Robust Peer-to-peer Protocols via Randomized Forwarding

by

Adam Michael Costello

B.S. (Washington University in St. Louis) 1993
M.S. (Washington University in St. Louis) 1996

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Randy H. Katz, Chair
Professor Eric Brewer
Professor Rainer Sachs

Fall 2005

The dissertation of Adam Michael Costello is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2005

Robust Peer-to-peer Protocols via Randomized Forwarding

Abstract

Robust Peer-to-peer Protocols via Randomized Forwarding

by

Adam Michael Costello

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Randy H. Katz, Chair

Forwarding is the relaying of a message from one machine ("node") to another toward a destination, to provide a communication path between non-adjacent nodes. Forwarding services are traditionally designed to construct optimal paths, detect faults, then reconstruct the paths to route around the faults. Peer-to-peer systems, by using multi-hop forwarding to allow the peers to find each other as needed, enable large-scale reliable services to be built from many small diverse unreliable components, which makes faults more common and less predictable. To let these systems better tolerate faults we introduce *randomized forwarding*—peers retransmit messages along random paths rather than optimal paths, preemptively routing around faults without detecting them or waiting for paths to be reconstructed. We demonstrate this technique in two new protocols, Search Party and Rumor Mill, which use network-layer randomized forwarding to solicit local retransmissions for lost multicast packets; and in a third new protocol, Tangle, which uses application-layer randomized forwarding to find rendezvous points in a distributed hash table. Through analysis and experiments in these different problem domains we show that peer-to-peer

protocols, by using randomized forwarding instead of deterministic forwarding, can pay a small performance penalty in exchange for a great improvement in fault tolerance.

Internet applications have a long history of using both the client-server architecture, where a large central server provides a service to many independent clients, and the peer-to-peer architecture, where many small peers cooperate to provide a service to themselves by acting as both clients and servers to each other. Traditional Internet applications like file transfer, mail, Usenet, the domain name system, and chat have use a hybrid "client-peer" model where many unaffiliated always-running peers interact as both clients and servers to provide a service to an even larger number of short-lived strict clients. Other applications have used a pure peer-to-peer model, like the Mbone conferencing tools that use IP multicast rather than application servers to reach other interested nodes. Other applications, particularly the World Wide Web, have used a pure client-server model. The Web was so explosively successful, and the client-server model suddenly so dominant, that it seemed revolutionary when the peer-to-peer model reemerged with instant messaging and file-sharing systems. Developers and researchers turned to pure peer-to-peer systems to spread out the sometimes prohibitive cost of scaling up centralized services and to eliminate the single point of failure.

Scalability without the need for expensive concentrations of machinery, network connections, and personnel is peer-to-peer's strength, but it can also be a source of weakness. In a system that depends on multi-hop forwarding for scalability, and which is composed of many low-budget poorly maintained components, traditional forwarding services' emphasis on optimal paths can lead to frequent transient outages as faults are continually detected and paths reconstructed, and even to long-lived outages when improvements in automatic fault detection are slow to deploy. For such environments we propose *randomized forwarding* for its greater fault tolerance in normal operation. Routing tables used for traditional deterministic forwarding map a destination address to a single next hop, but routing tables used for randomized forwarding map a destination address to multiple next hops, any of which would make progress toward the destination, and one is chosen at random. Hence several retransmissions of the same message will all take different paths, proactively routing around faults without the need to first detect them.

We demonstrate this technique in the design of three peer-to-peer protocols. Search Party and Rumor Mill recover lost multicast packets by using randomized forwarding at the network layer to solicit retransmissions from members of the multicast group outside the loss region. Previous peer-to-peer multicast loss recovery solutions have organized the members into a hierarchy, relying on a single representative node in the loss region to solicit a retransmission on behalf of all the nodes, thus magnifying the effects of faults in the nodes and edges near the top of the hierarchy. By using randomized forwarding instead of hierarchies, Search Party provably spreads the responsibility equally among all nodes in the loss region, and reduces the average cost of adversary-chosen packet losses from $O(N)$ to $O(\log N)$ (where N is the group size). Simulations show that the cost of this greater robustness is a doubling of the redundancy and delay of retransmissions. Rumor Mill is a more easily deployable follow-on to Search Party that relies on much simpler network-layer support, in exchange for additional overhead and delay by a factor of $O(\log N)$. The third protocol, Tangle, is a distributed hash table, which maintains a map from arbitrary labels to nodes, distributing the map across the nodes and performing lookups by successive approximation along a forwarding path, so that nodes with common interests (represented by labels) can discover each other without depending on any centralized servers. Other distributed hash tables developed concurrently with Tangle use only deterministic forwarding, while Tangle uses a configurable combination of deterministic and randomized forwarding. Our experiments show that the addition of randomized forwarding reduces the frequency of lookup failure by an order of magnitude at a cost of only 1.5x in lookup time, and for other faults like network routing anomalies and application bugs the failure rate is reduced by two to five orders of magnitude at no cost in lookup time.

Professor Randy H. Katz
Dissertation Committee Chair

For Helen

Contents

List of Figures

Acknowledgements

This dissertation was made possible by the help, support, guidance, and encouragement I have received over the years from professors, colleagues, staff, family, and friends, all of whom I wish to thank.

My first research experience was due to Ron Loui, who during my undergraduate years at Washington University in St. Louis enlisted me to implement his defeasible reasoning algorithms. After I graduated, George Varghese took me on as a research assistant and provided clear and enthusiastic teaching, a ready supply of interesting problems, guidance into the world of conferences and publications, and ultimately the confidence to accept his selfless advice to leave the comfort zone of Wash.U. with a master's degree and seek a doctoral program at a top-tier school. During my first year at UC Berkeley Tom Anderson made a place for me in the NOW project, and Eric Brewer served as my temporary advisor, until I found a research topic with the help of Steve McCanne. As my new advisor, he provided invaluable feedback, suggestions, pointers, and insights, until he had to leave academia to answer his true calling as a founder of network technology startups. At this point, Randy Katz, already extremely busy with many students of his own, generously agreed to "adopt" me, and saw me through to completion (a test of patience neither of us anticipated!). In this latter period I also received unofficial guidance from Ion Stoica. I am also thankful to Anthony Joseph, Charles Stone, and Rainer Sachs, for serving on my qualifying exam committee and my dissertation committee; to Christos Papadopoulos, Sylvia Ratnasamy, and Ben Zhao for making themselves available to me for open discussions of our related work; to Katherine Crabtree, Peggy Lau, and La Shana Porlaris for navigating the university bureaucracy on my behalf; and to Eric Fraser and Keith Sklower for their responsive system administration and for treating me as a collaborator, not just a user.

The long road to this degree was sometimes straight and sometimes crooked, sometimes quick and oftentimes glacial, but my spirits were always kept high by dear friendships. For dinners, picnics, travel, show-and-tell, games, carnivals, karaoke, Dance Dance Revolution, climbing, costume parties, fireworks, easter-egg hunts, New Year's reunions, Buffy screenings, weddings, and countless other fond memories, I am very grateful to Amy Hsu, Andy Begel, Angie Schuett, Cathy Huang, Chen-Nee Chuah, Chris Dunphy, Dan Garcia, Dan Jiang, Dan Rice, David Gay, David Margolis, David Oppenheimer, Debbie Rosenthal, Diana Laulainen-Schein, Diane Pietrzak, Drew Roselli, Eric Stuebe, Erick Davidson, Ian Flanigan, Jeanna Neefe Matthews, Jing Zhu, Joyce Feng, Ken Krieger, Lilly Tao, Lucy Lu, Marie Arnou, Mark Spiller, Marlene Wan, Megan Thomas, Melanie Leitner, Miranda Capra, Morley Mao, Munazza Bukhari, Nancy Chang, Olga Popova, Pat Niemeyer, Pete McCann, Rob Capra, Seth Golub, Sharad Agarwal, Steve Fink, Susan Dick, Tao Ye, Ted Yu, Tina Wong, Tom Zambito, Toni Reiss, Veronica Lim, Weidong Cui, and Yatin Chawathe.

My ability to meet the challenges of starting and finishing this doctorate rests on a foundation of love and support from my family—Marty, Wendy, and Jill—who have always believed in me, and who have my love and gratitude. I also wish to thank a member of my extended family, Hedda Herzog, for welcoming me to Berkeley and providing friendship and moral support ever since.

Finally, there is one person who more than any other has enabled me to finish this work. Helen Wang, the girl next door (in 473 Soda), has been my closest friend, my most passionate supporter, my most inspirational coach, and my greatest source of strength. This dissertation, and my heart, are dedicated to her.

Chapter 1

Introduction

Forwarding is the relaying of a message from one machine ("node") to another toward a destination, to provide a communication path between non-adjacent nodes. For example, Internet routers forward packets from one local-area network to another, along multi-hop paths, so that any two Internet hosts can exchange messages. Forwarding can also be useful at higher layers, where an entire network-layer path is viewed as a single hop in a forwarding chain, and two nodes are considered adjacent if either knows the other's network address. Traditional forwarding services have been designed for performance, striving to construct paths that are the shortest, fastest, cheapest, or best in some other respect. Faults are handled as an exceptional case: links are monitored for some set of detectable faults, and when a faulty link is discovered, new paths are constructed to route around it. This approach works well in systems where one can assume that faults are well understood and relatively rare, but those assumptions break down for Internet applications based on the increasingly popular peer-to-peer architecture. We outline this trend in Section 1.1, then in Section 1.2 propose *randomized forwarding* as a more fault-tolerant design for forwarding in peer-to-peer protocols. Section 1.3 provides an overview of our design and evaluation of three new peer-to-peer protocols based on randomized forwarding. The organization of the rest of the dissertation appears in Section 1.4.

1.1 Fall and Rebirth of Peer-to-Peer on the Internet

There are two basic models for architecting network applications: client-server and peer-to-peer [40]. In the client-server model, a single large server or a few affiliated servers provide a service to many small independent clients. The clients communicate with the servers, not with each other. In the peer-to-peer model, many small peers provide a service to themselves by cooperating and sharing resources like storage, computation, and network bandwidth. Any peer may act both as a client by requesting service from another peer, and as a server by providing service to another peer.

For a long time it has been common for Internet applications to use a hybrid of both models. Many traditional Internet applications from before 1990 could be considered "client-peer", with many unaffiliated peers interacting as both servers and clients to provide a service to an even larger number of strict clients. The peers are normally expected to be running all the time, unlike the strict clients, which are routinely started and stopped by user actions. The client-peer pattern can be seen in the following examples. In the File Transfer Protocol (FTP) [49] there are many FTP servers and a larger number of FTP clients. A client can tell a server to transfer files to and from other servers (although this capability is not often used—usually the transfers are between the client and the server). In electronic mail [48] there are many mail transfer agents (MTAs) and a larger number of mail user agents (MUAs). The MUAs connect to the MTAs to submit and retrieve mail messages, while the MTAs relay the messages to other MTAs logically closer to the destination mail address, and also expand mail aliases (for mail forwarding and mailing lists). In Usenet [50] there are many news servers and a larger number of news readers. A news reader connects to a nearby news server to retrieve articles of interest to its user and to post articles written by its user to logical newsgroups, while the news servers relay the articles to all other news servers that opt to receive those newsgroups, along a graph of peer relationships. In the Domain Name System (DNS) [47] there are many name servers and a larger number of resolvers. The resolvers contact name servers asking for information associated with a domain name (typically the IP address of a host name), and the name servers recursively contact other name servers logically closer to the name of interest, until the answer is found, cached at all the name servers in the chain, and returned to the resolver. DNS illustrates the fact that peer-to-peer can still be somewhat centralized—the name servers are arranged in a hierarchy, and if the root servers disappear, the whole system fails. In Internet Relay Chat (IRC) [51] there are many IRC servers connected in an acyclic graph, and a larger number of IRC clients. Each client connects to any server and sends chat messages to a logical channel, and the servers relay the message through the graph to all other clients subscribed to that channel.

In contrast to the client-peer pattern, some applications use a pure peer-to-peer model. For example, the first half of the 1990s saw the introduction of IP multicast [10], a forwarding service that allows packets to be sent not to an explicit destination machine but rather to an abstract group address, which the network maps to all machines that have subscribed to that address. Several peer-to-peer conferencing applications, collectively known as the Mbone tools [14], were developed based on IP multicast, including video tools (nv, ivs, vic), audio tools (vat, rat), shared whiteboard (wb), and a session directory (sdr). Instead of clients connecting to servers to have their messages relayed to a set of other interested clients at the application layer, as in mailing lists, Usenet newsgroups, and IRC channels, here we have peers communicating directly with each other by listening on and sending to multicast group addresses. Sometimes IP packets are dropped in the network, and when the dropped packet is a multicast packet, it causes a subset of the group members not to receive that packet. For applications that need reliable packet delivery, like shared whiteboard, a retransmission protocol is needed. Given the prevailing peer-to-peer architecture of the multicast applications, it was natural to consider peer-to-peer approaches, where any peer that received the packet can potentially provide a retransmission for peers that did not, as opposed to a client-server approach, where the node that sent the original packet acts as the sole retransmitter for that packet. Several peer-to-peer multicast loss recovery protocols were proposed, as we discuss in Chapter 2.

At the same time that the peer-to-peer multicast work was proceeding, an unrelated application was being developed using a pure client-server model: the World Wide Web [2], designed to enable users to easily browse remote hypertext documents and follow links between documents. The Web is both distributed and centralized—there are many unaffiliated web servers, but for any given web site (set of jointly controlled of web pages/services) there is one central server (or a few replicated servers). Web servers generally do not cooperate with other web servers (except for web caches, which are a small minority of web servers); the model is client-server.

In the latter 1990s the Web exploded in popularity and dwarfed (in terms of network traffic and public awareness) all other Internet applications. Because of the Web's client-server architecture, very popular web sites, striving to keep up with demand and meet expectations of high availability, require powerful computers, thick network connections, and professional maintenance staff, all of which are expensive. Some extreme examples are large shopping sites, which allow millions of users to search and purchase millions of products, and web search engines, which make centralized copies of the Web, index them, and allow millions of users to search them. Traditional Internet applications had tended to distribute the cost of a service among many maintainers of smaller servers, but in the newly commercialized Internet funds were available to support a variety of large-scale centralized Web-based services.

The explosion of the Web accompanied the exponential growth of the Internet itself, so that soon most Internet users and providers thought of the Internet primarily in terms of the client-server model. Therefore the resurgence of the peer-to-peer model in the late 1990s seemed revolutionary. Instant messaging protocols that track and connect individual users existed earlier (Zephyr [12] was used at MIT in 1987), but they became known to the new booming Internet populace starting with ICQ [22] in 1996, which used both the client-server and peer-to-peer models: a central server kept track of which user agents were running and what hosts they were running on, and referred them to each other, but the agents connected to each other directly to exchange text messages and streaming audio and video. In 1999 the peer-to-peer model was brought to center stage by the enormous popularity of the music-sharing application Napster [61], which used a very similar model: a central server kept track of which peers were online and what music files they had, and answered search queries from peers with referrals to other peers holding matching music files, which were then transferred directly from peer to peer. If we described the traditional Internet applications as “client-peer”, we might describe these newer applications as “peer-server”, with many peers acting as both clients and servers to each other while relying on a strict server for centralized coordination of metadata.

Even a server that handles only metadata faces scaling issues. In a client-server or peer-server system, the servers communicate with all the other nodes (or at least all the active ones), so the addition of more nodes requires a proportional increase in the capacity of the servers. In a pure peer-to-peer system, nodes are not expected to be large enough to communicate with (or even know about) all other nodes; instead, they use multi-hop forwarding to find other nodes as needed, and more peers can be accommodated by increasing the path lengths. Therefore a peer-to-peer architecture allows a large-scale service to be deployed even in situations where it is not feasible to concentrate the machinery, funding, and expertise necessary to maintain a large reliable server. Also, a central server represents a single point of failure, whether from malfunctions, overload, legal action, or malicious attacks. Motivated by these concerns, developers and academics began moving in the direction of more decentralized peer-to-peer solutions in which even the metadata handling is distributed. For example, the Jabber instant messaging system, centered around the Extensible Messaging and Presence Protocol (XMPP) [52], uses many unaffiliated peers, like mail, rather than a central server like ICQ. Decentralized peer-to-peer file sharing protocols like Freenet [5] and Gnutella [60] were developed, as were scalable decentralized peer-to-peer name lookup protocols, known as distributed hash tables, which we discuss in Chapter 2.

1.2 Fault Tolerance via Randomized Forwarding

One of the strengths of peer-to-peer systems—their ability to scale up without the need for an expensive concentration of machinery, network connections, and personnel—can also be a source of weakness. The system may be composed of many low-budget poorly maintained components, and therefore faults are likely to be the rule rather than the exception. Peer-to-peer systems depend on multi-hop forwarding for scalability, but with traditional forwarding services and their emphasis on optimal paths, more frequent faults means more frequent transient outages while the faults are being detected and routed around. Furthermore, without a professional staff measuring, testing, reconfiguring, and upgrading the system, there is a greater risk of unanticipated faults going undetected and causing long-lived outages, with no one to "open a ticket" against. This environment calls for the forwarding services used by peer-to-peer systems to be designed with a greater emphasis on fault tolerance in normal operation.

We propose *randomized forwarding* as a mechanism that can be used to make forwarding more fault tolerant, as we show by example in Chapters 4, 5, and 6. We distinguish randomized forwarding from random repeating. The latter has long been used in *gossip* protocols [13] to quickly disseminate information among all nodes in a fully-connected graph (or one that is trying to become fully connected): whenever a node receives a message it hasn't seen before, it relays the message to a small random set of other nodes in the system. These messages are not moving "forward" to distant nodes, they are moving around among nodes that are (or soon will be) all adjacent to each other. Forwarding, on the other hand, is a mechanism that allows systems to scale beyond the point where fully connected graphs are feasible. Each node has *routing table* that maps destination patterns to next hops. When a message arrives, its destination field is compared with the patterns, and the best matching row is selected. Traditionally, the row contains exactly one next hop (next node or edge in the path), and the message is forwarded there. In randomized forwarding, each row contains multiple next hops, any of which would make progress toward the destination, and one is chosen at random. The original sender of a message, operating above the forwarding layer, uses the randomized forwarding service by continually resending a message until at least one succeeds. If each retransmission took the same path, as in traditional deterministic forwarding, a single fault might easily strike them all, but because each takes a different path, they are less likely to meet the same ill fate. Whereas traditional forwarding detects faults and then routes around them, randomized forwarding preemptively routes around everything, without trying to detect faults and without waiting for new paths to be constructed—an appropriate strategy when faults are very commonplace or difficult to detect.

1.3 Three New Protocols

We demonstrate the use of randomized forwarding for fault tolerance in our design of three peer-to-peer protocols, in order of decreasing network-layer involvement. The first two protocols, *Search Party* and *Rumor Mill*, are built around randomized forwarding at the network layer (random decisions are made inside the network routers), and provide two solutions to the problem of *multicast loss recovery*. In point-to-point communication packet losses can be recovered via retransmissions from the sender, but in one-to-many communication the frequency of losses grows with the group size, so unicast retransmissions from the source will not scale. On the other hand, the average fraction of receivers affected by a packet loss shrinks as the group grows (because most links in the distribution tree are near the leaves), so multicast retransmissions would become less and less efficient. The challenge is to recovery from losses locally, involving a subset of the group commensurate with the subset affected by the loss.

Previous peer-to-peer solutions to this problem have enabled the group members to repair each other's losses by organizing the members into a hierarchy, but hierarchies magnify the adverse effects of faults in the nodes and edges near the top. The multicast distribution tree itself is hierarchical, but it is composed of network routers, which are generally maintained at a much higher standard of reliability than end hosts. Search Party and Rumor Mill provide local recovery without hierarchies of group members, using randomization to prevent faulty members from ruining the system for the others.

Search Party relies on two additional forwarding services in routers, *subcast* and *randomcast*. Subcast, which has been proposed by others in earlier work, forwards packets to all group members downstream from a specified internal point in the distribution tree, and is used for sending retransmissions. Requests for retransmissions are sent via randomcast, which is our proposal to forward a packet to a single randomly-chosen member of the group, usually one near the sender. Through analysis and simulations we demonstrate and quantify Search Party's costs and benefits compared to the hierarchy-based approach. Unlike the latter, where all members affected by a loss depend entirely on a single member to solicit a retransmission on their behalf, Search Party provably divides the responsibility equally among all the members, minimizing the impact of one malfunctioning member on the others. In the hierarchy-based approach, recovery from a loss is typically local, but depending on where losses occur, can be very non-local, involving more members than the loss itself by a factor of $O(N)$ (where N is the group size); but for Search Party we prove that this factor cannot average more than $O(\log N)$ no matter where losses occur. These improvements in worst-case behavior come at the cost of redundancy and delay of retransmissions, a constant factor which simulations show to be about 2.

The deployability of Search Party boils down to the deployability of its two forwarding services, randomcast and subcast. Randomcast involves some additional per-group state in the routers and the ability to modify packets as they are forwarded. Subcast involves one-to-many transmissions from any group member, not just the source, which is technically not a problem but is likely to conflict with policies of wide-area network operators. To address deployment concerns we designed a second solution to the multicast loss recovery problem, Rumor Mill. Instead of randomcast, we propose a much simpler service, *trivial randomcast*, which requires no additional state in routers and no packet modification. Compared to Search Party, Rumor Mill pays some additional overhead and delay, a factor of about $O(\log N)$, to be able to use trivial randomcast instead of randomcast, and unicast instead of subcast, greatly reducing the deployment hurdles.

The third protocol, *Tangle*, requires no network-layer support at all, but is instead built around randomized forwarding at the application layer (each random hop is from one end host to another, along an ordinary network path). Tangle provides a *distributed* *hash table*, which really means a distributed rendezvous service. It maps every possible hash value, called a *label*, to a node in the system (or to a small set of nodes). Because the mapping is updated as nodes come and go, constant labels can be used as rendezvous points for higher-level services, like publish/subscribe. Without needing any particular node to be always present, nodes with a common interest identified by a label can always find each other via the node currently associated with that label. The mapping from labels to nodes is distributed across the nodes and is computed by successive approximation along a forwarding path. Tangle is one of several distributed hash table protocols developed independently and concurrently. Whereas the others construct deterministic routing tables and forward messages along deterministic paths, our design accommodates the unreliability of application-level routers by constructing randomized routing tables and forwarding messages along randomized paths. Experiments show that for a 10% packet loss rate, the use of randomized forwarding in addition to deterministic forwarding reduces the frequency of lookup failure by an order of magnitude, while increasing the average lookup time by only 50%. For various other kinds of faults besides packet loss, like network routing anomalies and application-level software bugs, there is no lookup time penalty at all, and the failure rate reduction is two to five orders of magnitude.

1.4 Organization of the Dissertation

The rest of this dissertation is organized as follows. An overview of related work is given in Chapter 2, with forward references to more focused related work discussions in later chapters. Our general methodology is described briefly in Chapter 3. Chapters 4, 5, and 6 present Search Party, Rumor Mill, and Tangle, respectively. We suggest directions for future work in Chapter 7, and finally conclude that even though the problems of multicast loss recovery and distributed hash tables have little in common besides their suitability to peer-to-peer solutions, analysis and experiments show that in both cases the incorporation of randomized forwarding provides a high degree of fault tolerance in exchange for a small performance penalty compared to conventional deterministic forwarding.

Chapter 2

Background and Related Work

Our work applies two design principles—randomized forwarding and peer-to-peer architecture—to two problem domains—multicast loss recovery and rendezvous services. Related work exists for some combinations of these topics.

Random repeating, a precursor to randomized forwarding, has been used in other problem domains, in the class of peer-to-peer protocols known as group communication, where all members are logically one hop away from all other members (or are trying to become so). Chapter 1 already described gossip protocols [13] for disseminating information throughout the group. A related protocol is Name-Dropper [19], which bootstraps the group—the information being disseminated is the membership of the group itself, and an arbitrary connected graph quickly converges to a fully connected graph. Name-Dropper was the inspiration for the peer discovery layer of Tangle, which adds multiple levels and random subsampling (see Section 6.1.1).

File-sharing applications like BitTorrent [6] and the original Napster [61] are peer-to-peer for the file transfer portion, but are client-server for the rendezvous portion that enables machines holding a file and machines seeking the file to find each other. In Napster, clients all connected to a central server, reported what files they had, and asked for referrals to clients holding the files they sought. BitTorrent is like a separate instance of Napster for each publisher, for the purpose of sharing only that publisher's files. Each publisher runs a server to coordinate peers which transfer the files among themselves. The publisher also runs one peer alongside the server, to make sure there is always at least one running peer that has the files. Some other file-sharing applications, like Freenet [5] and Gnutella [60], use a purely peer-to-peer architecture but punt on the rendezvous problem, using heuristics that in the worst case require a query to flood the entire peer network in order to find a file. Scalable peer-to-peer solutions to the rendezvous problem are possible, however, and several, known as distributed hash tables, were developed independently in the research community.

The rest of the related work consists of peer-to-peer solutions for the problems of multicast loss recovery and rendezvous. Therefore the remainder of this chapter is organized into Section 2.1 covering work on multicast loss recovery, and Section 2.2 covering distributed hash tables, which address the rendezvous problem. Each section presents background material about its problem domain before delving into the related work.

2.1 Multicast Loss Recovery

This section begins with a description of IP multicast and the problem of multicast loss recovery in Section 2.1.1. The main challenge of multicast loss recovery is *local* recovery. The difficulties and several avenues for overcoming them are described in Section 2.1.2.

2.1.1 IP multicast

The IP multicast service [10] allows a source to send a packet unreliably to a group of N members much more efficiently than sending N unicast packets, which would overwhelm the links and routers near the source if N is large. The source can instead send a single multicast packet, and the network makes copies en route wherever the paths to the group members diverge, so that the packet traverses every link just once. Like unicast, multicast is a *best-effort* service—sometimes packets are not delivered. Some applications, like real-time audio and video broadcasting, are delay-sensitive and loss-tolerant, and thus well-suited to IP multicast service, but other applications that could benefit from efficient multicasting require reliable delivery. Some are real-time, like distributed whiteboard [16] and multiplayer games, while others are bulk transfers, like software updates and news articles. To support these applications, we need a protocol that provides a reliable multicast transport on top of the unreliable IP multicast service, analogous to the way TCP [46] provides a reliable unicast transport on top of the unreliable IP unicast service. The first problem to solve is *loss recovery*—the detection and retransmission of lost packets. The second is *congestion avoidance* [35], but in this work we restrict our attention to loss recovery.

Before we can discuss further details we need to introduce some terminology. The Internet topology is a bipartite graph in which the nodes are *routers* and *networks*, and the edges are *interfaces*. Each interface connects a router to a network. Packets originate and terminate at *endpoints*. In reality an endpoint lies within a router (usually a degenerate router called an "end host", in which the endpoints are processes), but we will consider the endpoint to be adjacent to the router. Multicast packets are forwarded from a *source* to group *members* along a *distribution tree*. For each source/group pair, there is a tree (a subgraph of the Internet topology) in which the source is the root[1] and the members are the leaves (Figure 2.1). When a packet arrives at a node, a copy of the packet is forwarded to each child. Whenever a packet is lost, it fails to reach the members in the *loss subtree*—the subtree below the point where the loss occurred.

