berkeley, CA, 2003.
- [63] M. Meiss. Tsunami: A High-Speed Rate-Controlled Protocol for File Transfer. "<http://steinbeck.ucs.indiana.edu/~mmeiss/papers/tsunami.pdf>".
- [64] M. Meyer, J. Sachs, and M. Holzke. Performance Evaluation of a TCP Proxy in WCDMA Networks. *IEEE Wireless Communications*, 10(5):70–79, 2003.
- [65] T. Montgomery, B. Whetten, M. Basavaiah, S. Paul, N. Rastogi, J. Conlan, and T. Yeh. The RMTP-II protocol, April 1998. IETF Internet Draft.
- [66] L. Munoz, M. Garcia, J. Choque, R. Aguero, and P. Mahonen. Optimizing Internet Flows over IEEE 802.11b Wireless Local Area Networks: A Performance-Enhancing Proxy Based on Forward Error Correction. *IEEE Communications Magazine*, 39(12):60–67, 2001.
- [67] Jorg Nonnenmacher, Ernst Biersack, and Don Towsley. Parity-Based Loss Recovery for Reliable Multicast Transmission. In *ACM SIGCOMM*, Cannes, France, 1997.
- [68] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. *ACM SIGCOMM Computer Communication Review*, 28(4):303–314, 1998.
- [69] Kihong Park and Wei Wang. AFEC: An Adaptive Forward Error Correction Protocol for End-to-End Transport of Real-Time Traffic. In *ICCCN 1998: The 7th International Conference on Computer Communications and Networks*, Lafayette, LA, 1998.
- [70] C. Parsa and J. J. Garcia-Luna-Aceves. Differentiating Congestion vs. Random Loss: A Method for Improving TCP Performance over Wireless Links. In *WCNC 2000: The 2nd IEEE Wireless Communications and Networking Conference*, Chicago, IL, 2000.

- [71] R. Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. SnapMirror: File-System-Based Asynchronous Mirroring for Disaster Recovery. In *FAST 2002: 1st USENIX Conference on File and Storage Technologies*, Monterey, CA, 2002.
- [72] Dan Pritchett. BASE: An ACID Alternative. *ACM Queue*, 6(3), 2008.
- [73] I. Rhee and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. In *PFLDNet 2005: Third International Workshop on Protocols for Fast Long-Distance Networks*, Lyon, France, 2005.
- [74] Luigi Rizzo. Effective Erasure Codes for Reliable Computer Communication Protocols. *ACM SIGCOMM Computer Communication Review*, 27(2):24–36, 1997.
- [75] Luigi Rizzo. On the Feasibility of Software FEC. *Università di Pisa DEIT Technical Report LR-970116*, January 1997.
- [76] Luigi Rizzo. Dummynet and Forward Error Correction. In *USENIX Annual Technical Conference*, New Orleans, LA, 1998.
- [77] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.
- [78] Donna Scott. Survey shows the RTI Journey Continues. Gartner Research, ID G00146683, April 2007.
- [79] N. Shacham and P. McKenney. Packet Recovery in High-Speed Networks using Coding and Buffer Management. In *IEEE INFOCOM*, San Francisco, CA, 1990.
- [80] Amin Shokrollahi. Raptor codes. *IEEE Transactions on Information Theory*, 52(6):2551–2567, 2006.
- [81] H. Sivakumar, S. Bailey, and R. L. Grossman. Pockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks. In *Supercomputing '00: ACM/IEEE Conference on Supercomputing*, Dallas, TX, 2000.
- [82] T. Speakman, J. Crowcroft, J. Gemmell, D. Farinacci, S. Lin, D. Leshchiner, M. Luby, T. Montgomery, L. Rizzo, A. Tweedly, N. Bhaskar, R. Edmonstone, R. Sumanasekera, and L. Vicisano. PGM Reliable Transport Protocol Specification. *Internet RFC3208*, December 2001.

- [83] Lakshminarayanan Subramanian, Ion Stoica, Hari Balakrishnan, and Randy H. Katz. OverQoS: An Overlay based Architecture for Enhancing Internet QoS. In *NSDI 2004: First Usenix Symposium on Networked Systems Design and Implementation*, San Francisco, CA, 2004.
- [84] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf-The TCP/UDP Bandwidth Measurement Tool. <http://dast.nlanr.net/Projects/Iperf>, 2004.
- [85] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A Gossip-Based Failure Detection Service. In *Middleware 1998: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Lake District, England, September 1998.
- [86] D. Velenis, D. Kalogeras, and B. Maglaris. SaTPEP: a TCP Performance Enhancing Proxy for Satellite Links. In *IFIP Networking 2002*, Pisa, Italy, 2002.
- [87] Paulo Verissimo and Luis Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.
- [88] S. Wallace. Tsunami File Transfer Protocol. In *PFLDNet 2003: First International Workshop on Protocols for Fast Long-Distance Networks*, Geneva, Switzerland, 2003.
- [89] Paul. Wefel, Network Engineer. The University of Illinois' National Center for Supercomputing Applications (NCSA). *Private Communication*, Feb 2008.
- [90] D.X. Wei, C. Jin, S.H. Low, and S. Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking (TON)*, 14(6):1246–1259, 2006.
- [91] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI 2002: Fifth Usenix Symposium on Operating Systems Design and Implementation*, Boston, MA, 2002.
- [92] S.B. Wicker and V.K. Bhargava. *Reed-Solomon Codes and Their Applications*. John Wiley & Sons, Inc. New York, NY, USA, 1999.
- [93] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary Increase Congestion Con-

trol (BIC) for Fast Long-Distance Networks. In *IEEE INFOCOM*, Hong Kong, China, 2004.

- [94] Maya Yajnik, Sue B. Moon, James F. Kurose, and Donald F. Towsley. Measurement and Modeling of the Temporal Dependence in Packet Loss. In *IEEE INFOCOM*, New York, NY, 1999.