[1]Some proposed multicast routing protocols like PIM [11], CBT [1], and BGMP [27] can use *shared trees*, in which the source is not the root, but this chapter focuses on source-rooted trees. See Section 7.1 for more discussion of shared trees.

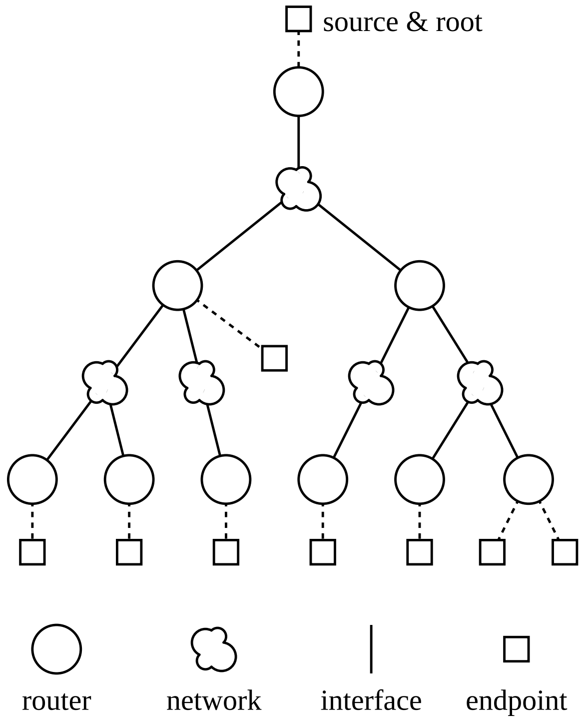

Figure 2.1: Example multicast distribution tree.

2.1.2 Local recovery

Multicast loss recovery is more challenging than unicast loss recovery because scalability and efficiency require *local recovery*—the loss recovery effort must be mostly localized around the loss subtree.

The clearest illustration of the need for local recovery is the problem of *implosion*. Reliable protocols designed for a single sender and receiver are often based on automatic repeat request (ARQ), where the receiver sends feedback to the sender, so the sender knows when it needs to send retransmissions. But if we extend ARQ to multicast, and multiple receivers send feedback to a single sender, the resulting implosion of packets can overwhelm the sender or the network links near it. If a loss recovery scheme is to scale to arbitrarily large groups, it must avoid this effect.

Implosion has been successfully addressed in a few pioneering works on reliable multicast. Scalable Reliable Multicast (SRM) [16] uses *multicast damping*. Members multicast repair requests to the entire group, but use random timers to ensure that only about one request is sent at a time. Other members waiting to send the same request see the first one and suppress their own. Any member capable of responding to a request may do so, and random timers are used again to limit the number of responses to about one.

The Tree-Based Multicast Transport Protocol (TMTP) [63] and the Reliable Multicast Transport Protocol (RMTP) [34] avoid implosion via *ACK fusion*. The members are organized into a hierarchy. ACKs are sent not to the source, but to the parent member, who merges the ACKs before sending them to its own parent.

But there is more to local recovery than just avoiding implosion. If a lost packet affecting a relatively small subtree triggers a recovery effort involving too much of the group, the protocol cannot scale to arbitrarily large groups, because the frequency of packet losses grows with the group size. SRM does not provide local recovery, because every recovery involves the entire group, although section 7.2 of [16] sketches a few potential ideas for changing that behavior.

One technique for addressing the problem of local recovery is to form additional multicast groups called *local groups* [20], each containing the members downstream of a point where losses commonly occur, so that retransmissions of packets lost at this point can be sent to the corresponding group. The difficulty with this technique is that network topology is not exposed to the source and group members—the IP service model abstracts the network as a black box.

Another technique is exemplified by TMTP and RMTP, which use their hierarchies of members not only for children to send feedback to their parents as described above, but also for parents to send retransmissions from their data caches to their children. This

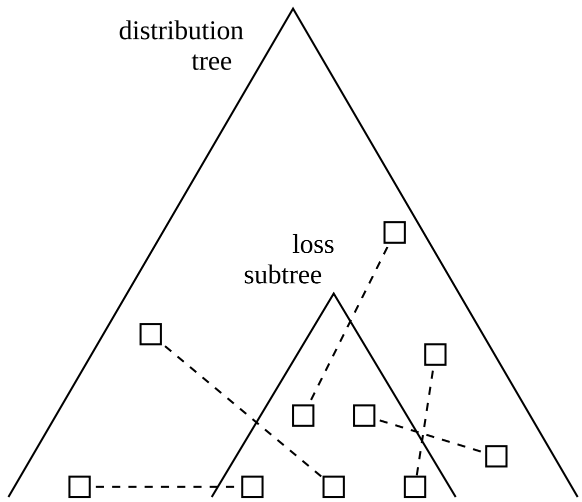

Figure 2.2: In a hierarchy constructed without knowledge of the distribution tree, identical retransmissions converge from parents outside the loss subtree onto the children inside the loss subtree.

approach is scalable because the size of the recovery effort is proportional to the number of members in the loss subtree. However, if the hierarchy of members is not well correlated with the multicast distribution tree, it will not provide the efficiency of *local* recovery. Members inside the loss subtree will typically have parents outside the loss subtree, and hence many copies of the same retransmission will converge from all directions onto the local region of the loss subtree (see Figure 2.2), causing identical packets to traverse a few links many times, which is exactly the phenomenon that multicast was designed to avoid in the first place. How can endpoints construct a hierarchy correlated with a distribution tree they cannot see (and track changes in that tree)?

Both local recovery techniques, local groups and hierarchies, are frustrated by the unavailability of topology information above the routing layer. There are at least three possible ways to get around this stumbling block.

One possibility is to expose topology information to the endpoints. The information could be obtained explicitly via diagnostic services like *mtrace*[2], as in OTERS [33] and Tracer [31], or inferred by the endpoints via observations of loss patterns [45]. Exposing topology information to endpoints has the advantage of requiring no changes to existing routers, but is somewhat counter to the Internet architecture.

A second possibility is for the network to provide transient subgroups, as in PGM [55] and BCFS [62]. Group members would inform the network of each packet loss they experience, and the network would lay down "trails of breadcrumbs" between the lossy members and the source. The network would then forward the retransmission from the source along the trails to the members that need it, removing the breadcrumbs at the same time. This approach is rather heavyweight, requiring state to be set up and torn down in routers for every packet loss.

A third possibility is for the network to provide additional forwarding services defined in terms of the multicast distribution tree. For example, RMTP proposed *subtree multicast* to send to the portion of the tree downstream from an internal router. Similarly, Lightweight Multicast Services (LMS) [42] proposed *directed multicast* to send to the portion of the tree downstream from an internal link, and also proposed *parentcast* to send to one's parent in a hierarchy constructed by the network. For our Search Party and Rumor Mill protocols we have proposed *randomcast* (Section 4.3) and *trivial randomcast* (Section 5.1) to forward a packet randomly in the tree (so that it probably goes to a nearby member). Forwarding services like these require router state changes only as often as multicast itself requires them, and they preserve the black-box abstraction of the network as viewed by endpoints.

[2] The mtrace command invokes a diagnostic function of multicast routers to discover the path from the source to the member running mtrace.

LMS is described in detail in Section 4.1 because its architecture is most similar to that of Search Party. Reliable Multicast Architecture (RMA) [32, 30] is also similar, but it proposes a single forwarding service called Addressable Internet Multicast (AIM) to provide the functions analogous to both the directed multicast and parentcast services of LMS. In both cases, their use of deterministic forwarding results in unnecessarily brittle hierarchies of members where an entire loss subtree is dependent on the correct operation of a single representative member to request the retransmission and on its single parent to send the retransmission and on the network path between the two, even though any member of the loss subtree would be capable of requesting the retransmission, and any member outside the loss subtree would be capable of sending it. One malfunctioning node can have a needlessly large negative impact on many of its peers, especially if it happens to be near the top of the hierarchy. In contrast, Search Party and Rumor Mill use randomized forwarding to achieve local recovery without hierarchies, so that they are more robust to faulty nodes and network paths.

2.2 Distributed Hash Tables

A *distributed hash table* is a data store similar to a *hash table*, but spread across many machines, which then need a protocol to coordinate their actions. There have been several such protocols developed independently, all of which derive their scalability from the *small-world phenomenon*.

2.2.1 Common characteristics

A *hash table* is a data structure that represents a set of $\langle key, value\rangle$ pairs and provides the operations put(*key,value*) to store a pair and get(*key*) to retrieve the value associated with a key. A *hash function* maps each key to a *hash value*, where the hash values are uniformly distributed over the space of all possible hash values. The hash values determine where the pairs are stored in memory, allowing the get() and put() operations to be implemented cheaply.

A *distributed hash table* is a hash table in which the pairs are not all kept on a single *node* (machine), but are spread redundantly across many peer nodes (although the usage of the term "distributed hash table" has evolved, see below). Distribution allows the table to be much larger than any single node could accommodate, allows operations to be performed at a higher aggregate rate than any single node could handle, and allows the table to remain continuously available even if every individual node is frequently unavailable.

In a distributed hash table the get() and put() operations involve multiple nodes, all of which need the hash value. Rather than recompute the hash function at every node, a distributed hash table typically sheds responsibility for computing the hash function at all, and expects a higher layer to do it. The interface becomes put(*label,value*) and get(*label*), and the table stores $\langle label, value \rangle$ pairs. The distributed "hash" table knows nothing of keys or the hash function applied to them to yield labels; it merely expects labels to be uniformly distributed. If the label space is sufficiently large then the possibility of collisions (two keys mapping to the same label) can be neglected. For example, a SHA-1 hash [58] can be considered a unique ID for all practical purposes.

The fundamental challenge of a distributed hash table is *rendezvous*. When a node performs put(*label,value*), the pair is stored at some node (or a few nodes) in the system, depending on the label. When another node later performs get(*label*), the label guides the operation back to the same place, so that the value can be retrieved. To allow the number of nodes to scale up, we cannot expect every node to know about the existence of every other node. Therefore, mapping a label to the node that stores it will involve the cooperation of many nodes, and hence the forwarding of messages along multiple hops between nodes.

The primitive operation is `send(`*message,label*`)`, which forwards a message through multiple hops to whichever node is responsible for that label. (Forwarding at the application layer typically means the same as it does at the network layer: recursively send a message to the next hop along a path to the destination. However, at the application layer there is an alternative: return the next hop's address to the original sender (via the network layer), who iteratively resends the message to the next hop. Any protocol that uses one form of forwarding can easily be adapted to use the other, and either way the central operation is choosing the next hop.)

The distributed hash table can be viewed as having two layers (although in actual protocol specifications and implementations they might be integrated): a *routing layer* that provides the `send()` operation, and a *hash table layer* that stores and retrieves values and uses the routing layer to forward `get` and `put` requests to the proper nodes. Both layers are inside the application, above the transport layer (whereas the network routing layer lies below the transport layer and is independent of the routing layer discussed here). The routing layer performs two tasks. It constructs and maintains *routing tables* (the partial information that nodes have about other nodes) as nodes join and leave the system; and it uses the routing tables to forward messages to their proper destinations.

Because the rendezvous service provided by the routing layer's `send()` operation is the fundamental contribution of distributed hash tables and is useful for more than just building hash tables, the term "distributed hash table" is now often used to refer to the routing layer alone, even though the routing layer by itself does not resemble a hash table.

2.2.2 Small-world phenomenon

The routing layer of any distributed hash table is ultimately based on some form of the *small-world phenomenon*. It has been observed that in many real-world graphs (such as the graph of people who know each other on a first-name basis) there exist short paths between arbitrary pairs of nodes (a principle popularly referred to as "six degrees of separation"). Furthermore, it is possible for these paths to be discovered using decentralized algorithms in which each node chooses the next edge using only its own knowledge of its own edges. The small-world phenomenon arises when the edges between nodes are a suitable mix of "local" and "long-range", and when there is a suitable distance metric between nodes. Kleinberg gives a good characterization of the circumstances that give rise to the small-world phenomenon [26].

2.2.3 DHT protocols

There have been a number of distributed hash table protocols developed independently, some of which are listed below along with brief descriptions of their routing layers. The most noteworthy differences between them are the different ways they exploit the small-world phenomenon to provide the `send()` operation. Each of the routing protocols works by assigning labels (or local regions of the label space) uniformly to the nodes, and forwarding a given message toward nodes whose labels are progressively "closer" to the target label according to some metric. The protocols differ in their metrics and routing table structure.

CAN

In CAN (Content-Addressable Network) [44] each label is a point in a d-dimensional toroidal space, and each node is assigned a cuboid region of the space (a subrange of each axis). Each node maintains an ongoing relationship with (and a routing table entry for) the owner of each region that borders its own region. Therefore, when a node is given a message whose target coordinates do not lie within the node's region, the node can forward the message to a peer who is closer to the target along one axis and no further from the target along all other axes. Each node maintains $O(d)$ routing entries, and the number of hops taken by a message is $O(dn^{1/d})$, where n is the number of nodes.

Chord

In Chord [56] each label is a point in a one-dimensional directed circular space of circumference M, and each node is assigned a point on the circle. The `send()` operation forwards a message to the first node after the target point, called the *successor* of the target point. A node at point x maintains an ongoing relationship with (and a routing table entry for) the successor of each of the points $x + \frac{M}{2}$, $x + \frac{M}{4}$, $x + \frac{M}{8}$, and so on, down to the successor of x itself. Given a message whose target label y does not already lie between x and its successor, the node forwards the message using the routing table entry for the furthest point that does not overshoot the target. Each node maintains $O(\log n)$ routing entries, and the number of hops taken by a message is $O(\log n)$.

Tapestry and Pastry

In Tapestry [64] and Pastry [53] each label is a sequence of symbols from some small alphabet of size b, and each node is assigned a label. The `send()` operation forwards a message to the node whose label shares the longest common prefix with the target label (breaking ties using lexicographic order). (Actually Tapestry uses suffixes while Pastry uses prefixes, but that is an arbitrary and inconsequential detail.) The routing table of a node X has as many levels as there are symbols in a label, and each level contains $b-1$ entries. At level $k \geq 0$ are $b-1$ peers whose labels agree with X's label in the first k positions and disagree in the next position, covering all $b-1$ possible symbols for the disagreeing position. Each entry is filled with the "best" peer whose label meets the constraints of that entry (for example, the peer with least network delay from X), or is empty if no peer exists whose label meets the constraints. Each node maintains ongoing relationships with the peers in its routing table. Whenever X is not the node whose label shares the longest common prefix with the target label of a given message, X has a routing entry for a peer whose label has a longer common prefix, and forwards the message there. This routing scheme is based on *PRR trees* [43]. As in Chord, each node maintains $O(\log n)$ routing entries, and the number of hops taken by a message is $O(\log n)$.

Tangle

In our distributed hash table protocol, Tangle (the subject of Chapter 6), each label is a sequence of bits, and each node is assigned a label. The `send()` operation forwards a message to the node whose label shares the longest common prefix with the target label. This is like Tapestry and Pastry with $b = 2$, except that ties are not broken (ties at the routing layer are accommodated by redundancy at the hash table layer). But unlike Tapestry and Pastry, the Tangle routing table contains at each level not the $b-1 = 1$ "best" peer for that level, but a random sample of the peers whose labels meet the constraints of that level. When a message is forwarded in the other protocols, the routing table lookup returns a single next hop, while in Tangle the routing table lookup returns a selection of next hops from which one is chosen at random, so that the same message sent several times in quick succession will take several entirely different paths, thereby preemptively routing around failures that might be present without bothering to detect any failures. While the other protocols invest resources in maintaining ongoing relationships with peers, Tangle invests those resources in continually discovering new random peers to replace the existing ones. A node does not bother trying to monitor the health of its peers because it's just going to replace them soon anyway. As in Chord, Tapestry, and Pastry, each node maintains $O(\log n)$ routing entries, and the number of hops taken by a message is $O(\log n)$.

A second-generation distributed hash table [39] slightly improved the path length to $O\left(\frac{\log n}{\log\log n}\right)$, and also used some aspects of randomized forwarding. In one version of the protocol (section 3 of [39]) each row of the routing table has a set of multiple next hops that is fixed (until nodes join or leave the system) and deterministic, rather than a random continually changing set. During forwarding the next hop is chosen randomly from among the choices, but no reason is given for the randomization, and the analysis does not depend on it. In a second version of the protocol (section 4 of [39]) a hybrid deterministic/randomized forwarding service is used in which only the deterministic hops make progress, and non-progress-making randomized hops are used only to route around detected faults. In contrast, Tangle uses progress-making randomized hops to route around undetected faults. Tangle uses a different form of hybridization at a higher layer: the original sender can send a message using pure deterministic forwarding, and send retransmissions using pure randomized forwarding (see Section 6.1.2).

Another difference between Tangle and the other protocols is the method of constructing and maintaining the routing tables. The other protocols not only use the routing tables to implement the `send()` operation, but also use the `send()` operation to construct the routing tables. A node joining the system invokes the `send()` operation using its own label as the target, and exchanges routing entries with the nodes traversed. It is easy to see that this bootstrapping technique works well when nodes join the system serially, but it is not so easy to see how well it works when all the nodes join concurrently. We never thought of that clever trick, and therefore Tangle does not use the `send()` operation for constructing and maintaining its routing tables; it uses a gossip-style protocol designed with the mass-join scenario in mind.

2.2.4 Multipath forwarding

Tangle benefits from the use of multiple redundant paths provided by randomized forwarding, but randomized forwarding is not the only mechanism that could be used to provide redundant paths.

Another approach is to build redundant paths above the routing layer, by relaying messages through a random label. For example, if a node wishes to send message M to target label t, it instead chooses a random relay label r and sends the message "send M to t" to r. This technique is used in the Distance-Halving DHT [38] for load balancing rather than fault tolerance. Even if the routing layer provides only one path for each sender/target pair, retransmissions take different paths because each uses a different intermediate relay r. The appeal of this approach is its clean separation from the routing layer, but there are two costs. First, going through a relay doubles the number of hops and the total delay. Second, if the routing layer chooses next hops based on network proximity, then even the multiple redundant paths are likely to coincide in their first few hops and last few hops. To the extent that hops are shared, the purpose of using redundant paths is defeated.

Those costs can be avoided by putting multipath support in the routing layer. The routing tables contain multiple next hops, and the routing layer provides a forwarding mode in which network proximity does not factor into the choice of next hop.

Multipath support in the routing layer does not necessarily imply randomization. A non-randomized approach has been proposed in the context of secure routing [3] in distributed hash tables, primarily Pastry. In that proposal, each node maintains a second routing table in which the "best" node for each next-hop entry is based not on network proximity but rather on the low bits of node labels. The metric is designed to avoid overlap between paths from distinct senders to the same target. Each message carries a flag indicating whether it is to be forwarded using the proximity-based routing table or the overlap-avoidance routing table. To send several copies of a message along different paths, the original sender sends each copy to a different node, and from there the copies are forwarded using the overlap-avoidance routing tables.

Tangle does not use network proximity in the construction of its routing table, but rather defers the use of proximity until forwarding time. While building the routing table it gathers several choices for each next hop, to give the application the flexibility to choose on a per-message basis whether to use network proximity (for greater efficiency) or not (for greater path diversity and hence greater fault tolerance). The choice is controlled by a hint carried in the message being forwarded. The cost of this flexibility is that the nearest next hops in the routing table will not be quite as close as they could have been if proximity had been used in the construction of the table.

Chapter 3

Methodology

Each of the three segments of this work—Search Party, Rumor Mill, and Tangle—proceeded along the following trajectory: First, we designed a peer-to-peer protocol in which peers use randomized forwarding to find each other. Then we evaluated the protocol by analyzing it, implementing it, and running network simulation experiments using the implementation. Finally, we improved the protocol based on insights gained from the initial evaluation, and reevaluated. Section 3.1 below gives several examples of this iterative process, and Section 3.2 is an overview of the simulation environments we used. Many more details regarding the analyses and experiments are provided in the forthcoming chapters.

3.1 Protocol improvements after initial evaluation

None of the three protocols originated in its final form, but rather underwent multiple revisions as early evaluations revealed problems or suggested areas for improvement. For example, in Search Party, the performance analysis was originally done to learn how much delay and communication overhead to expect from the system, but the results led directly to the unified tuning algorithm (Section 4.6.5) that peers use to dynamically adapt their request rate to achieve the delay/overhead tradeoff appropriate for their application, such as low overhead for delay-tolerant applications like news article dissemination, and low delay for delay-sensitive applications like online games. Another example is the internal response suppression mechanism (Section 4.5.5), which was not present in the first implementation, but was added after simulations revealed a race condition between retransmissions and requests for retransmissions arriving at the same endpoints within the loss subtree, causing those endpoints to send redundant retransmissions.

Rumor Mill originally sent each request in a separate message, just like Search Party, but the first round of simulations drew our attention to the resultant high request overhead, which is greater in Rumor Mill than in Search Party. We therefore incorporated the simple but effective technique of batching multiple requests into a single request packet (Section 4.5). The most straightforward tactic—batching *all* outstanding requests into a single request packet—would have caused too many retransmissions to come from the same responder, threatening to overwhelm its network link. Echoing our experience with Search Party, we analyzed the expected behavior to derive an adaptive algorithm for controlling the number of requests per message (Section 4.6).

In Tangle, the peer-discovery portion of the protocol (Section 6.1.1) chooses peers to mention in pong messages by repeatedly flipping a coin to decide how much to shrink the ID prefix, then repeatedly flipping a coin to decide how much to lengthen the ID prefix. The prefix-lengthening phase was intuitive from the start, but we did not imagine the prefix-shrinking phase until we had analyzed the behavior without it and found that the flow of new peer information was not balanced. The forwarding portion of the protocol (Section 6.1.2) defines four forwarding hints: three random distributions and one deterministic one. Originally we imagined that the application would choose one hint and stick with it. It was not until after early experiments showed the cost of a pure randomized strategy in the absence of faults that we introduced the notion of a hybrid forwarding strategy, which starts with the deterministic hint and switches to progressively more random hints for subsequent retransmissions.

3.2 Simulation environments

Search Party and Rumor Mill, being similar protocols, were evaluated using very similar simulation environments, based on an off-the-shelf network simulator, **ns** (version 2) [28], but for Tangle we developed a new environment that was only as capable as necessary, and was thereby faster and more scalable.

ns models the network routers, links, packet queues, packet header fields, transport layer protocols, and so on, in great detail, so that the simulated network accurately mirrors the dynamics of a real network. We chose **ns** partly because of its track record of usefulness in fruitful research projects [15, 36], and also because of the availability of an active **ns** user community at Berkeley to help iron out kinks. The simulator is object-oriented, and objects have both a C++ interface and an OTcl interface (Object Tcl [59] is an object-oriented extension of Tcl [41], a scripting language), so that pieces of the system can be extended in either C++ (for performance) or in Tcl (for rapid prototyping) or in a combination of both. We implemented the Search Party and Rumor Mill endpoints and forwarding services in Tcl, inheriting the C++ implementations of the network nodes, links, background traffic generators, and event scheduler. **ns** was designed for accuracy but unfortunately not for scalability (at least, not back when these simulations were first run), and attempts to increase from 64 to 128 nodes were unsuccessful. Since the simulated networks were already unrealistically small, and since Search Party was provably insensitive to topology in some key respects (Section 4.7), we did not attempt to use realistic topologies, but instead used two synthetic topologies (Figure 4.8): a balanced binary tree, because it had been used in the evaluation of a related protocol (LMS) with which we wanted to compare results, and a square topology, to expose sensitivities to degree and diameter (which are greater and grow faster in the square topology than in the balanced topology).

Though the multicast loss recovery simulations were small, they yielded the results we needed. At this point, we could have reimplemented those protocols in C and sought a more scalable simulator/emulator/testbed to conduct more realistic experiments, but we decided that a better use of time would be to expand the scope of the research to another problem domain. Scalable peer-to-peer rendezvous was the missing piece in the rising tide of peer-to-peer applications, so we developed Tangle to fill that gap.

Tangle was less amenable to analysis than Search Party and Rumor Mill, so we compensated with more realistic and larger scale simulation experiments. Because Tangle's forwarding is done at the application layer, any node can use any other node as a next hop—the "topology" is simply a fully connected graph, and the only thing to simulate is the end-to-end network delay. By writing the delay component ourselves (Section 6.2.2), we were freed from the scaling limits of **ns**, and had a welcome excuse to write a real protocol implementation in C rather than Tcl (Section 6.2.1).

Chapter 4

Search Party

Sending the same data to a group of N receivers using the Internet Protocol (IP) unicast (point-to-point) forwarding service is expensive, requiring $O(N)$ bandwidth at the source. IP multicast is a forwarding service introduced to reduce the bandwidth requirement to $O(1)$ at the source and throughout a distribution tree constructed by the network. Multicast, like unicast, is a *best-effort* service—the packets are sent unreliably. Some applications, like distributed whiteboard and news articles, require detection and retransmission of lost packets. For scalability and efficient use of the network, *local recovery* is needed to repair packet losses affecting small regions of the distribution tree without involving many nodes far from that region. Peer-to-peer solutions have been proposed that achieve local recovery by constructing a hierarchy among the group members, but this makes the many peers near the bottom of the hierarchy very sensitive to faults affecting peers near the top of the hierarchy. To address this issue we propose Search Party [9], which takes the local recovery structure of a hierarchy-based local recovery solution, LMS [42], and substitutes randomized forwarding in place of the hierarchy, eliminating the strong dependencies between the peers and improving the overall fault tolerance of the system.

This chapter is organized as follows. Section 4.1 describes LMS and identifies the fault tolerance issues that Search Party improves upon. In Section 4.2 we give an overview of our Search Party scheme, which is built on top of our proposed new forwarding service, randomcast (described in Section 4.3), and another forwarding service proposed by others, subcast (described in Section 4.4). Section 4.5 gives the details of the protocol. Analytic results in Section 4.6 show greatly reduced susceptibility to faults. Proofs and derivations of several formulas appearing in Sections 4.3, 4.5, and 4.6 are factored out into Section 4.7, with forward references from the earlier sections, to avoid interrupting the flow. Simulation results in Section 4.8 corroborate analytic approximations showing a modest cost in overhead or retransmission delay or a tunable combination of the two. The chapter concludes with a summary in Section 4.9.

4.1 LMS

This section describes Lightweight Multicast Services (LMS) [42], a piece of related work whose details are needed as a point of departure for the presentation of Search Party. For more general background and related work regarding multicast loss recovery, see Section 2.1.

LMS provides local recovery using an implicit hierarchy based on the multicast tree itself, freeing the group members from having to construct and maintain the hierarchy. To make this possible LMS extends the routers with a new but simple forwarding service that was unnamed, but which we call *parentcast* because it forwards a packet to the sender's parent in the implicit hierarchy. Each router in a multicast tree selects one of its child links to be its *replier link*. (LMS is described in terms of point-to-point links rather than networks and interfaces.) A packet received from a non-replier link is forwarded to the replier link, while a packet received from the replier link is forwarded to the parent. When a member

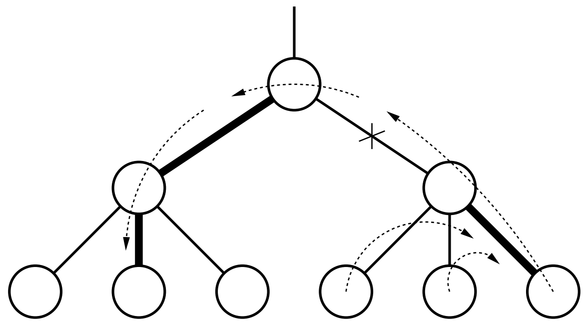

Figure 4.1: The paths taken by three request packets after a packet loss in LMS. The replier links are indicated by heavy lines.

detects a lost packet, it uses this forwarding service to send a request for a retransmission. The remarkable consequence of the forwarding rule is that when every member in a loss subtree sends a request for the missing packet via parentcast, exactly one request will escape the subtree, even though no one (not even the routers) knows where the loss occurred or how many requests were sent. The other requests go to members inside the loss subtree, who are unable to respond (Figure 4.1).

The router that bounces an upward-moving request back downward is called the *turning point* (notice that a request packet can reverse direction at most once). The turning point router inserts into the request its own address and an identifier for the link on which the request arrived. This allows the recipient of the escaping request to send a response packet via *directed multicast*, which is very similar to RMTP's subtree multicast, except that it uses the link ID to further restrict the audience. Whereas the subtree multicast service forwards a packet to a subtree rooted at a router, directed multicast service forwards a packet to a subtree rooted at a link.

Loss recovery is typically local, but not always. When the loss occurs on a non-replier link the locality is perfect—the response goes only to the loss subtree. When the loss occurs on a replier link, the turning point is higher than necessary and the response goes to a larger subtree. The locality of retransmissions therefore depends heavily on where losses occur and on which links are chosen to be replier links.

LMS also depends heavily on the correct functioning of members near the top of the hierarchy. As we saw in Section 2.1.2, hierarchy-based loss recovery is localized only if the hierarchy is aligned with the distribution tree. LMS achieves perfect alignment effortlessly by letting the network select replier links, but this alignment also brings a risk, because every subtree has just one member who sends requests on its behalf and one member who responds to them. Responsibility is concentrated—a single member bears the entire burden of sending retransmissions for any given lossy link, and one malfunctioning member can adversely affect a large number of other members below it in the hierarchy.

4.2 Overview

The rest of this chapter presents our solution for more fault-tolerant local recovery in reliable multicast, addressing concerns raised at the end of the previous section regarding LMS. The key insight is that fault tolerance can be improved through randomization. If requests are directed to other members chosen randomly rather than deterministically, responsibility is diffused, and it becomes impossible for one malfunctioning member to have a significant impact on a large number of other members. Whereas LMS uses a simple deterministic forwarding service, we propose a simple randomized forwarding service called *randomcast*, and build upon it a scalable, efficient, and robust multicast loss recovery protocol called *Search Party*.

Like LMS, Search Party introduces new network functionality that respects clean layering principles—the routers do not see any transport-layer information like sequence numbers, nor do the endpoints know anything about topology. As in LMS, requests for retransmissions are sent up the tree toward the root and bounced back down with inserted information about the turning point, and responses are sent to the subtree below the interface through which the request arrived at the turning point node. Unlike LMS, Search Party requests are forwarded randomly via randomcast. Randomizing the routing of requests trades off performance (a constant factor) for improved fault tolerance. Because responsibility is diffused evenly throughout the group, a malfunctioning member has a limited impact on its near neighbors, and a vanishing effect on the entire group. Also, the burden of sending retransmissions for a particular lossy link is shared among many members, rather than a single replier or designated receiver. The performance penalty takes the form of increased retransmission delay and/or increased recovery overhead, and applications with varying delay sensitivities can fine-tune the tradeoff between the two.

When a member detects a loss, it continually sends requests until a retransmission arrives. In effect, each member conducts a random search for the missing data. Because no one knows where the loss occurred, no one knows how broad a search ought to be conducted; however, because all and only those members affected by the loss participate in the search, they automatically form a *search party* of just the right size and scope.

4.3 Randomcast Forwarding

We propose a new service called *randomcast* that forwards packets randomly, without backtracking, along the nodes and edges of a multicast distribution tree. The mechanics are given in Section 4.3.1, but an interesting policy question is what probability distribution to use to select the outgoing interface at each hop, because this choice is critical to the behavior of systems using the randomcast service. We evaluated two distributions: uniform, and weighted by subtree population. Section 4.3.2 discusses the tradeoffs for choosing between the parent and the children, and concludes that the weighted distribution is preferable because it better diffuses responsibility for soliciting retransmissions. Section 4.3.3 discusses the tradeoffs for choosing among the children and concludes that the uniform distribution is preferable because it more evenly distributes load on the network.

4.3.1 Mechanics

A regular multicast packet contains a multicast group address and a source address, which routers use to infer which distribution tree to use, but the mapping from source-group pairs to trees is private to the routers (it may be a source-rooted tree or a shared tree). A randomcast packet includes two source addresses—the address of the sender, and the source address that would have appeared in the multicast packet that the sender is trying to recover. The routers need the second source address to determine which tree to use. When a randomcast packet arrives at a node, it may be forwarded to any neighbor in the tree except the one the packet came from. (This is not a problem if the node is a router, but is slightly tricky if the node is a network, because networks are generally not intelligent. The best solution is to have routers adjacent to the network perform the forwarding decisions on the network's behalf. Alternatively, networks can be ignored, and routers can be considered neighbors of each other.) Whenever a node acts as a turning point (receives a randomcast packet from a child and forwards it to another child), it inserts into the packet information sufficient to address the subtree below the arrival interface (see Section 4.4).

4.3.2 Parent or child?

When a randomcast packet arrives at a node X from a child C, the node first decides whether to forward the packet to the parent or to another child. If the distribution is uniform, the probability that the parent is selected is $1/c$, where c is the number of children of X. (The probability is not $1/(c+1)$ because the node is forbidden from returning the packet to the child it came from.) If the distribution is weighted by subtree population,

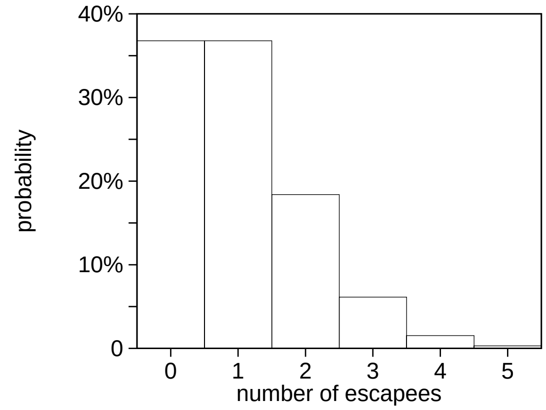

Figure 4.2: Probability that k randomcast packets escape a large subtree in which each leaf sends one packet (Poisson distribution with mean 1).

the probability that the parent is selected is $L(C)/L(X)$, where $L()$ denotes the number of leaves descended from a node (1 if the node is itself a leaf). An essential feature of both distributions is that the sum of the probabilities over all the children is 1.

Some multicast routing protocols already require routers to be aware of their children in the tree, so each router could occasionally communicate with its children and obtain a count of the number of leaves below each one, computing the sum for its own use and its parent's. The EXPRESS multicast service proposal [21] suggests that routers keep track of subtree populations. The uniform distribution does not require knowledge of subtree populations, but the weighted distribution has two attractive properties.

First, the probability that a randomcast packet originating at any leaf below node X reaches X's parent is $1/L(X)$, regardless of the topology (see Section 4.7.1 for derivations of the formulas in this paragraph). This diffuses responsibility for sending requests evenly among the members, and also eases analysis: If m randomcast packets are sent from each of the L leaves in a subtree, the number of packets that escape the subtree has a binomial distribution with parameters $n = mL$ and $p = 1/L$, for which the expected value is $np = m$. It is well-known that as n increases while np remains constant, the binomial

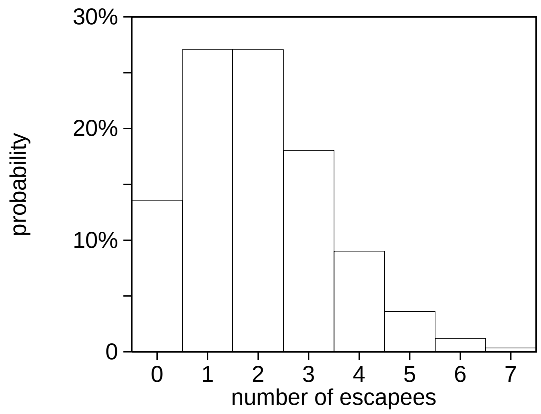

Figure 4.3: Probability that k randomcast packets escape a large subtree in which each leaf sends two packets (Poisson distribution with mean 2).

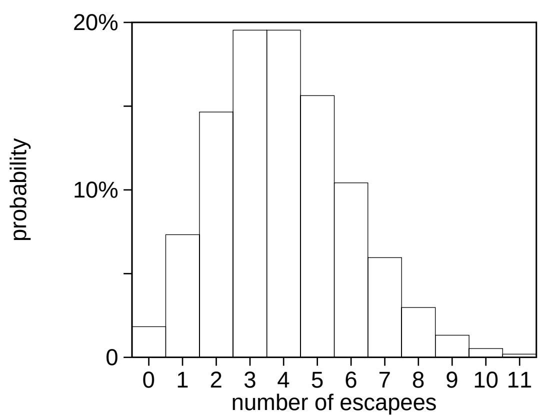

Figure 4.4: Probability that k randomcast packets escape a large subtree in which each leaf sends four packets (Poisson distribution with mean 4).

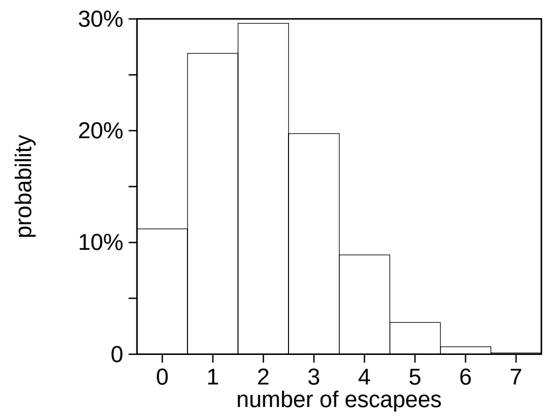

Figure 4.5: Probability that k randomcast packets escape a subtree of six leaves in which each leaf sends two packets (binomial distribution with $n = 12$ and $p = 1/6$). Compare this to Figure 4.3.

distribution asymptotically approaches the Poisson distribution with $\lambda = np$. Therefore, as L increases, the probability that k packets escape approaches $\frac{m^k}{k!e^m}$ (which is independent of L). The Poisson distribution is fairly narrow, as shown in Figures 4.2 through 4.4, meaning that the number of escapees is fairly predictable. The binomial distribution is slightly narrower for smaller subtrees, but even with only six leaves it has about the same shape, as shown in Figure 4.5.

The second attractive property of the weighted distribution is locality. Consider the scenario of Figure 4.6. With a uniform distribution, there is a 50% chance that a request from the smaller subtree will be forwarded to X's parent, causing the response to be delivered to the members in both subtrees, of which 99% are not interested. With a weighted distribution, there would be only a 1% chance of this undesirable occurrence. If the loss occurs just above the larger subtree, the weighted distribution will almost always forward the request to X's parent, but the response will still be of interest to 99% of the recipients.

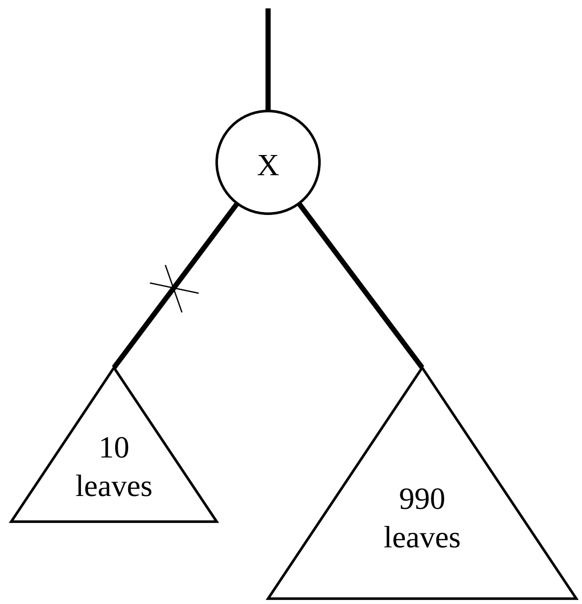

Figure 4.6: A packet loss in an imbalanced tree.

Thus, for the choice between parent and children, Search Party uses the weighted distribution.

4.3.3 Which child?

When a packet is to be forwarded by a node X to one of its children, either because it arrived from the parent or because the node has already decided against forwarding to the parent, there is again a choice between uniform and weighted distributions. With a uniform distribution, the probability of selecting child C is $1/c$ or $1/(c-1)$ depending on whether the packet arrived from a child (and is therefore forbidden from being forwarded to that child). With a weighted distribution, the probability is $L(C)/L(X)$ if the packet arrived from the parent. If the packet arrived from child A, the probability that it gets forwarded to child C is $L(C)/[L(X)-L(A)]$. It follows that before the node has decided whether to forward to the parent, if it is using a weighted distribution for both choices, the probability that it forwards to C is $L(C)/L(X)$.

A node using the weighted distribution for selecting children will tend to distribute randomcast packets evenly among its descendent leaves, but packets from many incoming edges can get concentrated onto one outgoing edge. If that happens several times in a succession of nodes, the downward traffic on a single edge could grow exponentially. With the uniform distribution, a node distributes packets evenly among its downward edges, avoiding that hazard. Because trees with more leaves tend to be deeper, the weighted distribution will tend to route packets to farther away leaves, whereas the uniform distribution will tend to choose shorter paths, leading to smaller round-trip times. For these reasons, Search Party uses the uniform distribution for selecting children.

4.4 Subcast Forwarding

Subtree multicast, proposed for use with RMTP, and directed multicast, used by LMS, are similar services. We use the generic term *subcast* to refer to any forwarding service that allows packets to be sent to a subtree. Recall from Section 2.1.1 that the nodes of the Internet topology alternate between routers and networks, so that either may be the root of a subtree. Ideally, subcast would support both router-rooted and network-rooted subtrees; if not, a loss just above a non-addressable node X will have to be treated as c losses, one above each of X's child nodes.

The root node of the destination subtree is called the *exploder*. There are at least two ways the sender of a subcast packet could specify the desired exploder: by address, or by distance. If addresses are used, a member can subcast a packet to a subtree that it does not belong to, as required by LMS and Search Party. If distances are used, the exploder must be on the path from the member to the root, so a member can subcast only to subtrees that it belongs to. In this case, there is a simple workaround: a response can be sent indirectly via unicast to the requestor, who then relays the response via subcast to the desired subtree. Therefore, Search Party does not constrain the design of the subcast service.

4.5 Loss Recovery Protocol

Search Party is a loss recovery protocol built on top of the randomcast and subcast forwarding services, as outlined in Section 4.2. This section fills in the details of the protocol. Section 4.5.1 explains the basic operation, like how members detect losses, and on what schedule they resend repair requests. The choice of rate at which to send repair requests will depend on response time estimates (see Section 4.6.5), so in Sections 4.5.2 and 4.5.3 we specify the method by which members estimate response times. Another use for the response time estimates is to jump-start the recovery process to decrease retransmission delays, as described in Section 4.5.4. Section 4.5.5 identifies a race condition that exists in both LMS and Search Party that can cause redundant retransmissions, and tells how we avoid this pitfall. Finally, Section 4.5.6 discusses available options for members to decide when it is safe to delete old data.

4.5.1 Basic operation

Each group member is responsible for detecting missing packets and for obtaining retransmissions by sending requests until a response arrives. A member is able to detect losses because the source multicasts *updates* that contain sufficient state information, like the greatest sequence number sent so far. The updates can be piggybacked with data except during lulls in the data stream.

Because the Internet can reorder packets, each member must estimate the delay variance, which it can do if the source puts a timestamp in every update[1]. Each member chooses as a *loss detection threshold* a multiple of the estimated standard deviation of the delay. After observing a gap in the data, the member waits for the threshold to elapse before concluding that a packet has been lost. The choice of multiplier is a trade-off between delay and probability of false detection. If the network layer were to provide almost-always-in-order delivery, a threshold of zero would be sensible.

After a member detects a loss, it acts as a *requestor*, emitting randomcast *request packets* as a Poisson process with mean rate[2] μ until it gets a corresponding *response packet*. Like the loss detection threshold, μ is a trade-off between overhead and delay. If requests are sent more often, the expected time to escape the loss subtree is lower, but more requests are likely to escape unnecessarily and generate duplicate responses. Each member may choose its own value of μ. During a recovery, the effective μ value is the average over all members in the loss subtree.

Any member who receives a request acts as a *responder*. If the responder has the requested data, it subcasts a response packet to the subtree indicated in the request, otherwise it does nothing. When the source receives a request, it behaves the same way a member would, except that it uses multicast rather than subcast (the request has no turning point).

[1] We need not assume that the source clock and the member clocks all run at the same rate, because each member can estimate its drift relative to the source as in NTP [37], and every time value appearing in a packet (except the timestamps in the updates) can be converted to source clock ticks by the sender and then to local clock ticks by the receiver.

[2] Packet emission rates are measured in inverse time units. For example, 1/s means one packet emission per second.

4.5.2 Response time estimation

If a request generates a response, the amount of time between the emission of the request and the reception of the response by the requestor is called the *response time* (r). Each member must estimate the expected response time (both the mean and variance) for recoveries involving that member, for reasons to be given in Section 4.5.4 and Section 4.6.5. The best-guess estimate is called $\hat{r}$, and the estimate of a rarely-exceeded upper bound is called r_{hi}. Suppose a requestor puts a timestamp in an outgoing request, and the timestamp is copied into the response by the responder. This allows the requestor, upon receiving the response, to calculate the response time[3]. It does not, however, allow the other recipients of the response to calculate the response time. We could have the requestor announce the response time via a subcast packet, but we can avoid that extra traffic by altering the representation of the timestamp.

Let every update contain a unique ID, and every request contain a *time reference*, which is a pair of numbers: the ID of the most recently received update, and an offset, which is the time difference between the reception of that update and the sending of the request. Every response contains a time reference copied from the request. The requestor can calculate the response time r from the time t_{resp} that the response is received, the time t_{ref} that the referenced update was received (if it was received), and the offset:

$$r = t_{\mathrm{resp}} - (t_{\mathrm{ref}} + \mathit{offset})$$

Furthermore, every recipient of the response can perform the same calculation and get approximately the same result, because the difference between the times two members receive a packet is usually about the same for any multicast or subcast packet, so the variations in t_{resp} and t_{ref} will cancel out.

[3]It is not sufficient for the requestor to remember when the request was sent, rather than insert a timestamp, because it sends multiple requests that must be disambiguated [25].

4.5.3 Null requests

If loss rates are low, there may be inadequate losses to generate accurate response time estimates. To allow adaptation to continue in the face of low loss rates, members may send null requests as a Poisson process with mean rate ρ. A null request always generates a null response. If every member chooses the same ρ, every member can expect to receive null responses at a rate between ρ and $\rho \cdot (1 + \ln N)$, where N is the number of members (derived in Section 4.7.2). The responses will have turning points distributed evenly over the balanced heights of the tree (proved in Section 4.7.2), so there will be samples representing the full range of possible response times. The *balanced height* of a node X is $\ln L(X)$; for any balanced tree, the balanced height is proportional to the actual height. Like μ values, ρ values effectively get averaged: a member's choice of ρ affects itself most, and other members decreasingly with distance.

4.5.4 Request burst

One use of the response time estimates is to refine the request behavior. According to the earlier description, a requestor sends requests as a Poisson process until a response arrives. Therefore, requests scheduled to be sent after the response arrives will not be sent, but requests scheduled to be sent before the response arrives are destined to be sent, so they might as well have been sent as soon as the loss was detected, to reduce the recovery time. Of course, the requestor does not know ahead of time when the response will arrive, but it can take a reasonably conservative guess that it will be at least $\hat{r}$ after the loss is detected. The requestor expects to send $\hat{r}\mu$ requests during the first $\hat{r}$, so it should send them in a burst when the loss is detected, then wait $\hat{r}$ before beginning the Poisson process. Note that $\hat{r}\mu$ is generally not an integer; the requestor issues $\mathrm{trunc}(\hat{r}\mu)$ requests, plus one more with probability $\mathrm{frac}(\hat{r}\mu)$. The request burst can decrease retransmission delay and duplicate responses if the loss detections happen at about the same time, and the actual response times are all about the same. If this is not the case, the burst will not significantly alter the aggregate behavior of the members, compared to the original Poisson-only strategy.

4.5.5 Internal response suppression

There is a problem with duplicate responses coming from *inside* the loss subtree. Before any response arrives, requests are circulating inside the loss subtree, arriving at members who, being inside the loss subtree and lacking the data, ignore the requests. But when the first response arrives, there are requests are already in flight, about to arrive at members who just now received the same retransmission that has already satisfied the requestors. We do not wish for these requests to generate responses.

To prevent these internal duplicate responses, each member remembers the last time it received a retransmission of the data, and ignores requests that predate the retransmission. If a responder has the requested data, and has seen a retransmission of the requested data, it maps the request's time reference to a moment $t_{\mathrm{req}} = t_{\mathrm{ref}} + \mathit{offset}$, and compares t_{req} to the arrival time t_{rtx} of the most recent retransmission of the requested data[4]. If it were the requestor performing the comparison, then obviously the request has already been satisfied iff $t_{\mathrm{req}} \leq t_{\mathrm{rtx}}$. It is actually the responder who performs the comparison, but because the difference between the times two members receive a packet is usually about the same for any multicast or subcast packet, the variations in t_{rtx} and t_{ref} will cancel out. To allow for some noise, the responder compares $t_{\mathrm{req}} - t_{\mathrm{rtx}}$ against a positive threshold rather than zero. The threshold is a multiple of the combination of the requestor's and responder's estimated standard deviations of data packet delay, so the requestor must include its estimate in the request packet. Because variances tend to be additive, standard

[4]If the responder is unable to calculate t_{req} because it has not received the referenced update, it must use a default guess. For example, it could use *now* if low delay is valued more than low overhead, or $\mathit{now} - r_{\mathrm{hi}}$ if low overhead is valued more, or perhaps $\mathit{now} - \hat{r}/2$.

deviations are combined as follows:

$$\sigma = \sqrt{\sigma_1^2 + \sigma_2^2}$$

4.5.6 Message stability

When may the source and members delete old data and associated state information? If the source wishes to ensure that all members receive the data, there must be a group membership protocol and a scheme for merging ACKs, problems not addressed by Search Party. If it is left to each member to ensure that it has received all the data, a simple course-grain timeout may suffice. The source discards old data when the deletion timeout expires, and includes information in the updates so that members can discard their old data also. If a member cannot recover some lost data before it is discarded, it might mean that the information is too old to be useful anyway, or that something is so seriously broken that an entirely different avenue for obtaining the information should be tried instead, like fetching from an archive.

4.6 Performance Analysis

An application using Search Party must choose the rate at which it sends requests. The choice can be a function of the response time estimates, but what function? Before we can answer that question, we need to understand the effects that these variables (μ, $\hat{r}$, r_{hi}) have on the performance of the system after a loss. Sections 4.6.1 through 4.6.4 give analytic results for four effects: response duplication, request traffic, response locality, and retransmission delay. Section 4.6.5 then shows how the equations can be used to tune μ.

To make the analysis tractable, we make the following simplifying assumptions:

- Every member in the loss subtree detects the loss at the same time, called time 0.

- Every member in the loss subtree has the same value for $\hat{r}$.
- The response time r is the same for every *escapee* (request that escapes) from the loss subtree.
- A response arrives simultaneously at all members.
- Requests and responses are not lost.

These assumptions are approximations, so the results they imply are approximations. An analysis of the sensitivity to these approximations is left as future work; we instead use simulations to test the results (see Section 4.8).

4.6.1 Response duplication

The results in this section are derived in Section 4.7.3.

After a loss is detected, members in the loss subtree send requests until the first response arrives. The members then stop sending requests, but some of the requests already in flight can escape the loss subtree and generate duplicate responses. Let μ denote the average request rate among the members in the loss subtree. If $r \leq \hat{r}$, the expected number of escapees (and hence, responses) is

$$x(r) = \hat{r}\mu + (1 + r\mu)e^{-\hat{r}\mu}$$

If $r \geq \hat{r}$, the expected number of escapees is

$$x(r) = r\mu + (1 + \hat{r}\mu)e^{-\hat{r}\mu}$$

Our best guess is $r = \hat{r}$, at which point the two expressions are equal, yielding

$$x = \hat{r}\mu + (1 + \hat{r}\mu)e^{-\hat{r}\mu}$$

A pessimistic prediction is $r = r_{\text{hi}} > \hat{r}$, yielding

$$x = r_{\text{hi}}\mu + (1 + \hat{r}\mu)e^{-\hat{r}\mu}$$

If the network is not working at time 0 (perhaps because the loss was caused by a persistent malfunction), all of the immediate requests will be lost, rendering the request burst ineffective. The expected number of escapees is then

$$x(r) = 1 + r\mu$$

which is worse (greater) if $r \geq \hat{r}$, though not necessarily if $r < \hat{r}$. Therefore, an extra-pessimistic prediction is

$$x = 1 + r_{\text{hi}}\mu$$

This prediction is an especially safe upper bound on x, because it does not rely on the network working at time 0, nor on the accuracy of $\hat{r}$, and assumes that the response time of every escapee is r_{hi}, though in actuality almost all will be less than r_{hi}.

4.6.2 Request traffic

During a recovery, members inside the loss subtree receive requests that they ignore. Let L be the number of leaves in the loss subtree. Since each request has a $1/L$ chance of escaping, and the expected number of escapees is x, the expected number of requests generated must be xL. The $x \cdot (L-1)$ requests that do not escape are redistributed among the L leaves. Therefore, on average, a member in the loss subtree expects to receive about the same number (x) of requests as responses.

During the recovery each edge in the loss subtree carries upward request traffic with mean rate equal to the average μ value among the members below itself (proved in Section 4.7.3). Above the loss point, the request traffic dissipates—this is the sense in which the search for the missing data automatically has the proper scope, even though no one knows where the loss occurred. If each member uses the same μ value, the downward request traffic on each edge in the loss subtree must be less than 2μ (proved in Section 4.7.3). Although the μ values are not necessarily the same, this result shows that the randomcast routing itself does not concentrate request traffic. As a corollary, the null request traffic on every edge is ρ upward and less than 2ρ downward.

4.6.3 Response locality

[42] defines *exposure* as the ratio of the number of members who receive a response over the number of members in the loss subtree. Exposure is thus a measure of non-locality, and perfect locality corresponds to an exposure of 1. For Search Party, the expected exposure of a loss depends on the topology, but cannot exceed $1 + \ln \frac{N}{L}$ (derived in Section 4.7.3).

For comparison with LMS, consider a balanced tree with losses equally likely on all edges. In this scenario, LMS gives upward-moving requests a $1/c$ chance of continuing upward. Search Party also gives a $1/c$ chance (regardless of how losses are distributed), so the expected exposure is the same $(1 + \frac{c-1}{c} \log_c \frac{N}{L})$.

Response duplication (see above) and non-locality are orthogonal phenomena that each cause more response packets to be delivered than necessary. The total expansion factor is the product of the individual factors.

4.6.4 Retransmission delay

The probability that no retransmission has arrived before time t is derived in Section 4.7.3 to be

$$p(t) = \begin{cases} 1 & \text{for } t < r \\ e^{-\hat{r}\mu} & \text{for } r \leq t \leq r + \hat{r} \\ e^{(r-t)\mu} & \text{for } t \geq r + \hat{r} \end{cases}$$

The middle section of the distribution relies on the effectiveness of the request burst, which is questionable. Therefore, when choosing a μ value, applications should consider the probability distribution only for $t \geq r_{\mathrm{hi}} + \hat{r}$. The best-guess prediction is thus

$$p(t) = e^{(\hat{r}-t)\mu}$$

and the pessimistic prediction is

$$p(t) = e^{(r_{\mathrm{hi}}-t)\mu}$$

The average retransmission delay (expected value of t when the retransmission arrives) is derived in Section 4.7.3 to be

$$y(r) = r + (\hat{r} + \frac{1}{\mu})e^{-\hat{r}\mu}$$

which is $\frac{x}{\mu}$ if $r \geq \hat{r}$, which works for our best-guess ($r = \hat{r}$) and pessimistic ($r = r_{\mathrm{hi}}$) predictions. Of course, the delay cannot be predicted at all if the network is not working at time 0.

Notice that the delay is independent of N. To consider the asymptotic behavior as r increases, we can ignore the distinction between r, $\hat{r}$, and r_{hi}. Since x is a function of $r\mu$, holding x constant makes $r\mu$ constant, so that the expected retransmission delay is $O(r)$, whereas it was simply r in LMS.

4.6.5 Examples of tuning

To see how application programs can tune μ to suit their own needs, we consider three hypothetical applications.

News articles Suppose this application can tolerate long delays, but wants a probability of less than one in a billion that a retransmission will take longer than ten minutes (perhaps because the source will discard the data by then). Round-trip times of more than ten seconds are unheard-of in the Internet, so for simplicity the application could let $r_{\mathrm{hi}} = 10\,\mathrm{s}$ and not bother to estimate it. The request burst (sending $\hat{r}\mu$ requests immediately and then waiting $\hat{r}$) has a negligible benefit when μ is small, as it will turn out to be. Therefore, the application could forgo the burst by letting $\hat{r} = 0$ and not bothering to estimate it. The pessimistic delay requirement is

$$\begin{aligned} e^{(r_{\mathrm{hi}} - 600\,\mathrm{s})\mu} &\leq 10^{-9} \\ (10\,\mathrm{s} - 600\,\mathrm{s})\mu &\leq \ln 10^{-9} \\ \mu &\geq \tfrac{1}{28.5\,\mathrm{s}} \end{aligned}$$

The expected number of responses per loss is, extra-pessimistically, $x = 1 + r_{\mathrm{hi}}\mu = 1.35$, but if r has a more realistic value, say $1\,\mathrm{s}$, then $x = 1.04$. The expected retransmission delay is, pessimistically, $y = r_{\mathrm{hi}} + \frac{1}{\mu} = 38.5\,\mathrm{s}$.

Distributed whiteboard Suppose this application wants retransmissions to take less than $t_{\max} = 2\,\mathrm{s}$ from the time the loss is detected, with a failure probability of $p = 10\%$. The pessimistic delay requirement is

$$\begin{aligned} e^{(r_{\mathrm{hi}} - t_{\max})\mu} &\leq p \\ \mu &\geq \frac{\ln p}{r_{\mathrm{hi}} - t_{\max}} \end{aligned}$$

The application must adjust μ in response to changes in r_{hi}. Suppose the estimates $\hat{r}$ and r_{hi} are $0.2\,\mathrm{s}$ and $0.8\,\mathrm{s}$ respectively. Then $\mu = 1.92/\mathrm{s} = \frac{1}{521\,\mathrm{ms}}$. The expected number of responses per loss is $x = 1 + r_{\mathrm{hi}}\mu = 2.54$ (extra-pessimistic) or $x = \hat{r}\mu + (1 + \hat{r}\mu)e^{-\hat{r}\mu} = 1.33$ (best-guess). The expected retransmission delay is $y(r) = r + (\hat{r} + \frac{1}{\mu})e^{-\hat{r}\mu} = r + 0.49\,\mathrm{s} = 1.29\,\mathrm{s}$ (pessimistic) or $0.69\,\mathrm{s}$ (best-guess). If network performance were to improve, causing r_{hi} to decrease, this application would respond by decreasing μ to decrease the overhead while maintaining the delay constraint.

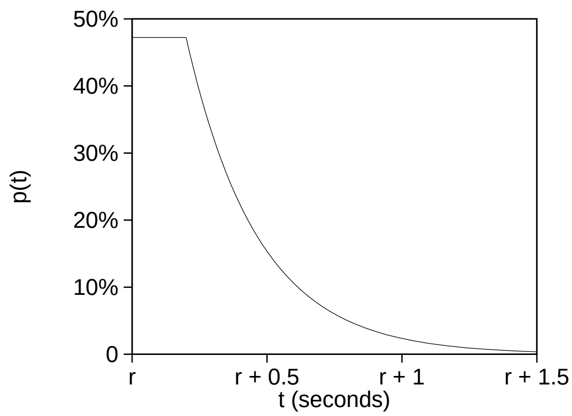

Figure 4.7: The probability $p(t)$ that no retransmission has yet arrived at time t for the multiplayer game application. Notice that the left edge of the graph is time r, not time 0.

Multiplayer game This application wants to spend at most $x_{\text{max}} = 4$ responses per loss, on average, to get retransmissions as quickly as possible. The extra-pessimistic response requirement is

$$\begin{aligned} x = 1 + r_{\text{hi}}\mu &\leq x_{\text{max}} \\ \mu &\leq \frac{x_{\text{max}} - 1}{r_{\text{hi}}} \end{aligned}$$

As with the whiteboard application, the game application must adjust μ in response to changes in r_{hi}. If we again suppose that $\hat{r}$ and r_{hi} are 0.2 s and 0.8 s respectively, then $\mu = 3.75/\text{s} = \frac{1}{267\,\text{ms}}$. The expected number of responses is 4.00 (extra-pessimistic) or 1.58 (best-guess). The expected retransmission delay is $r + 0.22\,\text{s} = 1.02\,\text{s}$ (pessimistic) or 0.42 s (best-guess). The probability that the response has not arrived before time t is shown in Figure 4.7 in terms of r. If network performance were to improve, causing r_{hi} to decrease, this application would respond by increasing μ to decrease the delay while maintaining the overhead constraint.

Unified tuning algorithm The methods used by the whiteboard and game applications for calculating μ can be combined into a single algorithm. Let the application choose a target retransmission delay $t_{\max}$, an acceptable failure probability $p > 0$, and a maximum average number of responses per loss, $x_{\max} > 1$. The response constraint overrides the delay constraint. The protocol library can then calculate

$$\mu = \begin{cases} \frac{x_{\max}-1}{r_{\text{hi}}} & \text{if } r_{\text{hi}} \geq t_{\max} \\ \min\left(\frac{\ln p}{r_{\text{hi}}-t_{\max}}, \frac{x_{\max}-1}{r_{\text{hi}}}\right) & \text{otherwise} \end{cases}$$

If the application is required to choose p and $x_{\max}$ such that

$$p \leq e^{1-x_{\max}}$$

then the library will always be obeying the admonition to use the $p(t)$ distribution only for $t \geq r_{\text{hi}} + \hat{r}$, because the response constraint will override the delay constraint whenever $t_{\max} < 2r_{\text{hi}}$ (proved in Section 4.7.3).

4.7 Proofs and Derivations

This section uses the notation introduced in Sections 4.3 through 4.6, augmented by the use of L_i as an abbreviation for $L(X_i)$.

4.7.1 Randomcast forwarding statistics

Suppose there is one member M at node X_0. The probability that a randomcast packet sent by M successfully travels upward through nodes $X_1, X_2, X_3, \ldots$ to emerge upward from node X_i, is

$$\frac{L_0}{L_1}\frac{L_1}{L_2}\frac{L_2}{L_3}\cdots\frac{L_{i-1}}{L_i} = \frac{1}{L_i}$$

(Recall that $L_0 = 1$.)

If a subtree contains L leaves, and mL randomcast packets are sent from its leaves, they constitute mL independent Bernoulli trials, each with a $1/L$ probability of success (escaping). Therefore the probability of exactly k escapees is given by the binomial probability function

$$\begin{aligned} b(x;n,p) &= b(k;mL,1/L) \\ &= \binom{mL}{k}\frac{1}{L^k}\left(1-\frac{1}{L}\right)^{mL-k} \end{aligned}$$

By Poisson's limit law [57], as n increases while np remains constant, $b(x;n,p)$ approaches the Poisson probability function $p(x;\lambda)$ with $\lambda = np$. Therefore the probability of exactly k escapees is

$$b(k;mL,1/L) \approx p(k;m) = \frac{m^k}{k!e^m}$$

If the number of randomcast packets sent by each leaf has a Poisson distribution with mean m, then the number of escapees has a true Poisson distribution $p(k;m)$, not a binomial distribution.

4.7.2 Null response traffic

The null responses received by M using X_k as exploder are the ones corresponding to null requests that were forwarded upward from X_k then bounced downward by X_{k+1}. If all members use the same ρ, null request traffic is ρ on every upward link, so X_{k+1} bounces null requests from X_k at the rate $\left(1-\frac{L_k}{L_{k+1}}\right)\rho$. The total null response traffic received by M is thus

$$\begin{aligned} \rho \quad \cdot \quad & \left[\left(1-\frac{L_0}{L_1}\right)+\left(1-\frac{L_1}{L_2}\right)+\left(1-\frac{L_2}{L_3}\right)+\ldots \right. \\ & \left. +\left(1-\frac{L_{n-1}}{L_n}\right)+1\right] \end{aligned}$$

(The last 1 is for the source, who never forwards requests, but always responds to them.)

Consider the sum

$$S = \frac{L_0}{L_1}+\frac{L_1}{L_2}+\frac{L_2}{L_3}+\ldots+\frac{L_{n-1}}{L_n}$$

The product of the terms is L_0/L_n. It is well-known that the geometric mean of a set of n positive numbers is less than or equal to the arithmetic mean, which implies that the sum is greater than or equal to n times the n^{th} root of the product. Equality obviously holds when all the terms are equal. In our case, this implies that the minimum possible value of S is

$$n \cdot \left(\frac{L_0}{L_n}\right)^{1/n}$$

This in turn implies that the maximum null response traffic received by M is $\rho \cdot (n - S + 1) = \rho z$ where

$$z = n - n \cdot \left(\frac{L_0}{L_n}\right)^{1/n} + 1 = 1 + n \cdot \left(1 - e^{\frac{1}{n}\ln\frac{L_0}{L_n}}\right)$$

$$\text{let} \quad u = \frac{1}{n}\ln\frac{L_0}{L_n}$$

$$\frac{dz}{dn} = 1 + (u-1)e^u$$

Because $u < 0$, we have $\frac{dz}{dn} > 0$, so z is maximized as n approaches infinity. Expanding z using the Taylor series for the exponential function reveals that

$$\lim_{n\to\infty} z = 1 - \ln\frac{L_0}{L_n} = 1 + \ln\frac{L_n}{L_0} = 1 + \ln N$$

(Recall that $L_0 = 1$ and $L_n = N$.)

Using the same analysis to calculate response traffic using only X_i through $X_{j<n}$ as exploder, we would obtain $\rho \ln \frac{L_j}{L_i} = \rho \cdot (\ln L_j - \ln L_i)$. That is, the null response traffic from an interval of a member's ancestry is proportional to the difference between the balanced heights of the top and bottom of the interval, or in other words, null responses are distributed evenly over the balanced heights.

4.7.3 Recovery performance

For analyzing performance during recovery, we assume that all members in the loss subtree detect the loss at time 0, that they all agree on $\hat{r}$, that the response time r is the same for every escapee, that a response arrives simultaneously at all members, and that requests and responses are not lost. Let μ denote the average μ value among the members of the loss subtree.

Number of escapees

If $r \leq \hat{r}$, a member will know before $\hat{r}$ has elapsed whether any of the burst requests escaped, so the expected number of escapees is the expected number of burst requests that escape ($\hat{r}\mu$), plus the probability that none of them escape ($e^{-\hat{r}\mu}$) times the expected number of non-burst requests that escape. The first non-burst request to escape will produce a response that causes the requests to cease, so an average of $r\mu$ requests per member get sent after the one that escaped first, of which an expected $r\mu$ escape. Therefore the total expected number of escapees is

$$x(r) = \hat{r}\mu + (1 + r\mu)e^{-\hat{r}\mu}$$

If $r \geq \hat{r}$, the story is the same, except that in the case where one of the burst requests escapes (probability $1 - e^{-\hat{r}\mu}$) an extra $(r - \hat{r})\mu$ requests will be sent before the response comes back at time r. Adding this term to the previous result yields

$$\begin{aligned} x(r) &= \hat{r}\mu + e^{-\hat{r}\mu} + r\mu e^{-\hat{r}\mu} \\ &\quad + r\mu - \hat{r}\mu - r\mu e^{-\hat{r}\mu} + \hat{r}\mu e^{-\hat{r}\mu} \\ &= r\mu + (1 + \hat{r}\mu)e^{-\hat{r}\mu} \end{aligned}$$

If the network is not working at time 0, none of the burst requests can escape. There must be a first escapee, which was sent at time t_0, say. The corresponding response is received at time $t_0 + r$. In the interim, an extra $r\mu L$ requests are sent, of which an expected $r\mu$ escape. Therefore the expected number of escapees is

$$x(r) = 1 + r\mu$$

Because $(1 + u)e^{-u} \leq 1$ for $u \geq 0$, a network failure at time 0 makes $x(r)$ larger if $r \geq \hat{r}$. If $r \leq \hat{r}$, then the network failure does not necessarily increase $x(r)$; for example, if $r = 0$, the network failure always makes $x(r)$ smaller.

Upward request traffic

Let L be the number of members below any particular edge in the loss subtree. If a member M_i below this edge is sending requests at a rate μ_i, it contributes μ_i/L upward request traffic to the edge in question. The total upward request traffic on the edge is

$$\frac{\mu_1}{L} + \frac{\mu_2}{L} + \frac{\mu_3}{L} + \ldots + \frac{\mu_L}{L} = \frac{\mu_1 + \mu_2 + \mu_3 + \ldots + \mu_L}{L}$$

which is the average μ value below the edge.

Downward request traffic

If all members use the same μ, we can prove by induction that the downward request traffic is less than 2μ on any edge. It is certainly true of the edge where the loss occurred, since there is zero downward request traffic there. Any node with only one child passes all packets straight through in both directions, so the downward request traffic on the edge below is the same as that on the edge above. For a node with $c \geq 2$ children, suppose the request traffic arriving from above is less than 2μ. That traffic will be split evenly among the children, contributing less than μ to each. In addition, a child can draw at most $\frac{\mu}{c-1}$ request traffic from each of the other $c - 1$ children, so the total for each child is less than 2μ.

Exposure

Suppose a loss occurs just above node X_i. For each escapee, the number of members who receive the response will be L_i if X_{i+1} is the turning point, L_{i+1} if X_{i+2} is the turning point, and so on, up to L_{n-1} if X_n is the turning point, and $L_n = N$ if the request makes it all the way to the source (there is no turning point). Therefore the expected number of response-recipients per escapee is

$$\begin{aligned}&\left(1-\frac{L_i}{L_{i+1}}\right)L_i+\frac{L_i}{L_{i+1}}\left(1-\frac{L_{i+1}}{L_{i+2}}\right)L_{i+1}\\&\quad+\frac{L_i}{L_{i+1}}\frac{L_{i+1}}{L_{i+2}}\left(1-\frac{L_{i+2}}{L_{i+3}}\right)L_{i+2}+\ldots\\&\quad+\frac{L_i}{L_{i+1}}\cdots\frac{L_{n-2}}{L_{n-1}}\left(1-\frac{L_{n-1}}{L_n}\right)L_{n-1}\\&\quad+\frac{L_i}{L_{i+1}}\cdots\frac{L_{n-1}}{L_n}L_n\end{aligned}$$

which simplifies to

$$\begin{aligned}L_i \quad \cdot \quad &\left[1-\frac{L_i}{L_{i+1}}+1-\frac{L_{i+1}}{L_{i+2}}+1-\frac{L_{i+2}}{L_{i+3}}+\ldots\right.\\&\left.+1-\frac{L_{n-1}}{L_n}+1\right]\end{aligned}$$

Using the same derivation as for null response traffic, we see that this expression cannot exceed $L_i \cdot \left(1+\ln\frac{L_n}{L_i}\right) = L \cdot \left(1+\ln\frac{N}{L}\right)$. Dividing by L yields the expected exposure.

Retransmission delay

The probability $p(t)$ that no response has arrived before time t is the probability that no request sent before time $t-r$ escaped. For $t<r$, no requests were sent before time $t-r<0$, so $p(t)=1$. For $r \le t \le r+\hat{r}$, the number of requests sent before time $t-r \le \hat{r}$ is $\hat{r}\mu$ per member, so $p(t)=e^{-\hat{r}\mu}$. For $t \ge r+\hat{r}$, the number of requests sent before time $t-r \ge \hat{r}$ is $(t-r)\mu$ per member on average, so $p(t)=e^{(r-t)\mu}$.

The expected value of the retransmission delay is

$$\begin{aligned}&\int_0^\infty t\,d\,(1-p(t))\\&\quad= \quad r\cdot(1-e^{-\hat{r}\mu})+\int_{r+\hat{r}}^\infty t\,d\left(1-e^{(r-t)\mu}\right)\\&\quad= \quad r+(\hat{r}+\tfrac{1}{\mu})e^{-\hat{r}\mu}\end{aligned}$$

(Hint: integrate by parts.)

Request rate selection

The end of Section 4.6.5 suggests an algorithm for choosing μ based on application-specified parameters $t_{\max}$, p, and $x_{\max}$, and asserts that if $0 < p \le e^{1-x_{\max}}$ and $x_{\max} > 1$ and $0 < r_{\mathrm{hi}} < t_{\max} < 2r_{\mathrm{hi}}$, then the response constraint overrides the delay constraint, which is to say

$$\frac{x_{\max} - 1}{r_{\mathrm{hi}}} \le \frac{\ln p}{r_{\mathrm{hi}} - t_{\max}}$$

The first two prerequisites imply

$$0 < x_{\max} - 1 \le -\ln p$$

The third prerequisite implies

$$r_{\mathrm{hi}} > t_{\max} - r_{\mathrm{hi}} > 0$$

We can divide to obtain the desired assertion.

4.8 Simulation Results

4.8.1 Simulation specification

Simulations were run using **ns** [28] on synthetic topologies. Every link is identical and symmetric:

propagation delay	10 ms
bandwidth	1.5 Mbps
queue limit	10 packets

Each link carries background traffic in both directions that originates at one end and terminates at the other:

rate	90% of capacity
packet size	1088 bytes
interarrival time	exponential

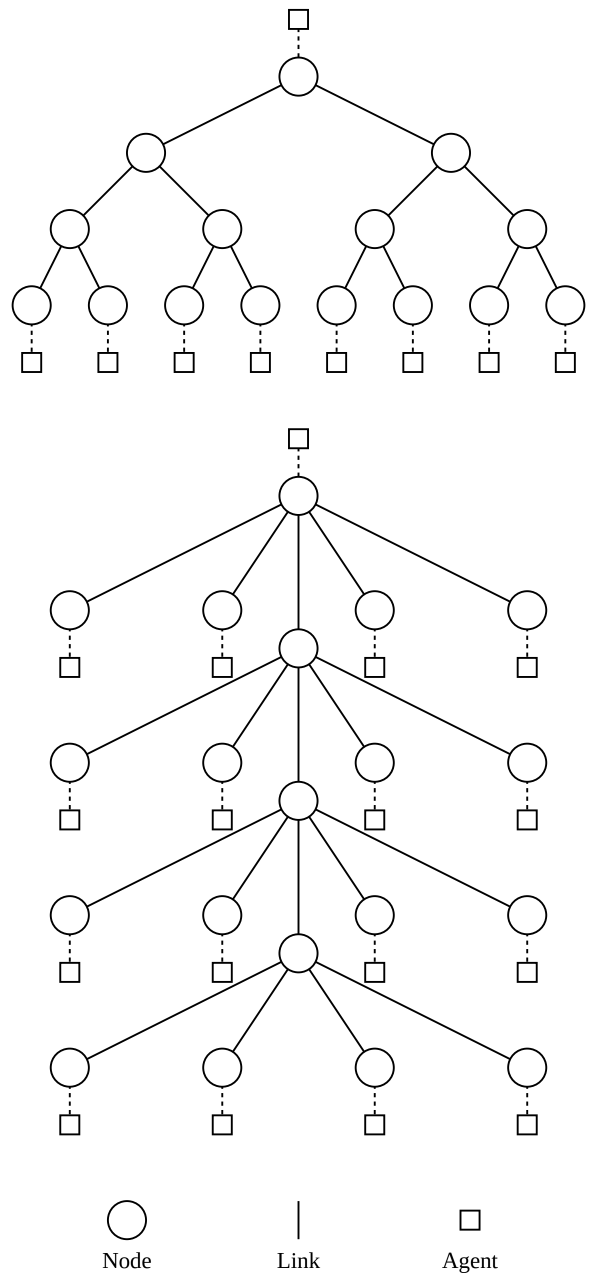

Figure 4.8: Topologies used in the simulations: balanced binary and square.

The background traffic causes an observed loss probability of 2.2% of 1088-byte data packets, and an observed queueing delay with mean 19.8 ms and standard deviation 15.0 ms.

The source sends a 1088-byte data packet (including an update) every 100 ms. Requests are 64 bytes and responses are 1088 bytes.

The members use the following parameter values:

Loss detection threshold multiplier: 0
$\hat{r}$: exponential weighted moving average with weight
$\frac{1}{8}$ for the mean and $\frac{1}{4}$ for the variance (like TCP).
r_{hi}: $\hat{r}$ mean plus 3.0 times $\hat{r}$ standard deviation.
Response suppression threshold multiplier: 3.0
Default t_{req}: $\mathit{now} - r_{\mathrm{hi}}$
Null request rate: 0
Deletion timeout: infinite

For tuning μ, three behaviors were simulated, all using the unified tuning algorithm:

application	$t_{\max}$	p	$x_{\max}$
news	600 s	10^{-9}	2.0
whiteboard	5 s	10%	4.0
game	0.2 s	1%	4.0

Two topologies were used, a balanced binary tree and an unbalanced square tree (Figure 4.8), with sizes varying between 8 and 64 members (**ns** could not handle 128).

Every run lasts 15 minutes of simulated time, during which 9000 data packets are sent.

4.8.2 Response duplication

Each curve of Figure 4.9 shows, for the game application with 64 members, the number of responses received by one member for each loss it experienced. The losses are situated along the horizontal axis, sorted by number of responses. The number of responses per loss varies between 1 and 7, with a mean of about 1.8, well below the $x_{\max}$ constraint

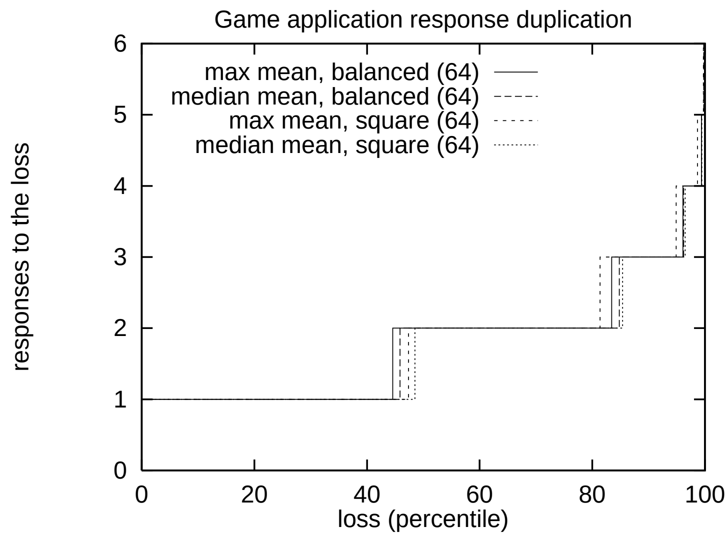

Figure 4.9: Duplicate response distribution.

of 4.0 (the application used the extra-pessimistic prediction, which did not come true). There are two members shown for each topology: the one with the maximum mean responses per loss, and the one with the median mean. The maximum and median are very close, and topology makes very little difference. The curves for the whiteboard and news applications (not shown) are similar, but shifted to the right. The whiteboard application got 1 to 4 responses per loss with a mean of 1.2 to 1.3, while the news application got 1 to 3 responses per loss with a mean of 1.1 to 1.2.

4.8.3 Request traffic

The analysis predicts that members should receive, on average, as many local requests as local responses (where "local" means pertaining to a loss that the member experienced), and that the requests should not be concentrated at a few members. For the balanced topology, the ratio of local requests to local responses received by any single member (over all losses) varied between 0.4 and 1.1, depending on the application and

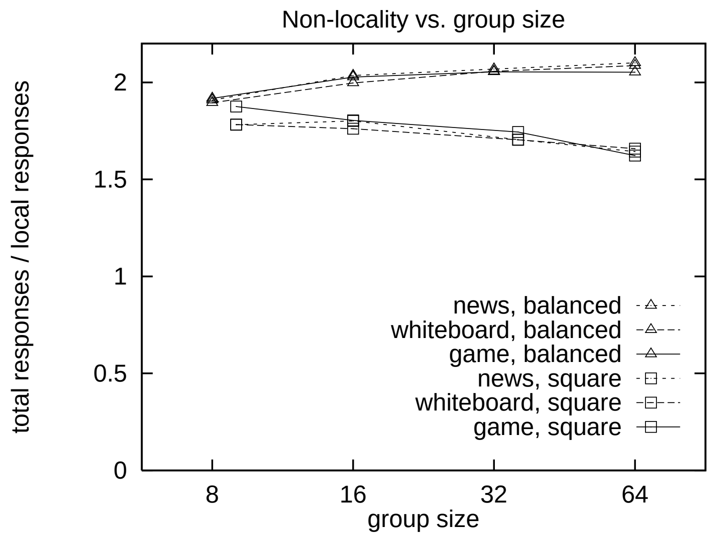

Figure 4.10: Exposure, averaged over all losses by dividing the total number of responses received by all members by the total number of local responses received by all members.

group size. For the square topology, the ratio varied between 0 and 1.4, depending on the application, group size, and position in the tree.

4.8.4 Response locality

The analysis predicts that exposure (non-locality) depends on topology, but at worst grows as $O(\log N)$. Figure 4.10 shows exposure versus group size for all three applications and both topologies. The application makes little difference. The balanced topology shows a modest increase with $\log N$, while the square topology shows a decrease, possibly because as it grows it becomes less balanced.

4.8.5 Retransmission delay

The application with a delay constraint is the whiteboard. Each curve of Figure 4.11 shows, for a group size of 64, the retransmission delay for each loss experienced by one member. The losses are situated along the horizontal axis, sorted by delay. The analy-

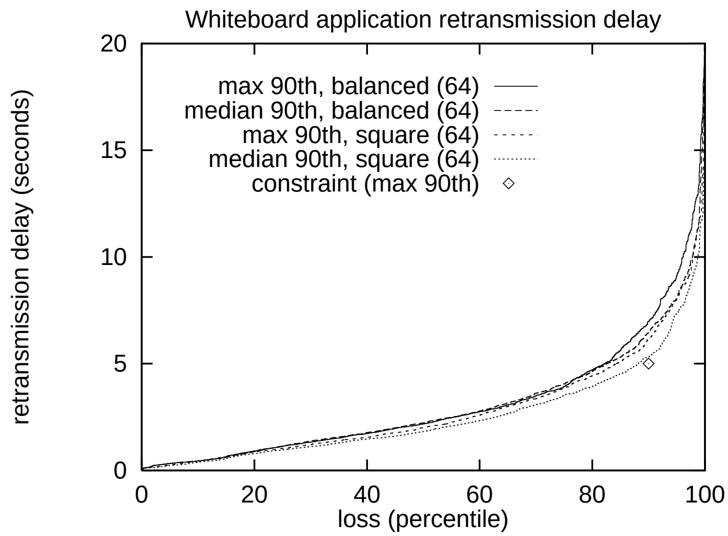

Figure 4.11: Retransmission delay distribution.

sis predicts the shape of the curves (exponential) and that they should all satisfy the delay constraint (10% chance of exceeding 5 s) by passing through the indicated point. There are two members shown for each topology: the one with the maximum delay at the 90$^{\text{th}}$ percentile, and the one with the median. The maximum and median are very close, and the topology makes little difference. The small deviation of the curves from the prediction is presumably due to the simplifying assumptions, particularly the neglect of the analysis to account for lost requests and responses.

The analysis predicts that retransmission delay is independent of N, but dependent on the response time r. Because response time grows as the topology scales up, let us *normalize* the retransmission delay by dividing by the mean round trip time between the member and the source. Figure 4.12 shows normalized retransmission delay versus group size for both topologies for the game application, the only one that attempts to maintain a constant overhead. The analysis predicts horizontal curves. The balanced topology shows a small increase with group size, presumably because paths lengthen as the topology grows,

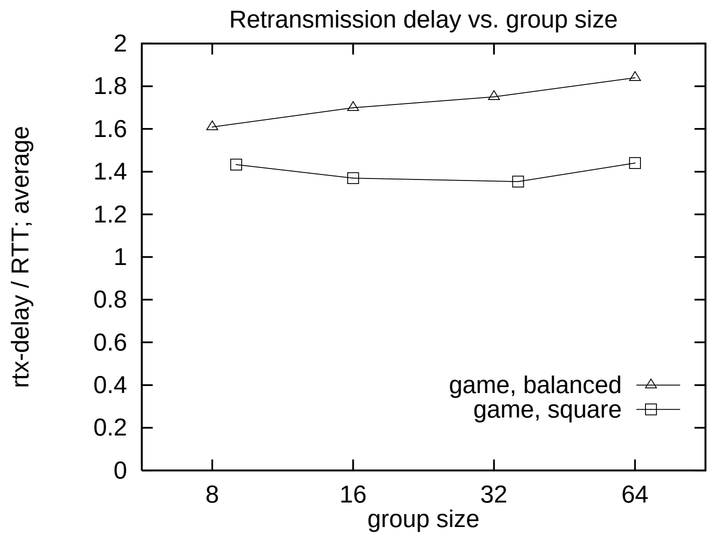

Figure 4.12: Normalized retransmission delay, averaged over every loss experienced by every member.

increasing the fraction of requests and responses that are lost, which the analysis does not account for. The square topology shows smaller delays, presumably because the average distance between members is smaller compared to the height.

Figure 4.13 shows the trade-off the game application can make between delay and overhead by choosing different values of $x_{\max}$. The third point from the right on each curve corresponds to $x_{\max} = 4.0$ as in the other graphs. For the balanced tree, this operating point gets 1.7 responses per loss compared to 1 for LMS (by construction), and the normalized retransmission delay is about 1.8 compared to 0.9 for LMS, so it is within a factor of 2 of LMS on both axes simultaneously. The unbalanced topology performs better than the balanced one on both axes, but we lack results for LMS on this topology.

4.9 Summary

Some applications need a reliable multicast service, which can be built on top of the unreliable IP multicast service. Scalability requires local recovery, which can be

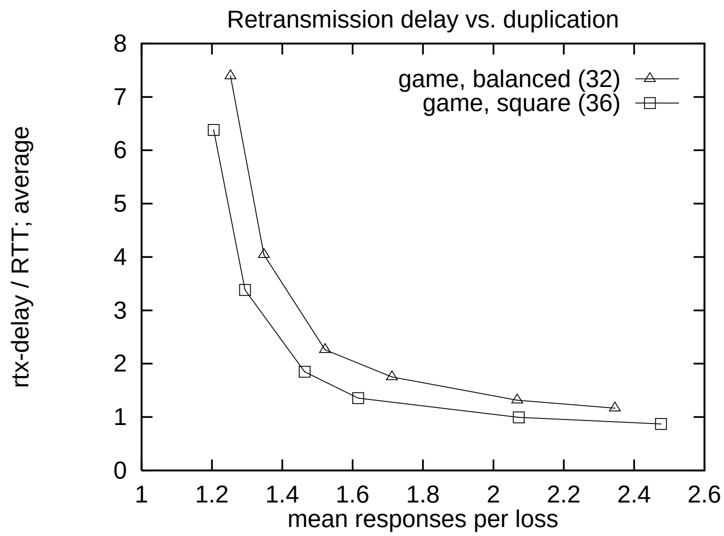

Figure 4.13: Delay/overhead trade-off.

accomplished by peer-to-peer protocols running among the group members. The LMS scheme has a particularly elegant architecture that achieves fast local recovery, but risks high sensitivity to faults because it concentrates responsibility at a few nodes. Search Party uses a similar architecture, but substitutes a randomized forwarding service in place of LMS's deterministic forwarding service. While LMS depends on a single member of a loss subtree to solicit the retransmission, Search Party is proven to spread the responsibility equally among all members of the loss subtree. In LMS there will always be some edges on which a loss affecting $O(1)$ members will consistently trigger a recovery affecting $O(N)$ members—an exposure factor of $O(N)$—but Search Party is proven to bound the average exposure to $O(\ln N)$ no matter where the losses occur. Analytic approximations indicate that the performance costs of the randomized forwarding, namely redundant and delayed retransmissions, are a constant factor, which simulations reveal to be about 2. Search Party thus serves to demonstrate how the incorporation of randomized forwarding can exchange a small reduction in performance for a large improvement in fault tolerance. Like LMS, Search Party is designed to be lightweight in its demands on multicast routers, but in the next chapter we give a second multicast loss recovery solution that makes do with even less support from the network.

Chapter 5

Rumor Mill

Multicast loss recovery protocols that send retransmissions via subcast, including RMTP, LMS, and Search Party (see Chapter 4), have two potential problems caused by their reliance on subcast. First, they all suffer from non-locality—retransmissions are often sent to members that did not experience the corresponding loss. Second, all members need permission to send multi-recipient packets that invoke the packet-duplication mechanism in the routers, but many network operators are reluctant to grant such permission except to large customers.

In RMTP and LMS both problems can be eliminated by sending retransmissions via unicast back to the requestor, rather than via subcast. The cost of this approach is slower response times, because retransmissions will end up being relayed from parent to child to grandchild, and so on, rather than going to the entire loss subtree at once. The fault tolerance issues of hierarchy-based protocols will still be present: one malfunctioning member can adversely affect a large number of other members below it in the hierarchy.

Once again, this problem can be addressed by sending requests via randomcast. We propose *Rumor Mill*, a multicast loss recovery protocol that uses a simplified variant of randomcast called *trivial randomcast* for requests and uses unicast for responses. As in

Search Party, each member experiencing a loss continually sends requests until it receives a retransmission, but in Rumor Mill the responder sends the retransmission only to the requestor, not to any subtree. The requestor, now that the loss has been repaired, can henceforth serve as a responder for incoming requests for the same data. In this way the missing data spreads like a rumor throughout the L members of original loss subtree. Turning point insertion (Section 4.3), subcast (Section 4.4), and internal response suppression (Section 4.5.5) are no longer needed. The cost of this simplification is an increase, by a factor of about $O(\log L)$, in the average retransmission delay, and in number of requests that must be sent and received. When losses are frequent the increase in request traffic is not an increase in the number of packets, but rather an increase in packet size, which is less burdensome on the network.

This chapter is organized as follows. Section 5.1 defines trivial randomcast and contrasts it with randomcast. Section 5.2 then describes the loss recovery protocol built on top of trivial randomcast, and contrasts it with Search Party. One detail of the protocol—the number of requests to bundle into each request message—is postponed until Section 5.3 where we analyze the performance and use the results as the basis of an adaptive algorithm for controlling the request bundling. Section 5.4 gives simulation results that bear out the analytic predictions of low response duplication and $O(\log N)$ overhead and delay. A summary appears in Section 5.5.

5.1 Trivial Randomcast Forwarding

Trivial randomcast is a forwarding service similar to randomcast (see Section 4.3), but much simpler. As in randomcast, a packet is forwarded along the edges of the multicast forwarding tree without backtracking, until it reaches a dead end (a member or the source). An in randomcast, the packet specifies the tree to use (not just which multicast group to use), but in trivial randomcast whenever the packet arrives on an interface of an internal node it is always forwarded to one of the node's other interfaces chosen uniformly at random. That is all there is to trivial randomcast.

Trivial randomcast is simpler than randomcast in three ways. First, it uses the uniform distribution for all forwarding decisions, never the population-weighted distribution, so there is no need to keep track of subtree populations. Second, no turning-point information is inserted into the packet. Third, as a consequence of the other two simplifications, the nodes need not know which of their outgoing interfaces is the path to the root, at least not for the purpose of forwarding trivial randomcast packets.

Section 4.3.2 explored the question of how to decide when a randomcast packet should stop traveling toward the root and be turned back away from the root. One option was the uniform distribution, where the packet arriving from one of c child interfaces has a $1/c$ chance of being forwarded to the parent interface (and thus continuing in the rootward direction). The alternative was to use a probability that depends on subtree populations, which had two advantages. First, it would diffuse responsibility for sending requests evenly among the members of the loss subtree. Second, it would allow the expected exposure (non-locality) of subcast responses to be $O(\log N)$ rather than $O(N)$. The second advantage does not apply to Rumor Mill, because responses are unicast rather than subcast, and therefore the exposure is guaranteed to be minimal (members never receive a retransmission they didn't ask for). The first advantage is sacrificed to gain the three simplifications listed above.

Section 4.3.3 explored the question of which child interface to choose when forwarding a packet away from the root, and concluded that the uniform distribution was preferable to the population-weighted distribution, because it avoids concentrating traffic on network edges, and because it tends to choose shorter (and hence faster) paths. Trivial randomcast retains those advantages.

5.2 Loss Recovery Protocol

Rumor Mill is a loss recovery protocol built on top of the trivial randomcast and conventional unicast forwarding services. It is similar to, but in most respects simpler than, the Search Party protocol described in Section 4.5.

In Rumor Mill group members detect losses exactly the same way as in Search Party (see Section 4.5.1), using timestamps sent by the source to estimate the standard deviation of the transmission delay, and waiting some multiple of that after observing a gap in the sequence numbers.

Whenever a member has outstanding (unrepaired) losses, it acts as a requestor, conducting searches with mean rate μ (measured in inverse time units). Each search consists of the emission of requests, one for each outstanding loss, which are partitioned among one or more request packets, each containing one or more individual requests. All the packets for a given search are sent at the same time, and forwarded independently via trivial randomcast. This searching behavior is the one aspect of the Rumor Mill protocol that is more complex than the Search Party protocol, in which a search sends only one packet containing only one request, and multiple outstanding losses are handled by running multiple sequences of searches independently in parallel. In Search Party, the members of the loss subtree expected to send and receive as many requests as they would receive responses, so there would not have been much to be gained from bundling multiple requests in a packet (less than a factor of two reduction in per-packet network overhead). In Rumor Mill, we will see that the expected number of requests exceeds the expected number of responses by a factor of $O(\log N)$, and it is preferable to let the network accommodate that increase by carrying larger request packets rather than more request packets.

A consequence of sharing a single sequence of searches among all losses is that the request burst behavior (which has to do with the timing of the first search after a loss is detected, see Section 4.5.4) no longer makes any sense. This aspect of the Search Party protocol is absent from Rumor Mill.

In Search Party searches were scheduled as a Poisson process, primarily to ease the analysis of the retransmission delay, but in Rumor Mill that analysis will not be applicable, because retransmissions are relayed an unknown number of times, so a more steady *quasi-periodic* schedule is used: the time between consecutive searches is uniformly distributed between $\mu/2$ and $3\mu/2$. This kind of random process was suggested in [17] to avoid synchronization effects.

Any member, or the source, who receives a request packet acts as a responder. For each request in the packet, if it has the requested data, it unicasts a response packet to the requestor, up to a limit of two responses per request packet, to avoid large bursts of large packets from the same source to the same destination, which could tend to overflow the network queues. (The bursts of request packets sent by requestors are less of a concern because they are small packets and they scatter to different random destinations.) In Search Party it was necessary to suppress retransmissions from responders who had acquired the data via a very recent retransmission, to prevent duplication of responses within the loss subtree (see Section 4.5.5). But in Rumor Mill, because each response goes only to the requestor, not to a whole subtree, there is no need for that extra logic; the rule is simply that responders always respond if they have any of the requested data.

Another simplification due to the unicast responses is in the response time estimation. Recall that the response time is the time between the emission of a request and the reception of the corresponding response (if there is one) by the requestor. In Search Party, because responses were shared among many members, it was necessary for members to remember when updates had arrived from the source and use timestamps relative to those events (see Section 4.5.2). In Rumor Mill, because each response is received only by the requestor, a straightforward absolute timestamp suffices.

Null requests and responses, for generating response time estimates during lulls between losses, work the same way in Rumor Mill as in Search Party (see Section 4.5.3). The only difference is that the number of null responses received by each member will of course be at most the number of null requests that it sends, because responses are unicast; there will be no factor of $1 + \ln N$.

Like Search Party, Rumor Mill does not prescribe any particular rules for when the source and members may stop keeping old data units available for sending retransmissions. The options suggested for Search Party are equally applicable to Rumor Mill (see Section 4.5.6).

5.3 Performance Analysis

An application using Rumor Mill must choose the rate at which it conducts searches, and for each search it must choose the number of request packets into which to partition the outstanding losses. To see how applications can make sensible choices, this section will analyze the behavior of the system. To make the analysis tractable, we neglect the loss of request and response packets, and we assume that every member has the same values for the variables defined below. These are approximations, so the derived results will be approximations. An analysis of the sensitivity to these approximations is left as future work; we instead use simulations to test the results (see Section 5.4).

5.3.1 Variables

- λ is the frequency of loss detections, the frequency at which losses are repaired, and the frequency at which responses arrive, all of which are assumed to be equal. This is a reasonable approximation, because false detections are rare, and in Rumor Mill duplicate responses are also rare, as we will see. The group members have no control over λ; it is determined by the source data rate and the network loss rate.

- r is the average request/response round-trip time, for which each member estimates the mean and variance and computes a best-guess estimate $\hat{r}$ and a rarely-exceeded upper-bound estimate r_{hi}. The group members have no control over r; it is a property of the network.

- μ is the average rate at which a member conducts searches (measured in inverse time units). The group members have complete control over μ.

- q is the average number of searches conducted after a loss is detected before it is repaired. Since each search sends one request for each outstanding loss, q is also the average number of individual requests sent per repair.

- y is the average retransmission delay; that is, the average time spent recovering from a loss, from the moment the loss is detected until the moment the retransmission is received.

The search rate μ will be tightly coupled to the average round-trip time r. Consider the effect of μ on the number of searches q and the retransmission delay y. The *search time* is the time between consecutive searches, which can be compared with the request/response round-trip time.

If the search time is always greater than the round-trip time ("slow search"), then no redundant searches will be conducted; the first successful search will be recognized as successful before the next search would have happened. If all members further increase their search times (by decreasing μ), they will simply conduct the same searches over a longer time period, increasing the retransmission delay y proportionately without affecting the number of searches q. The recovery proceeds at a pace proportional to the search rate μ.

On the other hand, if the search time is always less than the round-trip time ("fast search"), then the round-trip time will be the limiting factor, and information will spread at a rate proportional to $1/r$, independent of the search rate μ. The retransmission delay y will depend on r, not μ, and therefore the number of searches q will grow in proportion to μ (because searches are conducted at rate μ for a duration y). Some of the searches will be redundant because they will be started before earlier searches can be recognized as successful.

Clearly the fast search mode is undesirable. In Search Party members could make calculated tradeoffs between redundant searches and retransmission delay, skirting the boundary between the fast and slow modes, but in Rumor Mill the time required for the information to spread throughout the loss subtree depends on the size and topology of the loss subtree, both of which are unknown; therefore in Rumor Mill no attempt is made to calculate that tradeoff; $1/\mu$ is chosen to equal r_{hi}, which nearly minimizes the number of searches q without slowing down the recovery much more than necessary. A minimum number of searches implies a lack of redundant searches and a lack of duplicate responses. Assuming that r_{hi} is proportional to (and greater than) r, μ is proportional to (and less than) $1/r$.

Although the application can choose μ to minimize q, the application has no control over the minimum q; like r it is a property of the network conditions. As retransmissions arrive inside the loss subtree, more members become able to respond to the requests circulating throughput the loss subtree. The data spreads like an epidemic, exponentially, implying that the average time until infection (and hence the average number of searches q conducted during that time) is $O(\log L)$, where L is the number of members in the loss subtree. We make no assumptions about where losses occur; L can be as great the group size N.

The retransmission delay y is obviously proportional to the number of searches q, but it is also proportional to the round-trip time r, because the time between searches is chosen proportional to r. The lack of redundant searches implies that they do not overlap, and hence $y \geq qr$.

To sum up the variables, the external properties of the environment are the loss rate λ, the round-trip time r, and the searches per loss q. The derived properties are the search rate μ, which is inversely proportional to r, and the retransmission time y, which is proportional to both r and q. q grows as $O(\log N)$.

The average number of outstanding losses is λy, because the average depth of a queue in steady state is the average arrival rate times the average waiting time.

5.3.2 Operating regimes

The behavior of Rumor Mill can be divided into three disjoint regimes, as follows.

Regime 1: $1/\lambda > y$ and $1/\lambda > 1/\mu$

Losses are so infrequent that there is generally no more than one loss between two searches, and a loss generally gets repaired before the next loss occurs. Consequently, each search will generally have at most one request to send, and sometimes none. Request packets of size 1 will be sent at rate μ only while a loss is outstanding, and the number of request packets sent per repair is the same as the number of searches per repair q. The average rate at which request packets are sent is the search rate λ times the searches per repair q, so the average number of request packets per member in flight in the network is

$q\lambda r$, which is less than qr/y (because regime 1 assumes $1/\lambda > y$) which is less than one (because $y \geq qr$ as argued above).

$$\begin{array}{rcl} \text{request packet size:} & 1 & = O(1) \\ \text{request packet rate:} & q\lambda & = O(\log N) \\ \text{request packets per repair:} & q & = O(\log N) \\ \text{request packets in flight:} & q\lambda r < 1 & = O(1) \end{array}$$

Regime 2: $y \geq 1/\lambda > 1/\mu$

Losses occur less frequently than searches, but often enough that the time between loss detections is generally smaller than the recovery time. The average number of outstanding losses $\lambda y \geq 1$, so there is generally a loss outstanding all the time, so at least one request will be sent at each search. But since searches occur more frequently than losses, there is no reason to send more than one request packet per search—we expect each search to elicit one response or less on average. Therefore request packets will be sent at rate μ, and their average size will be the number of outstanding losses, λy, which is proportional to λqr. The average number of request packets sent per repair is the number of individual requests q divided by the request size λy, and is therefore no greater than $1/\lambda r$ (because $y \geq qr$). The average number of request packets per member in flight is $\mu r < 1$.

$$\begin{array}{rcccl} \text{request packet size:} & \lambda y & \propto & q\lambda r & = O(\log N) \\ \text{request packet rate:} & \mu & \propto & 1/r & = O(1) \\ \text{request packets per repair:} & q/\lambda y & \leq & 1/\lambda r & = O(1) \\ \text{request packets in flight:} & \mu r & < & 1 & = O(1) \end{array}$$

Regime 3: $1/\mu \geq 1/\lambda$

Losses occur at least as frequently as searches ($\lambda \geq \mu$), so every search generally has at least one request to send. One request packet generating many responses would be undesirable, because it could clog the network. The upper limit of regime 2 was one response per request packet, so we will aim for that (a request packet rate of λ). The λy requests per search must therefore be divided into λ/μ request packets of size $\lambda y/(\lambda/mu) = y\mu \propto q$. The average number of request packets per member in flight is λr, which is more than one, unlike regimes 1 and 2, but the number of request packets per response is the minimum possible (just one).

$$\begin{array}{rcl} \text{request packet size:} & y\mu \propto q & = O(\log N) \\ \text{request packet rate:} & \lambda & = O(1) \\ \text{request packets per repair:} & 1 & = O(1) \\ \text{request packets in flight:} & \lambda r & = O(1) \end{array}$$

5.3.3 Credit-based adaptation

The protocol described in Section 5.2 left open the question of how many request packets to divide a search into. The primary consequence of the above analysis is guidance for that decision process, as follows.

A member operating in regime 1 or 2 generally sends at most one request packet per search, and sends more request packets than it detects losses, while in regime 3 it generally sends multiple request packets per search, and sends as many request packets as it detects losses. The determiner is whether losses happen more frequently than searches. An easy way for the member to adaptively figure this out is to accumulate one credit for each loss detected, and consume one credit for each request packet sent. If the member needs to send a request packet and has less than one credit in the account, it goes ahead and sends the packet and sets the credit to zero (this is typical for regimes 1 and 2). But if it has more than one credit (typical for regime 3), it uses the extra credit to divide the search into multiple request packets, one for each whole credit, or one for each outstanding loss if there are too many credits to use in one search. The outstanding losses are distributed evenly (as evenly as possible) and randomly among the outgoing request packets.

In order to smooth fluctuations in loss rates, a member does not spend the full credit balance on each search—it spends a fraction of it, 1/8 for example. This builds up a reserve of credit until 1/8 of the reserve is equal to the average earnings per search interval. Therefore a member still uses credit as fast as it earns it, but fluctuations in the loss rate are dampened. The credit will not grow infinitely, because when a member has plenty of credit it uses a separate request packet for each individual request, and each loss leads to at least one individual request.

If a member receives a data packet (an original transmission from the source, not a retransmission) for a missing sequence number, it should deduct the credit it accumulated for the erroneous loss detection.

5.4 Simulation Results

5.4.1 Simulation specification

Simulations were run using the same setup as described in Section 4.8.1 except for the following differences. r_{hi} is the $\hat{r}$ mean plus twice (rather than thrice) the $\hat{r}$ standard deviation. The source sends data packets quasi-periodically rather than periodically. The other differences are motivated by the need to test the three regimes describe in Section 5.3.2, which call for a wide range of loss rates. Although the loss *rates* (losses per second) must vary widely, the loss *probabilities* (fraction of packets that are lost) must not get too small (because then the simulation would take too long to run) or too large (because the protocol is not designed for large loss probabilities). This can be achieved if the source data rate can be varied without affecting the queueing behavior, which is accomplished by using zero-size packets for the protocol, so that only the background traffic packets count in the queue. To make up for the lack of transmission delay for zero-size packets, the propagation delay is increased from 10 ms to 13.1 ms, so that the minimum request-response round trip time is the same as before. Since the loss rate varies so widely, each run lasts not 15 minutes but rather until the members have detected an average of 1000 losses each. The link rate and queue limit are both doubled, to 3.0 Mbps and 20 packets, so that the maximum queueing delay is the same as before but a full queue endures half as long, not long enough to drop two consecutive data packets even at the higher data rates. The background traffic rate is 98% of capacity, which causes queueing behavior similar to that observed in the Search Party simulation: a loss probability of 1.6% and a queueing delay with mean 24.6 ms and standard deviation 16.7 ms.

Three sets of simulations were run, designed to operate in each of the three regimes describe in Section 5.3.2, by adjusting the data rate of the source in response to the loss rates observed by the members and the search intervals chosen by the members. For regime 1 (low loss rate), the source data rate was adjusted to make the average loss interval $1/\lambda$ equal to four times the average retransmission delay y. For regime 3 (high loss rate), the source data rate was adjusted to make $1/\lambda$ equal to one-fourth the average search interval $1/\mu$. For regime 2 (medium loss rate), the source data rate was adjusted to make $1/\lambda$ equal to the geometric mean of y and $1/\mu$.

5.4.2 Requests per repair (q)

Section 5.3.1 introduced the variable q, the average number of searches between loss and repair, or equivalently the average number of individual requests sent per repair, and argued that it would be $O(\log N)$ where N is the group size. The simulation revealed, however, that the tree height is a much better predictor of q than the group size, but of course in any truly scalable network topology the tree height is proportional to $\log N$ anyway. Figures 5.1 through 5.3 show q versus tree height for the balanced binary and square topologies, for each of the three operating regimes. Other simulation runs (not shown) with a star topology (height $=1$) and a chain topology (height $=N$) likewise showed q to be proportional to the tree height.

The difference in q from one regime to another is due to the limit of two responses per request packet. In other simulation runs (not shown) with unlimited responses per

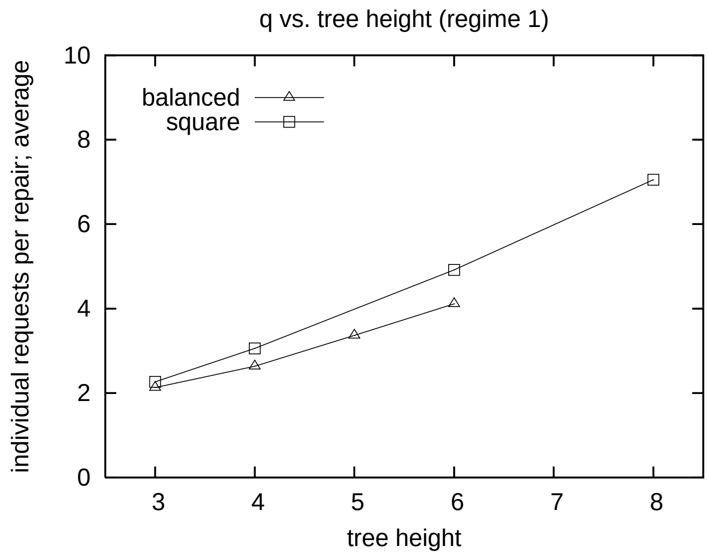

Figure 5.1: q versus tree height for regime 1 (low loss rate).

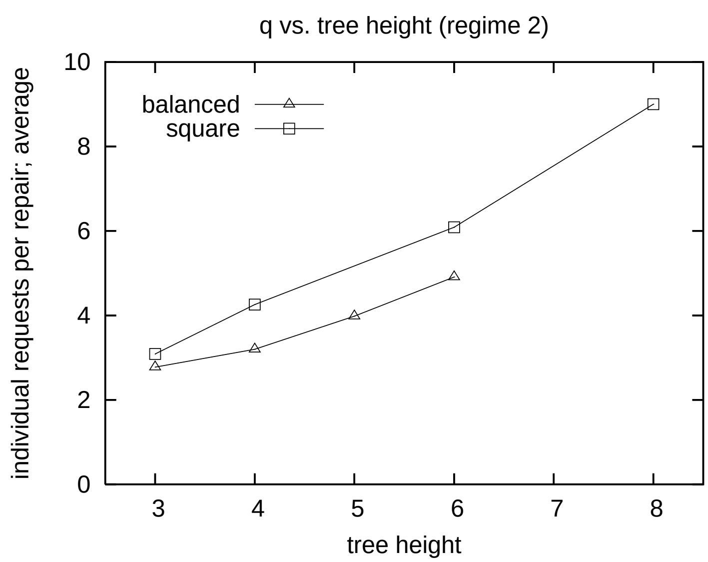

Figure 5.2: q versus tree height for regime 2 (medium loss rate).

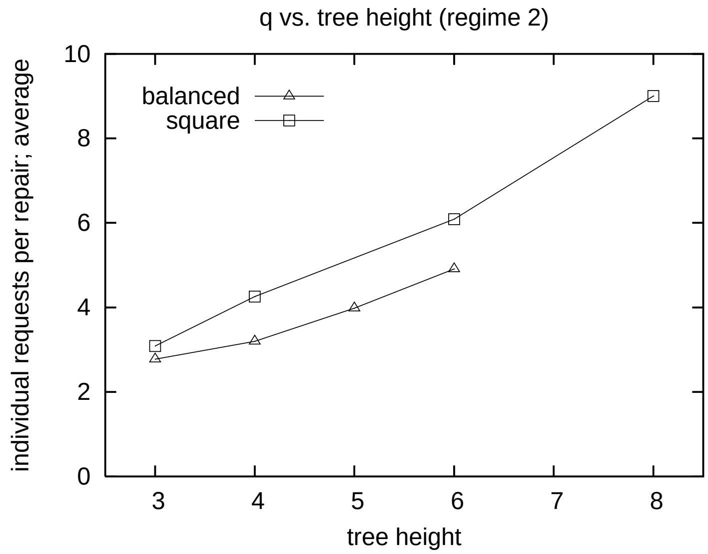

Figure 5.3: q versus tree height for regime 3 (high loss rate).

request packet, the q curves were nearly identical across all three regimes. The simulation setup treats response packets as zero-length, whereas in reality large bursts of large packets from the same source to the same destination would be likely to cause congestion, so the two-response limit is prudent, even if it increases q somewhat when loss rates are high.

5.4.3 Request packets

As the group size increases, the growth in the number of individual requests per repair q can take the form of more request packets, more requests per packet, or both. In regime 1 (low loss rate) there is about one request per packet (the growth is all in the packet count), while in regime 3 (high loss rate) there is about one request packet per repair (the growth is all in the packet size). In regime 2 (medium loss rate) the growth is split between packet count and packet size in such a way that the number of request packets per repair is supposed to be less than $1/\lambda r$, which is the average number request-response round

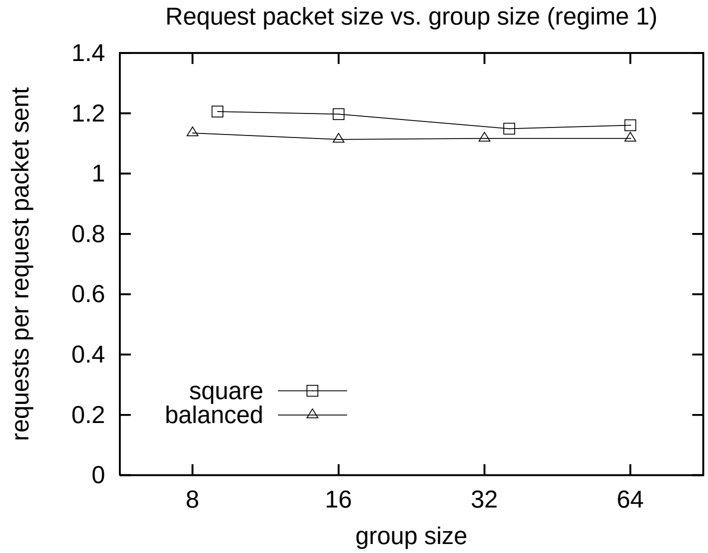

Figure 5.4: Requests per request packet versus group size for regime 1 (low loss rate). In this regime the credit-based adaptation algorithm aims for one request per packet regardless of the group size, leaving the growth of q to be accommodated with increasing numbers of request packets.

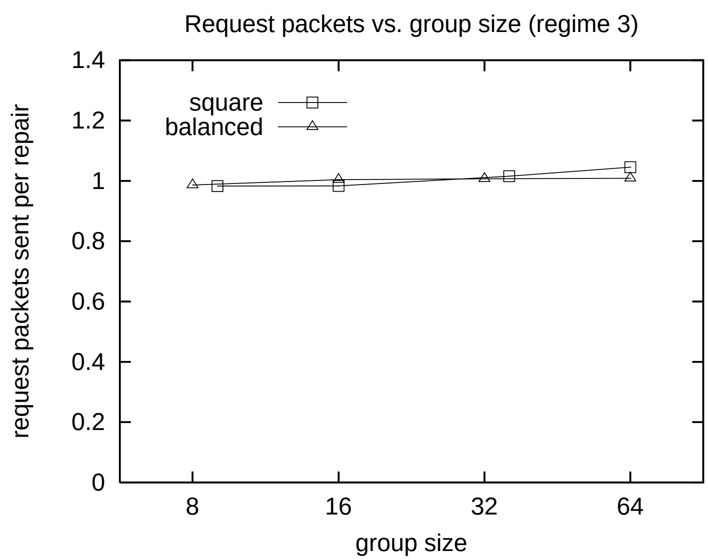

Figure 5.5: Request packets per repair versus group size for regime 3 (high loss rate). In this regime the credit-based adaptation algorithm aims for one request packet per repair regardless of the group size, leaving the growth of q to be accommodated with larger request packets.

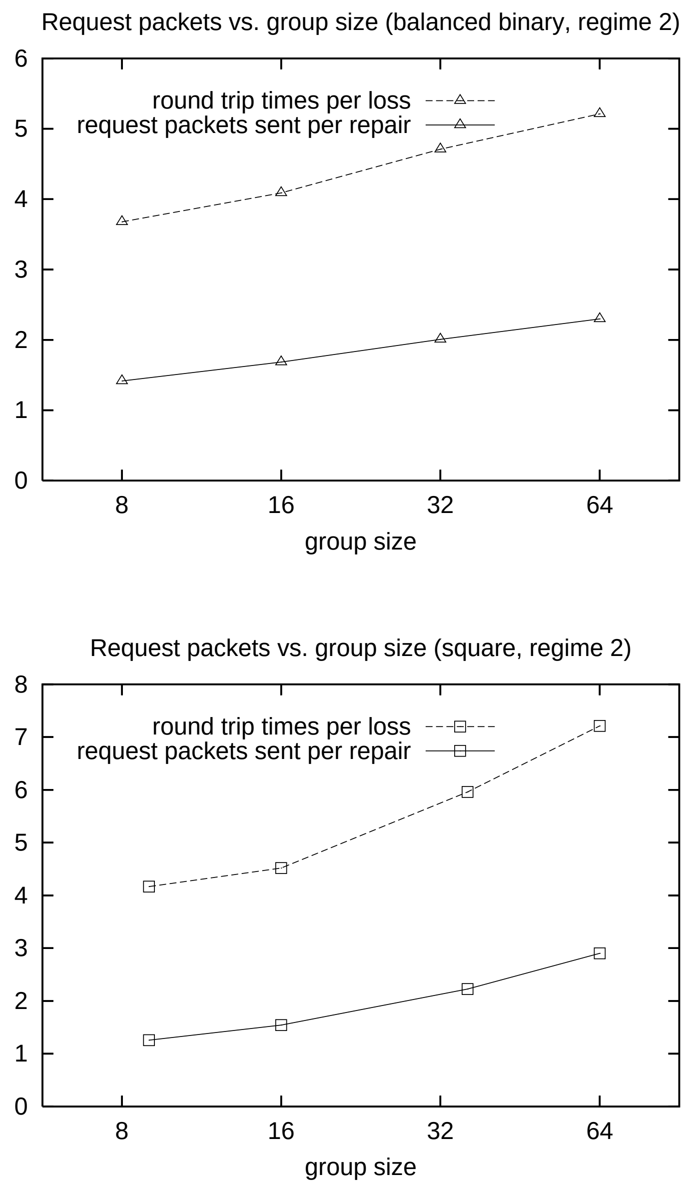

Figure 5.6: Request packets per repair versus group size, compared with round trip times per loss, for regime 2 (medium loss rate). In this regime the credit-based adaptation algorithm accommodates the growth of q by increasing both the number of request packet and their size, aiming keep the number of request packets per repair below $1/\lambda r$ (the number of request-response round trip times per loss).

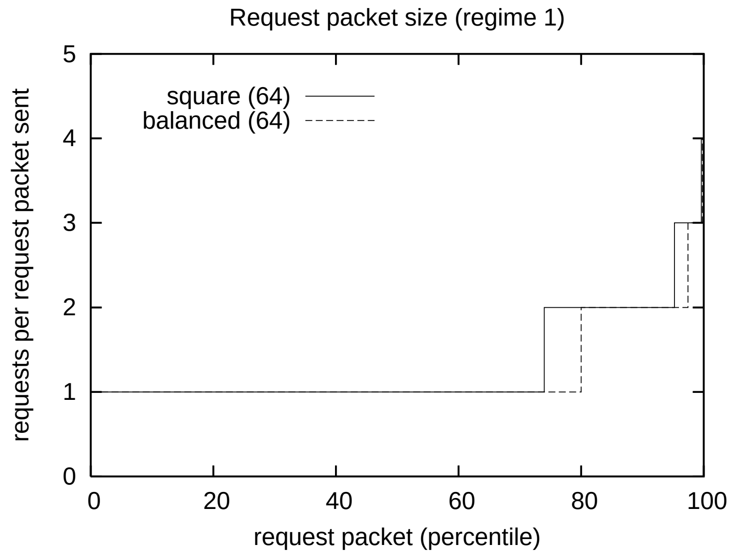

Figure 5.7: Distribution of request packet sizes for regime 1 (low loss rate) with 64 group members.

trips that fit between consecutive loss events. It is the job of the credit-based adaptation algorithm of Section 5.3.3 to achieve those behaviors in the three operating regimes, and Figures 5.4 through 5.6 show that it succeeds.

Figures 5.7 through 5.9 show that the request packet sizes within a given run are fairly consistent.

The randomized forwarding of the request packets prevents them from being concentrated. For any single member, the ratio of request packets received to request packets sent never exceeded 1.1 for the balanced binary topology, and never exceeded 1.4 for the square topology.

5.4.4 Response duplication

Section 5.3 claimed that duplicate responses would be rare; that is, there would be hardly more than one response per loss. Each curve of Figure 5.10 shows, for regime 3 (high

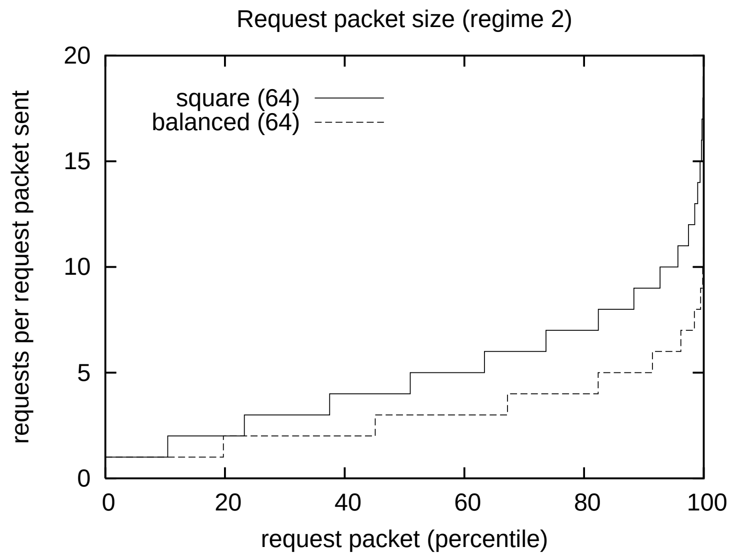

Figure 5.8: Distribution of request packet sizes for regime 2 (medium loss rate) with 64 group members.

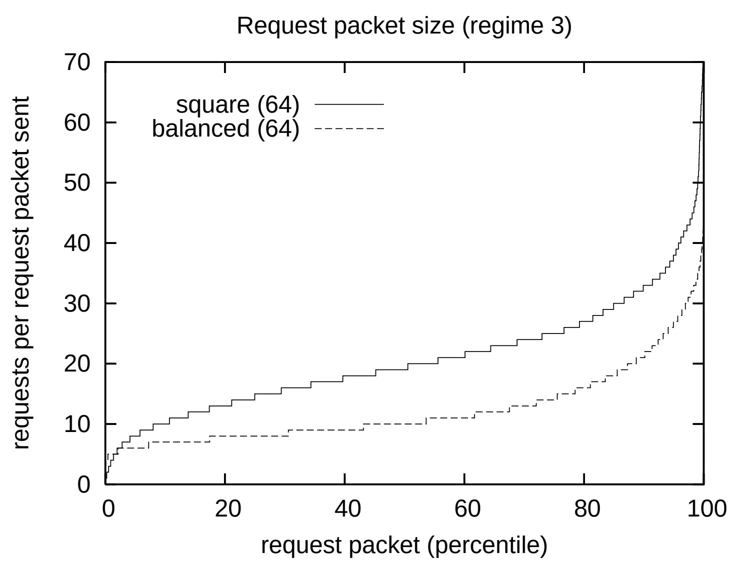

Figure 5.9: Distribution of request packet sizes for regime 3 (high loss rate) with 64 group members.

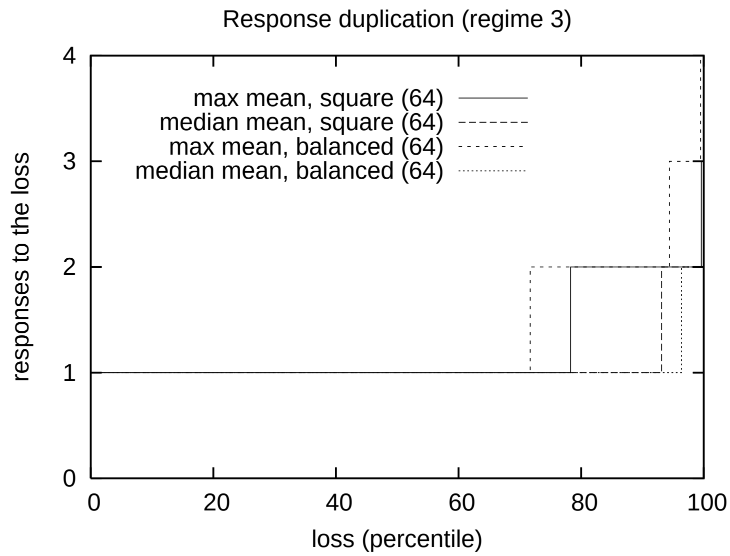

Figure 5.10: Duplicate response distribution for regime 3 (high loss rate) with 64 group members.

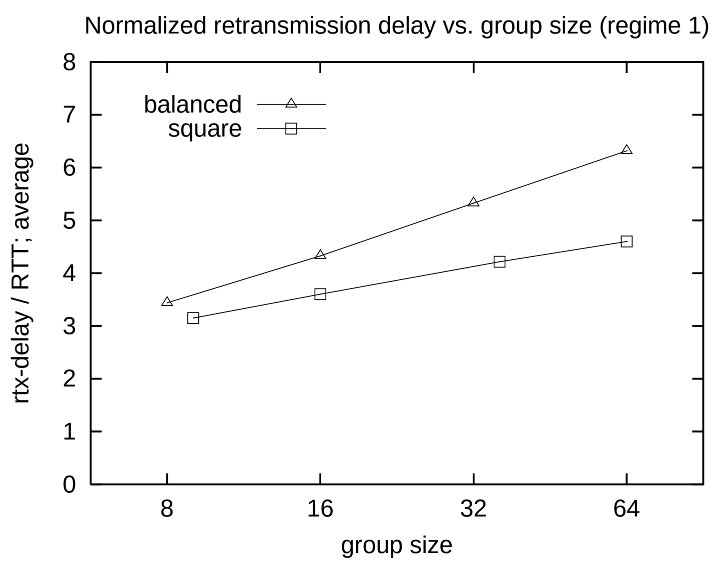

Figure 5.11: Normalized retransmission delay, averaged over every loss experienced by every member, for regime 1 (low loss rate).

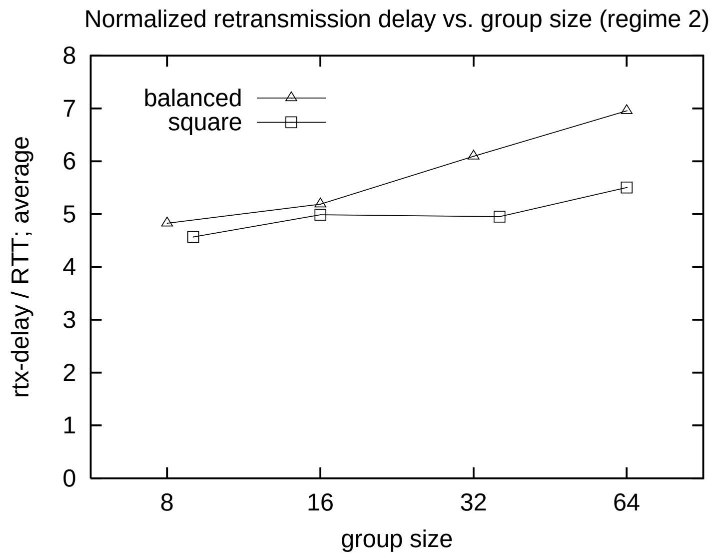

Figure 5.12: Normalized retransmission delay, averaged over every loss experienced by every member, for regime 2 (medium loss rate).

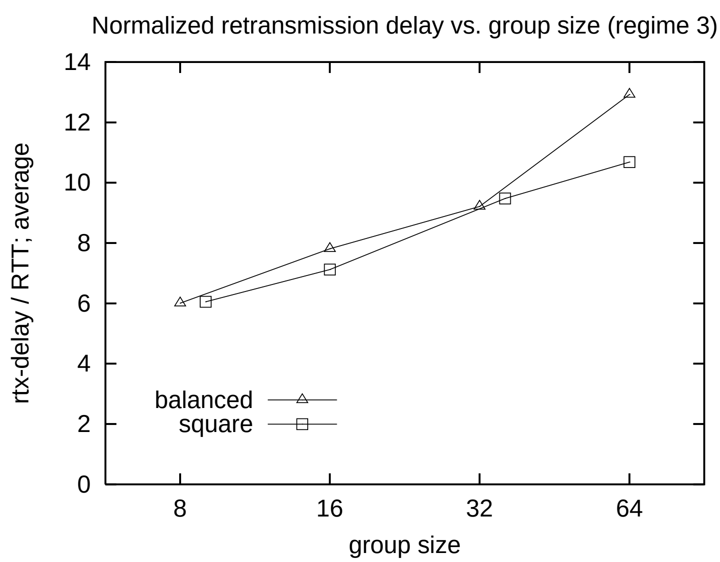

Figure 5.13: Normalized retransmission delay, averaged over every loss experienced by every member, for regime 3 (high loss rate).

loss rate) with 64 group members, the number of responses received by one member for each loss it experienced. The losses are situated along the horizontal axis, sorted by number of responses. There are two members shown for each topology: the one with the maximum mean responses per loss, and the one with the median mean. The average number of responses per loss over all members was about 1.1 for both topologies and all three regimes. The maximum average for any single member was about 1.3 for regime 3, 1.2 for regime 2 (not shown), and 1.1 for regime 1 (not shown).

5.4.5 Retransmission delay

Figures 5.11 through 5.13 show normalized retransmission delay versus group size, averaged over every loss experienced by every member in the group, for both topologies for each of the three regimes. Compared to the corresponding graph for Search Party in Figure 4.12 it illustrates the main disadvantage of Rumor Mill: the retransmission delay grows as $\log N$ rather than being (nearly) constant.

5.5 Summary

Chapter 4 presented Search Party as a demonstration of using randomized forwarding to improve fault tolerance in a peer-to-peer multicast loss recovery protocol. Search Party relies on some rather comprehensive support in the multicast routers: new state (subtree population counts), the ability to modify packets at turning points, and the willingness to accept one-to-many packets from all members, not just sources. Rumor Mill is a similar peer-to-peer multicast loss recovery protocol that makes do with minimal router support: just a uniform random number generator, no state, no packet modification, and no one-to-many packets from any node except the source. The price of this simplification is slower retransmissions and larger request packets, by a factor of about $O(\log N)$. Rumor Mill thus serves as further evidence of the practicality of trading performance for fault tolerance. The next chapter will continue this trend with another peer-to-peer protocol, this time for a distributed hash table, that needs no special router support at all.

Chapter 6

Tangle

The previous two chapters examined peer-to-peer protocols for multicast loss recovery, and showed how their fault tolerance could be improved through the use of randomized forwarding. In this chapter we turn our attention to a different sort of peer-to-peer protocol, a shared read/write data store called a *distributed hash table* (DHT), which is spread across many nodes to achieve greater availability, capacity, and aggregate throughput than an individual node. Once again we propose to use randomized forwarding, this time at the application layer rather than the network layer, to make the protocol more fault tolerant, and we offer the Tangle protocol as an instantiation of this approach.

As described in Section 2.2, a distributed hash table has two layers, a lower routing layer that maps labels to nodes by forwarding messages to nodes that are logically closer to the label, and an upper hash table layer that maps labels to values stored at the nodes. The research presented in this chapter is focused primarily on the routing layer, because it is the routing layer that faces the choice between deterministic versus randomized forwarding. Randomized forwarding is by no means the only effective approach for improving fault tolerance in a distributed hash table. Detecting faults and adjusting the routing tables accordingly can be effective at reducing failures of the routing layer (that is the technique used by the Internet routing layer). Another effective technique that can be applied to any distributed hash table is to introduce redundancy at the hash table layer, such as by running it on top of multiple independent instances of the routing layer, or by storing values at multiple nodes or under multiple labels. But it is interesting that randomized forwarding can improve the fault tolerance of the routing layer itself, and that this single simple technique is effective against a wide variety of faults without detecting them.

Background material regarding distributed hash tables and various schemes for building them has already appeared in Section 2.2; therefore this chapter dives straight into a description of Tangle itself. Section 6.1 details each of the layers of the Tangle protocol: the peer discovery sublayer, which constructs the routing tables; the forwarding sublayer, which uses the tables to forward messages; and the hash table layer, which uses messages to read and write a hash table. Section 6.2 describes the experimental framework we wrote and the experiments we ran to measure the performance of the protocol. The results are presented in Section 6.3. The main result is that for a wide variety of faults (none of which are detected), randomized forwarding greatly reduces the failure rate, typically by two orders of magnitude. The chapter finishes with a summary in Section 6.4.

6.1 Tangle Protocol

Figure 6.1 shows the conceptual layers of the Tangle protocol. The routing layer is divided into two sublayers. The standalone *peer discovery sublayer* constructs the routing tables by exchanging `ping` and `pong` messages in a three-way handshake with peers (messages are not forwarded), using a gossip-style protocol inspired by Name-Dropper [19]. The *forwarding sublayer* relays messages from one node to another, using the routing table to calculate the next hop.

put(*label*,*version*,*location*) adds $\langle$*version*,*location*$\rangle$ to *label*'s set
get(*label*) returns random $\langle$*version*,*location*$\rangle$ pairs from *label*'s set

hash table layer	soft-state redundant storage, implosion prevention, newer-version notification; at each hop store/retrieve data, send response, and/or terminate forwarding

send(*label*,*message*)

routing layer	forwarding sublayer: choose next hop for relaying messages
	peer discovery sublayer: construct soft-state peer information (routing table)

Figure 6.1: Tangle protocol layers.

Because it is the routing layer that uses randomized forwarding, the hash table layer need not be understood in order to appreciate the experiments and results later in this chapter, but for completeness it is covered in this section.

The hash table layer maintains the data store and uses the routing layer to relay post messages that store values and query messages that retrieve values. Both post and query messages are routed in the same way toward a target label. The hash table layer supplies to the forwarding sublayer a *forwarding hint* that influences the distribution use to select the next hop from the choices available in the routing table (uniform random, skewed random, or deterministic). The hash table layer also decides whether to continue relaying a message toward the target label or to stop it early (to prevent implosion). Values are returned via report messages, which do not need to be routed like post and query messages because the destination is a known network address, not a label.

The interface exported by Tangle's hash table layer is more specialized than the general distributed hash table interface, because it was designed to support a particular application—a serverless web, which would be like the World Wide Web except that the

Uniform Resource Identifiers (URIs) would not contain server names, so that links would not break when documents moved to different servers, and documents that became popular could automatically be replicated on enough nodes to carry the shared load, and the availability of a document could greatly exceed the availability of the individual nodes that serve it. Each document (or in general, any static resource) would have a label (which could be calculated from its URI), and all the nodes that serve a given resource would advertise that fact by storing pointers to themselves in the hash table under that label. Nodes looking for the resource could learn where to find it by looking up the label in the hash table. In support of this application, instead of an arbitrary *value* the Tangle interface has a resource location and some version information, which allow the hash table layer to detect nodes advertising old versions and notify them of locations of newer versions.

The remainder of this section describes Tangle at a level of detail sufficient for an in-depth but abstract understanding of the protocol. For detail sufficient to implement the protocol, refer to the Tangle specification [8].

6.1.1 Peer discovery

Tangle's peer discovery protocol is inspired by Name-Dropper [19], a gossip-style protocol that allows a set of nodes to each learn of all the other nodes. In Name-Dropper, each node initially knows some set of peers, in such a way that the graph is connected. Each node continually chooses a known peer uniformly at random and sends its entire list of known peers, and soon every node knows every other node, even if the initial graph was very weakly connected (like a linear chain).

In Tangle, nodes do not want complete knowledge of their peers, because that would not scale. Nodes merely want a random sample of their peers; therefore they include only a few peers (chosen randomly) in each sent list, and they forget peers (chosen randomly)

prefix length	pattern	confirmed	unconfirmed
0	**0**$*$	[set of peers]	[set of peers]
1	1**1**$*$	[set of peers]	[set of peers]
2	10**0**$*$	[set of peers]	[set of peers]
3	101**0**$*$	[set of peers]	[set of peers]

Figure 6.2: Peer data for a node whose label begins 1011. The column of confirmed sets is used as a routing table by the forwarding sublayer.

to keep their peer sets bounded. Intuitively, if Name-Dropper causes the peer sets to quickly become complete, then this adaptation should cause the peer sets to quickly become random. A proof of this conjecture is left as future work.

Another difference from Name-Dropper is that a Tangle node organizes its peers into two classes: unconfirmed peers (whom it has heard of but not heard from) and confirmed peers (whom it has heard from first-hand). Each node continually chooses an unconfirmed peer at random and handshakes with it. The two nodes add each other to their confirmed sets, and each adds a few of the other's confirmed peers to its own unconfirmed set. Only confirmed peers are used for forwarding, because unconfirmed peers are much less reliable. An unconfirmed peer might be a lie (sent by a malicious node), or a mistake (sent by a buggy node), or unreachable due to network anomalies (even if sent by a node that could reach it). By requiring each peer to be confirmed before it can be passed to another node, the protocol prevents a peer from being passed around for a long time after it goes down.

The final difference from Name-Dropper is that the confirmed and unconfirmed peer sets are each partitioned into several sets, one for each label prefix length. Each node's label is a well-known function of its network address, a hash function that yields uniformly distributed bit strings. The confirmed and unconfirmed sets at level k contain peers with labels whose longest common prefix (compared to the node's own label) has length k. For example, Figure 6.2 shows the first four levels of the peer data for a node whose label begins

1011. A node can be viewed as belonging to many communities: the level 0 community of all nodes, the level 1 community of nodes that share a 1-bit prefix with it, the level 2 community of nodes that share a 2-bit prefix with it, and so on. The node effectively participates in peer discovery with each of those communities, and thereby maintains a random sample of peers from each community, with the same refresh rate at every level.

This multi-level peer discovery is actually a single combined process. A node quasi-periodically[1] chooses a non-empty unconfirmed set uniformly at random (or cycles through them in a round-robin fashion), then chooses a peer from that set uniformly at random, deletes that peer, and initiates a three-way handshake with it by sending a `ping` message, which triggers a `pong` response message containing an embedded `ping`, which triggers a `pong` response without an embedded `ping`. An opaque cookie contained in each `ping` is echoed in the corresponding `pong`. By using the cookie to carry a timestamp and a keyed one-way hash covering the timestamp and the peer's network address, each node can verify that an arriving `pong` is not unsolicited but is a response to its own `ping`, without having to keep any record of the outstanding `ping` (which would have to be garbage-collected if the response never came).

Each `pong` contains a few peers selected from the sender's confirmed sets. Achieving a uniform refresh rate at all levels depends on the proper distribution of those selections. Consider two nodes whose labels have a common prefix of length k. They belong to the same level k community, but they also belong to the same level $k-1$ community, and the same level $k-2$ community, and so on. When they exchange peers, they could be performing peer discovery in the context of any of those communities. Therefore, when selecting a confirmed peer to put into a `pong` message, a node first needs to decide which community to draw it from. The level $k-1$ community is twice as populous as the level k community,

[1] The time between successive events is a uniform random variable between $T/2$ and $3T/2$, where T is the nominal period. This is suggested in [17] to avoid synchronization effects.

so we expect to handshake with a given member half as often in that context; the level $k-2$ community is twice again as populous, so we expect to handshake with a given member half again as often in that context; and so on. Therefore the procedure for choosing the community is to start at level k and flip coins and subtract one level for each head until a tail comes up (or until level 0 is reached). Suppose the node has selected the community at level j. Now a peer needs to be chosen uniformly at random from that community. Members of that community all have labels whose first j bits agree with the node's own label, but each subsequent bit has a 50% chance of agreeing with the node's own label. Therefore more coins are flipped, and starting from level j one level is added for each head until a tail comes up. At this final level a peer is chosen uniformly at random from the confirmed set, and included in the pong message. The whole process is repeated, starting back at level k, for each confirmed peer to be included in the pong message.

The peers in the confirmed and unconfirmed sets do not stay there very long, because confirmed peers are deleted whenever they reach an age limit, and unconfirmed peers are deleted whenever they are probed. If the network were to become partitioned, and stay partitioned long enough, all nodes would delete all their peers from the other side of the partition, and the two populations would never find each other again, even after the network healed. To guard against this outcome, each node maintains a list of historic peers. Quasi-periodically, the node selects a confirmed peer at random, adds it to the front of the list, flips a (possibly weighted) coin until it comes up heads, and deletes the i^{th} peer from the list, where i is the number of coin flips. If i exceeds the length of the list, no peer is deleted. This procedure causes the list to grow as $O(\log \mathit{time})$, and causes the peers in the list to have ages such that $\log \mathit{age}$ is uniformly distributed. Although nodes normally send ping messages to unconfirmed peers, they occasionally send them to historic peers instead (and the historic peer, unlike an unconfirmed peer, is not deleted in the process). Some historic peers will be so old that they no longer exist, and some will be so young that they don't predate the partition, but because there are historic peers of all ages, some peer is likely to contact a historic peer from the other side of the partition, at which point the graph will be connected once again, and will start converging to a well-connected graph.

6.1.2 Forwarding

When a message is being forwarded toward a given target label, the goal is to reach the node whose label is the best match for the target label; that is, it has the longest common prefix with the target label. (If there is a tie among several nodes, the goal is to reach any of them.) Each node chooses a next hop from among a set of peers whose labels are strictly better matches than its own label. The node determines the set of peers by comparing its own label to the target label and finding the length k of the common prefix. Because of the way the peers are organized in the routing table, it is exactly the peers at level k that are strictly better matches. If the confirmed set at level k is empty the node concludes that the best match is itself and forwarding ceases; otherwise the node selects a peer from the set and forwards the message there.

The selection of the next hop from among the set of candidates is influenced by a *forwarding hint* contained in the message, which is set by the original sender of the message and not modified as the message is forwarded. The hint requests one of four possible selection methods:

uniform The next hop is chosen uniformly at random from among the candidate peers.

reduce hop count The next hop is chosen randomly from among the candidate peers, using a nonuniform distribution that will tend to reduce the total number of hops to the target. This can be accomplished by favoring peers whose labels have longer common prefixes with the target label.

reduce total time The next hop is chosen randomly from among the candidate peers, using a nonuniform distribution that will tend to reduce the total time until the target is reached. This can be accomplished by favoring peers with a greater estimated rate of progress. A peer's rate of progress can be estimated as the increase in common prefix length divided by the network delay. The network delay to the peer can be estimated by the round-trip-time that was observed during the handshake that confirmed the peer.

minimize total time The next hop is chosen *deterministically* from among the candidate peers using a criterion that will attempt to minimize the total time until the target is reached. This can be accomplished using the same estimation as the previous hint and taking the favoritism to the extreme.

The Tangle specification allows implementors to design their own nonuniform distributions. Our implementation used for the experiments of Section 6.2 makes a nonuniform random choice from among n items by selecting $\lceil\sqrt{n}\rceil$ items uniformly at random and choosing the best of those.

The randomization requested by the first three forwarding hints becomes useful when a higher layer sends a message multiple times, because each instance of the message can take a different path and hopefully avoid whatever fault may have blocked its predecessors. If the messages are designed to be idempotent (harmless if received more than once), the higher layer need not wait to determine the success or failure of one instance before sending the next; the resending rate is simply a tradeoff between overhead and the expected time to success.

The higher layer could send all instances of the message using the same forwarding hint, but in practice a hybrid strategy would be used: the first instance of the message would request deterministic forwarding, and subsequent instances would request randomized forwarding. The hybrid strategy retains the benefits of randomized forwarding in the presence of faults, and retains the performance of deterministic forwarding in the absence of faults.

The hybrid strategy used in our implementation sets the forwarding hint as follows: the first instance of the message carries the “minimize total time” hint, the next several instances carry the “reduce total time” hint, and all subsequent instances alternate between the “reduce hop count” and “uniform” hints. The rationale is that as time passes, speed becomes less of a concern. The alternation in the final phase seeks to hedge our bets between two tactics for improving fault tolerance: the “uniform” hint increases the diversity in the paths, while the “reduce hop count” hint decreases the number of opportunities for failure.

6.1.3 Hash table layer

The hash table layer is orthogonal to the research issues of forwarding, and was not involved in the experiments; therefore it is not necessary to read this section to follow the remainder of the chapter. It may however provide perspective on the design of the routing layer, because the two layers were designed jointly.

The job of the hash table layer at each node is to maintain a *location map*, which maps labels to locations. Each label is presumed to be the hash of a resource (computed by a higher layer), and each location consists of a network address (of a node believed to have the resource) and some resource parameters: a timestamp to serve as a version number, an out-of-date flag indicating that a newer version of the resource is believed to exist, and a completeness measure indicating what percentage of the resource is present, in case the resource is still in the process of being transferred to the node (this enables pipelined distribution of a large resource). Each label is associated with a set of locations.

Locations are added to the map when they arrive in *advertisements*, which are carried in `post` messages. The location information is soft-state—it is discarded after a while and must therefore be continually refreshed. A higher layer that has a resource to offer must continually send an advertisement containing the resource label, its own network address, and the resource parameters. Advertisements are forwarded (using the routing layer) toward the label (that is, to nodes whose labels have increasingly long common prefixes with the resource label), and the location is added to the map of each node along the way. The location is stored at every hop, rather than just the last hop, for two reasons, the first of which is redundancy: As retransmissions of the advertisement take different paths toward the same point, the location is deposited at several nodes in the vicinity of the target label (abstract label-space vicinity, not network vicinity). Therefore, when a `query` is sent toward the same target label, there are several nodes that it is likely to pass through that are likely to have the requested information. The second reason to store the location at every hop is to prevent implosion: If the advertisement reaches a node that already knows "enough" locations for the same label, the advertisement is not forwarded any further. Therefore, if a great number of nodes are all advertising instances of the same resource, the advertisements will not implode and overwhelm the best-matching nodes.

A higher layer seeking a resource sends a `query` message containing the resource label, which is forwarded (using the routing layer) toward the label. Each node along the way consults its location map and sends any matching locations back to the requestor in a `report` message. If "enough" locations are found at one node, the forwarding is halted there, so that a resource stored at a many nodes for load-sharing of resource retrieval will have similar load-sharing of resource lookup. On the other hand, if the best-matching node knows no locations, it sends back an empty `report` as a negative acknowledgement. The sender of a `query` continually sends `query` messages, each of which takes a different path and potentially returns different results, for as long as it wants more locations to choose from. A higher layer can retrieve the resource from any of the locations, or from several in parallel.

Whereas queries are forwarded exactly as described in Section 6.1.2, advertisements are delayed at each hop and batched together—each `post` message can carry many unrelated advertisements. This is done to reduce traffic overhead, and is possible because advertisements, unlike queries, are in no real hurry to reach their destination. Whenever a `post` message arrives, it is split into separate advertisements, each of which needs to be forwarded using a potentially different row of the routing table. Each advertisement is held in a buffer associated with a row of the routing table. Whenever a buffer becomes full enough or its oldest advertisement becomes old enough, the buffer is flushed to form a new `post` message to be forwarded one hop.

The rendezvous provided by the routing layer is useful not only for allowing seekers to find advertisers, but also for allowing advertisers to find each other, to propagate version updates using the timestamps and out-of-date flags. As each node adds locations to its location map, it detects whenever a location's out-of-date flag is unset but should be set (because another location for the same label has a newer timestamp). Whenever this happens, the node sends a `report` to the out-of-date location containing the locations of the newer version(s), and sets the out-of-date flag so that the same notification won't keep getting sent again and again as more advertisements arrive. The node holding the out-of-date resource can then take appropriate action at a higher layer, such as deleting the resource and/or retrieving the newer version.

Unlike `post` and `query` messages, which are sent toward a label, `report` messages are sent to a network address, and could therefore be sent in one hop (one application-level hop). They are, however, always sent in two hops, by way of a random intermediate node,

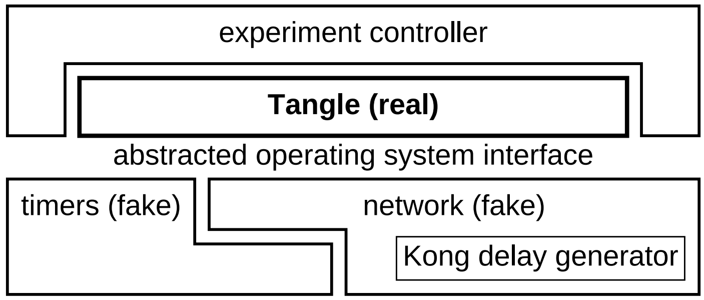

Figure 6.3: Experimental framework.

so that it is not extremely easy to discover the network addresses of nodes matching a given label, a capability that would be useful only to attackers.

6.2 Evaluation

Whereas the evaluations of Search Party (Chapter 4) and Rumor Mill (Chapter 5) included much analysis, corroborated by small-scale experiments based on simulated implementations in Tcl, the evaluation of Tangle shifts away from analysis toward large-scale experiments based on efficient real implementations in C (still running in a simulated network environment). This section describes the framework built to run the experiments, and the setup and parameters of the experiments themselves. The next section will present the results.

6.2.1 Experimental framework

Figure 6.3 shows a diagram of the experimental framework. At the heart is a real implementation of the Tangle protocol, written in C, including both the routing layer and the hash table layer, although only the former was used in the experiments. Rather than being written on top of real timer and network interfaces (like `select()` and the socket interface), the Tangle component is written on top of a simpler more abstract event-driven interface that could be glued onto real operating system interfaces. In the experimental framework, the abstract interface instead provides access to simulated timers and a simulated network, so that all the nodes run on a single machine in a single thread. Above all of this is an experiment controller that instantiates the other components, directs their behavior according to the parameters of the experiment, and collects the results.

The Search Party simulation needed to simulate the network topology (the routers and links) because it used network-layer forwarding defined in terms of the topology. Tangle, on the other hand, is an application-layer protocol that is much less involved with the network topology. The communication in Tangle uses very low data rates, so there is no need for the simulation to model congestion at bottleneck links. Furthermore, the communication in Tangle is intermittent and connectionless. Two packets are never sent back-to-back to the same destination, so there is very little to be gained by modeling the router queues where packets can get stuck behind other packets. In short, there was no need to model routers and links at all; the only aspect of the simulated network that needed to be modeled was the end-to-end delay. That need spawned a side project described in the next section.

The combination of an efficient protocol implementation in C and a very simple network simulator also written in C allowed the Tangle experiments to scale much larger than the Search Party and Rumor Mill experiments, which had simulated the full topology and were written in a scripting language (Tcl). The typical Tangle experiment with 32768 nodes ran at a rate of about 40% real-time on a single 864 MHz Pentium III, using about 85% of its 512 MB of memory.

The source code for the framework is available on the Tangle web site [8].

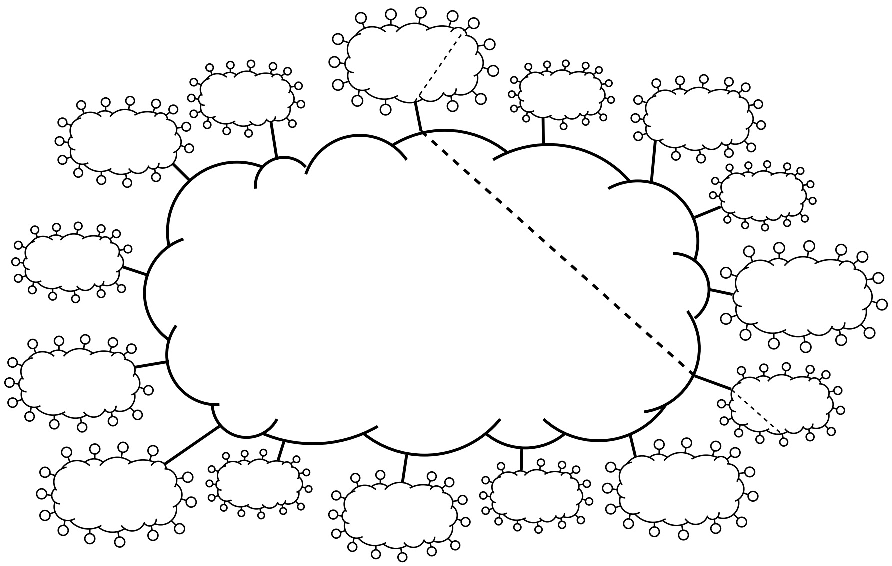

Figure 6.4: Kong: One level of recursion can turn 15 endpoints into 210.

6.2.2 Kong

The primary function of the framework's simulated network component is to generate end-to-end delays. No widely accepted analytic model of realistic network delays was to be found, but we did find a tool called **king** [18] that could measure round-trip times between two remote points in the Internet. From this we developed Kong, a package that finds random points in the Internet, uses **king**'s technique to measure the pairwise round-trip times between them, constructs a table of data, and from that table mimics (or "apes") those round-trip times inside a simulation run. We built Kong as part of the Tangle experimentation framework, but it be reused in unrelated simulations. The source code and data are available on the Kong web site [7].

The technique that Kong inherited from **king** uses recursive domain name queries to measure the round-trip time between any pair of recursive domain name servers. There is no need to install special instrumentation on machines scattered throughout the Internet, because recursive domain name servers are already widely and densely scattered throughout the Internet. The round-trip time between two servers X and Y can be estimated as the response time of a recursive query to Y via X minus the response time of a direct query to X. For greater accuracy and precision, each response time is actually the minimum over several queries.

Kong finds a number of random recursive domain name servers, then measures the round-trip time between every pair so that it can be mimicked later. The round-trip time between two points is not a constant, but a random variable (furthermore, it is a time-dependent random variable, but because communication in Tangle is connectionless and intermittent, it would not have been worth the trouble to try to capture the time dependence). To estimate the distribution of this random variable, Kong takes several independent measurements of the same pair of servers at widely separated times. These measurements are sorted, smoothed, subsampled, and stored in a table. Later, during a simulation run, an approximation of the original random variable is regenerated from a uniform random variable by cubic interpolation between the sample points.

Near the end of March 2003 Kong found 441 recursive servers. Over the next two weeks it performed about fifteen million domain name queries while attempting to gather 18 round-trip time measurements between every pair. Because some of the servers went down during the collection, only 386 servers were included in the final all-pairs data set. After subsampling, the table contained 891,660 floating-point numbers.

The number 386 reveals a limitation of this approach. With only 386 endpoints represented in the table, how can the simulation scale to thousands of nodes? The problem is that pairwise measurements between n servers require $O(n^2)$ time to collect and $O(n^2)$ space to store, so n cannot be very large. If each node in the simulation corresponds to a server, network delay component becomes the limiting factor in the scalability of the entire simulation. To work around this problem, Kong introduces one level of recursion when generating delays. Each of the n servers is viewed as a gateway to a local area that looks like a scaled-down copy of the wide area, as depicted in Figure 6.4 for $n = 15$. Each of the n local areas contains $n - 1$ endpoints, and the round-trip time between two endpoints in different local areas is the sum of three independently generated round-trip times (one to cross the wide area and two to cross the local areas). This workaround allows the simulation of $N \doteq n^2$ nodes using only $O(N)$ space for the table and $O(N)$ time to build it. The scale factor for each local area is chosen to make the gateway equidistant between its nearest neighbor in the wide cloud and its farthest neighbor in the local cloud (the median round-trip time is used as the distance metric). This workaround is a distortion—the real Internet is not organized this way—but the distortion is small, because the scale factors are small (the median is 0.01 and the 95^{th} percentile is 0.04). The total round-trip time is dominated by the wide-area portion, which is based directly on measurements.

To generate one-way delays, Kong simply divides the round-trip time by two, because the **king** technique does not detect asymmetric delays.

6.2.3 Experiments

The experiments used only the routing layer, not the hash table layer, because it is the routing layer that performs forwarding, and it is the routing layer whose fault tolerance is directly affected by the randomization of that forwarding. Improving the fault tolerance of the routing layer can benefit whatever is built upon it, be it a hash table or some other application of the rendezvous service provided by the `send()` operation.

The experiments measure the success rate and response time of `send()` operations in the presence of various injected faults. Each `send()` is performed by a random node to a random target, which is the label of another random node. The node performing the `send()` retransmits the message quasi-periodically at a rate of two per second until an acknowledgement is received from the target node, or until 20 seconds have elapsed, which is defined to be a failure. Each message transmitted by the original sender is relayed through intermediate nodes, which do *not* do any retransmission—they simply forward the message and forget it. The response time of a successful `send()` is the total time from the very first transmission until the receipt of the acknowledgement.

Each `send()` uses one of five strategies for setting the forwarding hints (see Section 6.1.2 in the messages:

uniform All instances of the message contain the "uniform" hint. Each message is forwarded along its own unbiased random path toward the target.

skewed (hops) All instances of the message contain the "reduce hop count" hint. Each message is forwarded along its own random path toward the target, biased toward paths containing fewer hops.

skewed (time) All instances of the message contain the "reduce total time" hint. Each message is forwarded along its own random path toward the target, biased toward faster paths.

deterministic All instances of the message contain the "minimize total time" hint. Each message is forwarded along the same deterministic path toward the target, the one estimated to be the fastest known path.

hybrid The first instance of the message contain the "minimize total time" hint. The next five instances contain the "reduce total time" hint. All remaining instances alternate between the "reduce hop count" and "uniform" hints. Each message is forwarded along its own path toward the target, with later messages using paths less biased toward speed.

The hybrid strategy (or one like it) is what would be used in practice. The deterministic strategy serves as a baseline to show how much failure is being masked by the randomization. The other strategies can expose the strengths and weaknesses of the three distributions for random next-hop selection.

Except where noted, each simulation run involves 32,768 nodes, assigned to a random subset of the 148,610 endpoints of the simulated network. At the start of each run the nodes are as weakly connected as possible: each node knows at most one peer, forming a linear chain, which is uncorrelated with the assignment of network endpoints. Near the beginning of the run, the `send()` operations all fail, because the routing tables are nearly empty and the messages hit dead ends before reaching their targets. As the simulation runs, the peer discovery process gradually fills the routing tables, and the failure rate drops. The system settles into an equilibrium after about 700 to 1100 seconds (depending on the faults that are injected), but the system is warmed up for 4000 seconds, just to be safe. Each run is repeated 20 times, and the data are aggregated.

Most of the experiments inject *steady-state faults*—ongoing faulty conditions. The steady-state faults injected in these experiments are packet losses, route losses, software bugs, and membership churn, described in more detail below. There is also a steady-state experiment that injects no faults at all. One pair of experiments injects *dynamic faults*—sudden jolts to the system. In one the number of nodes is suddenly halved; in the other it is doubled.

The faults injected in the experiments are designed to cover a diverse range of possible faults with enough severity to test the limits of the protocol's tolerance; they do not attempt to closely model the mechanisms and frequencies of real-world faults. The hope is that randomized forwarding is such a generic tactic that it provides resilience to virtually any kind of fault. Of course we cannot test it against all possible faults, but we can try a broad range of faults and see if they are all handled well.

The parameters of the individual experiments (see Section 6.3 for the results) are described below. The steady-state experiments share the following parameters: In each of the 20 runs of each experiment, after the system has warmed up for 4000 seconds, the routing tables are frozen and 20,000 random `send()` operations are performed for each of the five strategies. Freezing the routing tables allows for a clean comparison between the deterministic and randomized strategies; if the routing tables were allowed to continue their random shuffling between retransmissions, then even the "deterministic" `send()` operations would be slightly randomized.

No faults

In this experiment, no faults are injected; every packet reaches its destination, and every node is present from the beginning and performs the protocol correctly. The purpose is to test whether randomized forwarding incurs any cost when everything is working perfectly.

Packet losses

These three experiments inject packet losses at various points. In the first experiment, it is the network that is lossy: every packet has a 10% chance of being dropped. In the second experiment, the losses are concentrated at receivers: 20% of the nodes are lossy receivers, where each arriving packet has a 50% chance of being dropped. In the third experiment, the losses are concentrated at senders: 20% of the nodes are lossy senders, where each outgoing packet has a 50% chance of being dropped.

Route losses

These two experiments inject route losses. Of the N^2 end-to-end network-layer routes connecting the N nodes, 19% are broken, so that packets trying to traverse them are lost. In the first experiment, the broken routes cover all the sources, but are concentrated at 20% of the destinations, which are each invisible to (unreachable by) 95% of their peers. In the second experiment, the broken routes cover all the destinations, but are concentrated at 20% of the sources, which are each blind to (unable to reach) 95% of their peers.

These experiments omit `send()` operations with invisible senders, invisible targets, or blind targets, because such operations would usually fail even if the sender already knew the address of the target and sent the message directly there. Also, the initial who-knows-whom graph is not a linear chain of all nodes; instead the normal nodes form a linear chain, and each of the invisible/blind nodes knows a unique normal node. If the invisible/blind nodes were essential links in the initial chain, there would be too many opportunities for the population to become partitioned very early in the run.

Software bugs

This experiment injects a software bug into 10% of the nodes, causing them to silently fail to utilize their routing tables to forward messages, although they continue to perform the rest of the protocol (constructing and maintaining the routing tables) perfectly.

Membership churn

This experiment injects membership churn; that is, a population of transient nodes, each of which joins the system, stays for a short time (about 500 seconds, just long enough to become well-connected) and then leaves system, never to return. The rate of arrivals and departures is just enough to maintain an average transient population of 24,576, which outnumbers the permanent population of 8192 by a ratio of 3:1. The total population at any moment is about 32768, as usual. Every `send()` operation is from a permanent node to the label of another permanent node, so that the experiment measures not the performance of the transient population but rather its negative impact on the permanent population.

Membership burst

Unlike the previous experiments, the membership burst experiments inject dynamic faults. The system runs without faults for 4000 seconds, then it is jolted, and continues to run. Unlike the steady-state fault experiments, the routing tables are never frozen; random `send()` operations are performed over time throughout the run, both before and after the jolt.

In both of these two experiments, half of the 32768 nodes are permanent; the other half are used to perturb the system. In the first experiment, the non-permanent nodes are present from the beginning and leave en masse. In the second experiment, the non-permanent nodes are not present at the beginning; later they join en masse. As in the membership churn experiment, every `send()` operation is from a permanent node to the label of another permanent node.

6.2.4 Scaling

This experiment is not related to the investigation of randomized versus deterministic forwarding; it tests the scalability of Tangle's routing layer. No faults are injected. The number of nodes is varied in powers of two, and the steady-state mean response time and the settling time are measured. The steady-state mean response time is defined as the mean response time of hybrid `send()` operations during the last 1000 seconds of a 4000 second run, and the settling time is defined as the elapsed time until the filtered (smoothed)

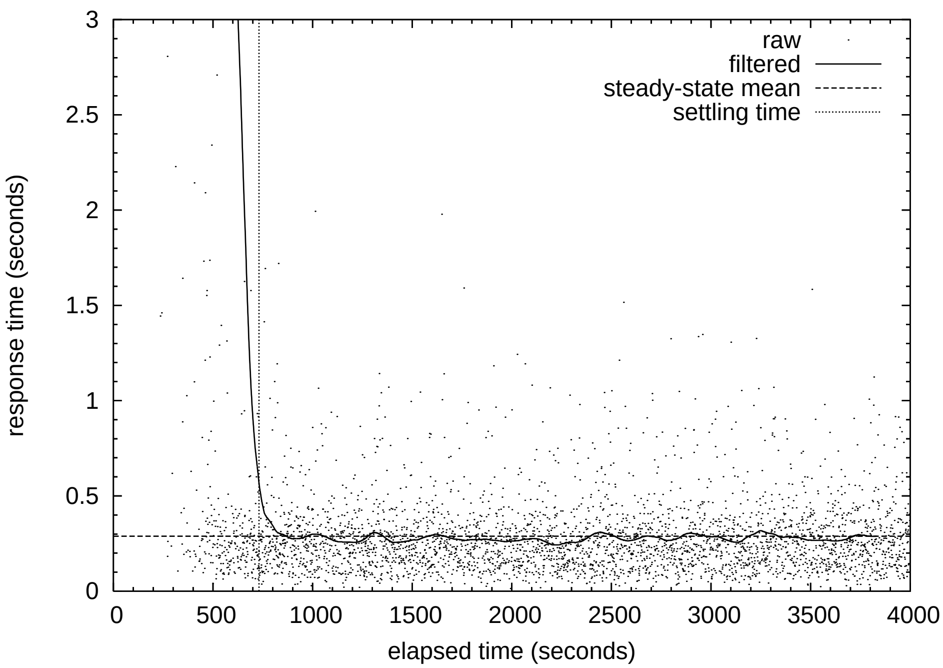

Figure 6.5: Illustration of the terms *steady-state mean response time* and *settling time* for the scaling experiment.

response time falls to within a factor of two of the steady-state mean (see Figure 6.5). The smoothing filter is a triangular filter with a radius of 100 samples.

6.3 Results

Here are presented the results of the experiments described in the previous section. The fault-injection results show that deterministic forwarding yields a mix of very good and very bad performance, depending on whether faults lie along the chosen path; pure randomized forwarding greatly diminishes the bad performance at the cost of some degradation of the good performance; and hybrid forwarding delivers the benefits of randomized forwarding with little or no degradation. The scaling results show that the protocol scales to large numbers of nodes, with costs growing roughly as $\log N$.

The first eight pairs of graphs, corresponding to the steady-state fault experiments, show the distribution of `send()` response times in the presence of various injected faults. Within each pair of graphs, both graphs show the same five curves, one for each forwarding strategy. The top graph of the pair shows a wide view, and the bottom graph zooms in on the rightmost portion of the curves. For the wide view, each curve originally contained 400,000 points (20,000 points from each of 20 runs, all sorted to form a single cumulative distribution function), but such fine detail is not visible and would only make the document file larger and slower to download or print; therefore each curve was subsampled to contain 1001 points. The narrow views were likewise subsampled to 1001 points, except for the ones so narrow that they already contained fewer than 1001 points.

6.3.1 No faults

Figure 6.3.1 shows the distribution of response times in the absence of faults. The four pure (non-hybrid) strategies show, not surprisingly, that the response time tends to

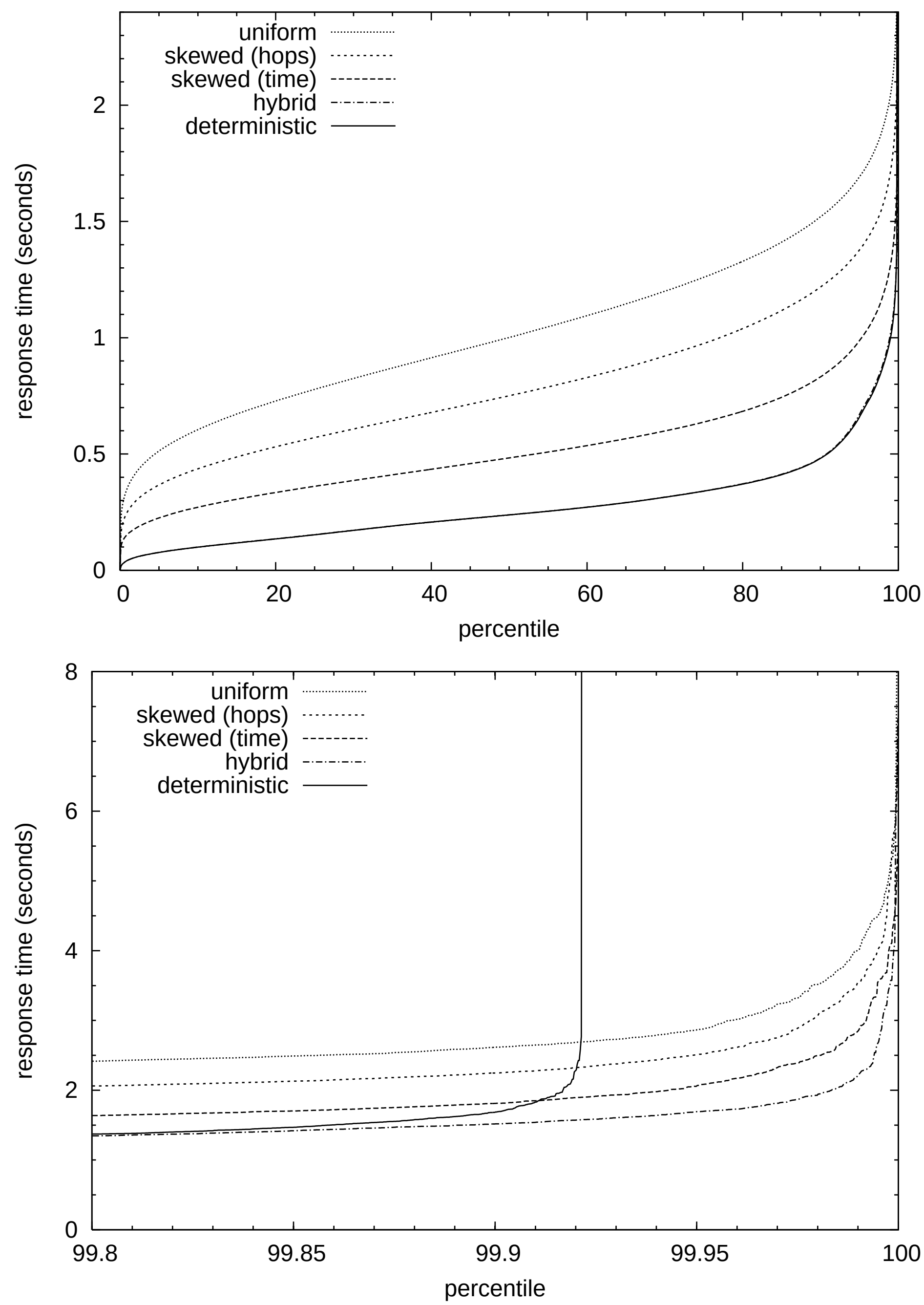

Figure 6.6: Distribution of response times with no faults.

improve (shrink) as the choice of next-hop becomes more skewed in favor of speed (where deterministically choosing the fastest peer is the most extreme form of skewing).

About 1/1000 of the deterministic `send()` operations fail to reach their targets, because the routing tables are never all 100% complete—they are always in a state of flux as the soft-state peer discovery protocol continually adds new entries and discards old entries. This is not considered a problem in Tangle because retransmitting packets along different paths takes care of it, as the non-deterministic curves show. Of course, retransmitting a packet along the same path is ineffective against an incomplete routing table, which is why the deterministic curve never reaches 100%.

The hybrid strategy delivers both the speed of the deterministic strategy and the low failure rate of the pure randomized strategies.

6.3.2 Packet losses

Figures 6.7, 6.8, and 6.9 show the distribution of response times in the presence of packet losses throughout the network, concentrated at some receivers, and concentrated at some senders, respectively. The distributions of the four pure (non-hybrid) strategies in all three packet-loss experiments resemble their counterparts from the no-faults experiment in shape and relative order, but the response times and the failure rates are greater. The failure rate for deterministic forwarding is about 1/500 for unconcentrated packet losses, and about 1/100 for the concentrated packet losses. Non-deterministic forwarding reduces the failure rate by more than an order of magnitude.

When a message is stopped by a packet loss rather than an incomplete routing table, retransmitting along the same path can be helpful, and therefore the hybrid strategy pays a price for abandoning the fastest path after the first transmission: its curve is

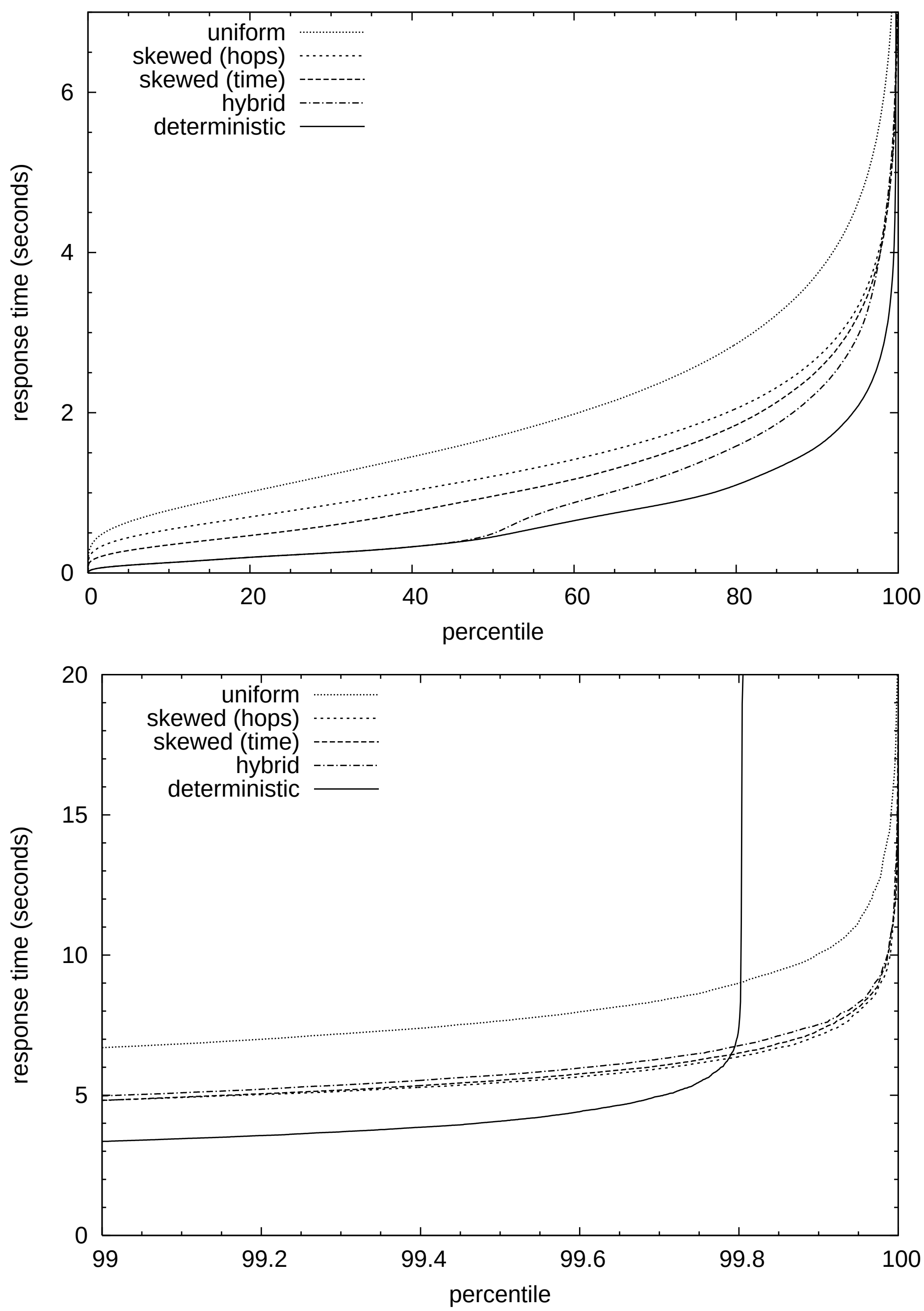

Figure 6.7: Distribution of response times with packet losses in the network. 10% of all packets are dropped.

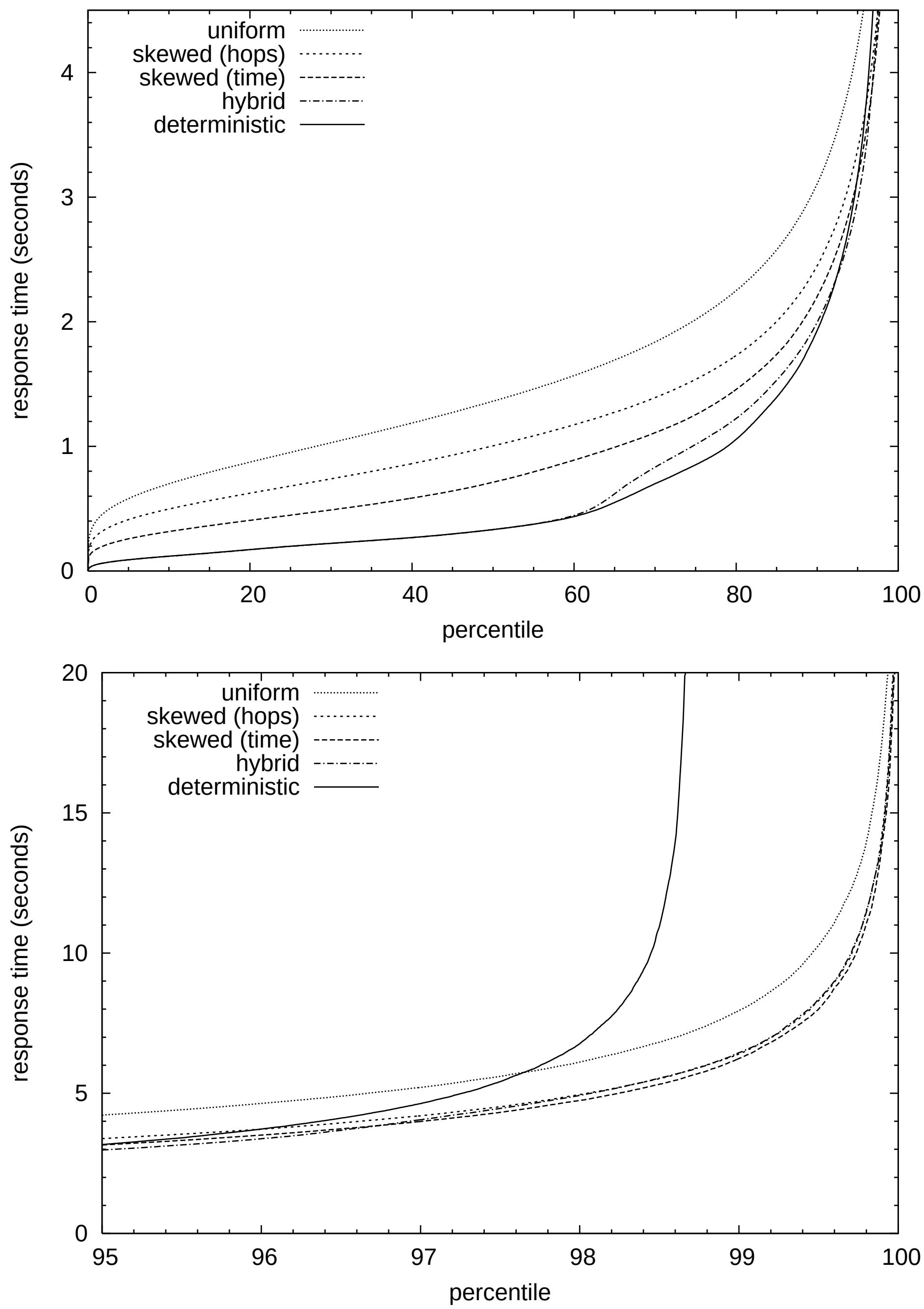

Figure 6.8: Distribution of response times with packet losses at receivers. 20% of all nodes drop 50% of their incoming packets.

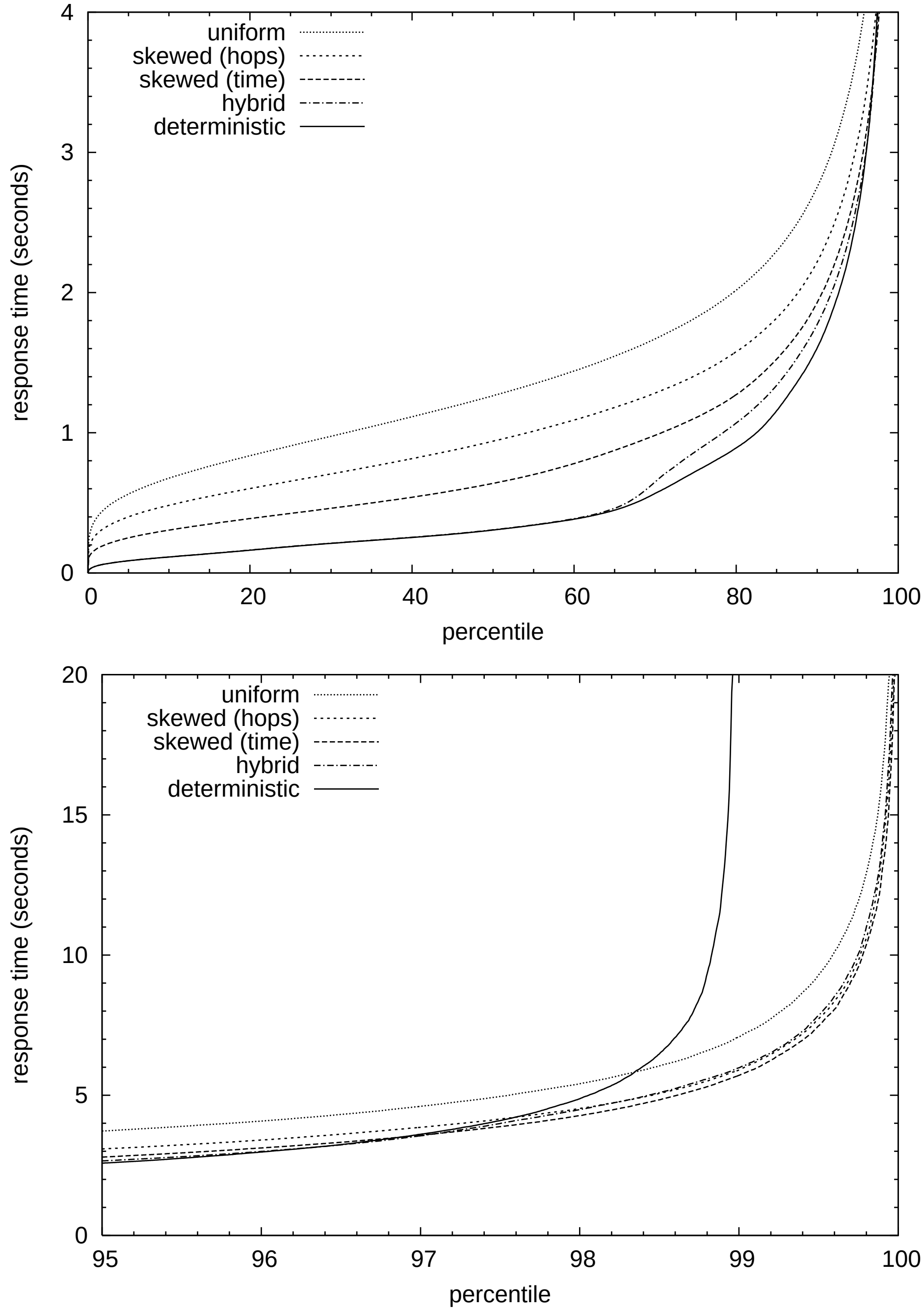

Figure 6.9: Distribution of response times with packet losses at senders. 20% of all nodes drop 50% of their outgoing packets.

sometimes higher than the deterministic curve, up to about 50% higher for unconcentrated losses, and up to about 30% higher for concentrated losses.

But incomplete routing tables are also present, and more common than they were in the no-faults experiment, because the packet losses interfere with the peer discovery process. Incomplete routing tables are presumably responsible for most of the failures of the deterministic `send()` operations.

6.3.3 Route losses

Figures 6.10 and 6.11 show the distribution of response times in the presence of route losses concentrated at destinations and sources, respectively. These faults mildly increase the response time for all forwarding strategies, and greatly increase the failure rate for deterministic `send()` operations, to about 1/30 in the first experiment and about 1/10 in the second. Non-deterministic forwarding reduces the failure rate by about five orders of magnitude or more (no failures occurred among 1.6 million non-deterministic `send()` operations in each experiment).

For this fault there is no performance penalty for hybrid forwarding compared to deterministic forwarding.

6.3.4 Software bugs

Figure 6.12 shows the distribution of response times in the presence of software bugs that block forwarding but leave peer discovery unaffected. This fault has effects similar to those of route losses, but to a greater extent: it increases the response time for all forwarding strategies, and greatly increases the failure rate for deterministic `send()` operations, to about 1/3. Compared to deterministic forwarding, hybrid forwarding incurs no performance penalty and reduces the failure rate by more than four orders of magnitude.

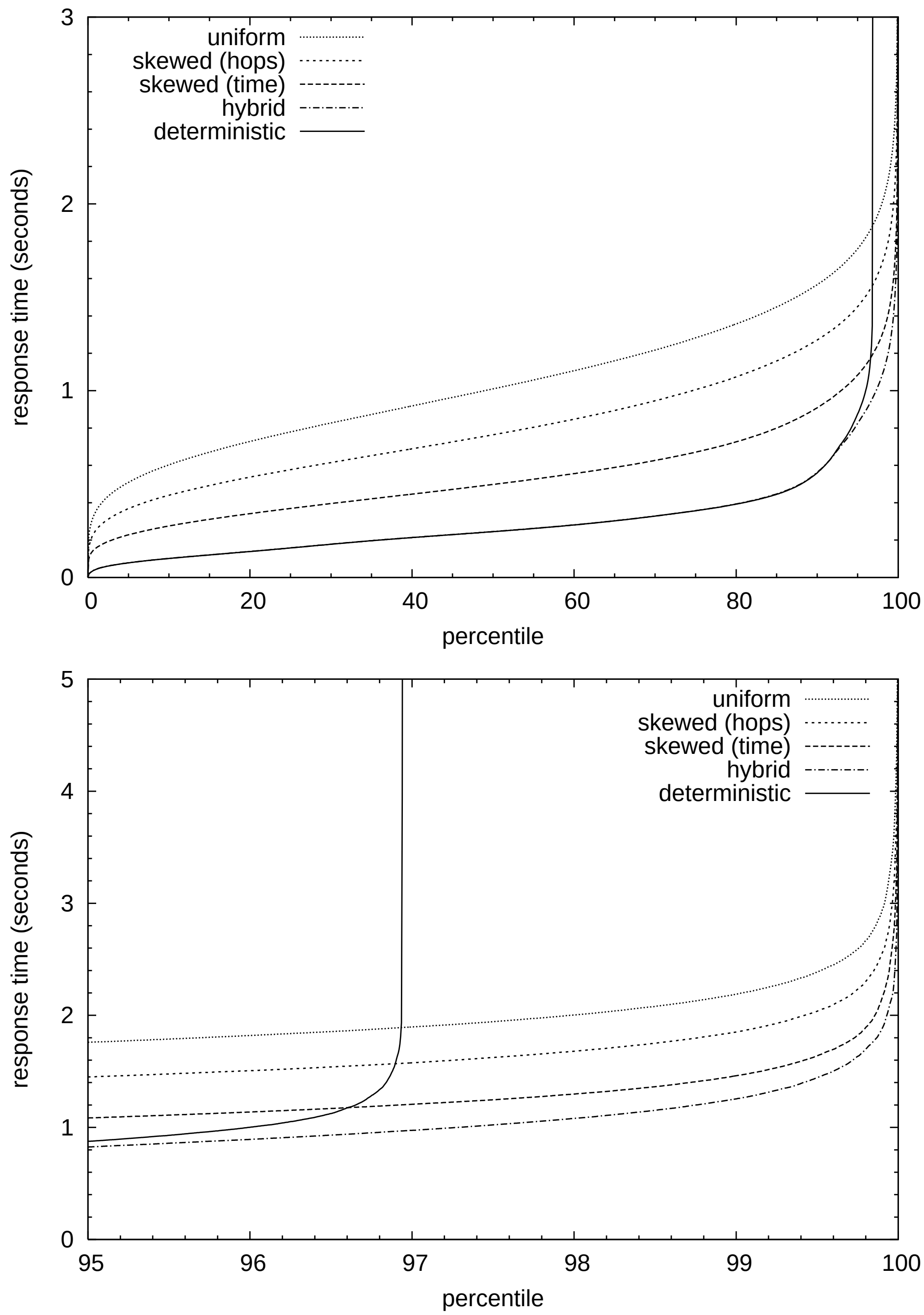

Figure 6.10: Distribution of response times with route losses concentrated at destinations. 20% of all nodes are invisible to (unreachable by) 95% of their peers.

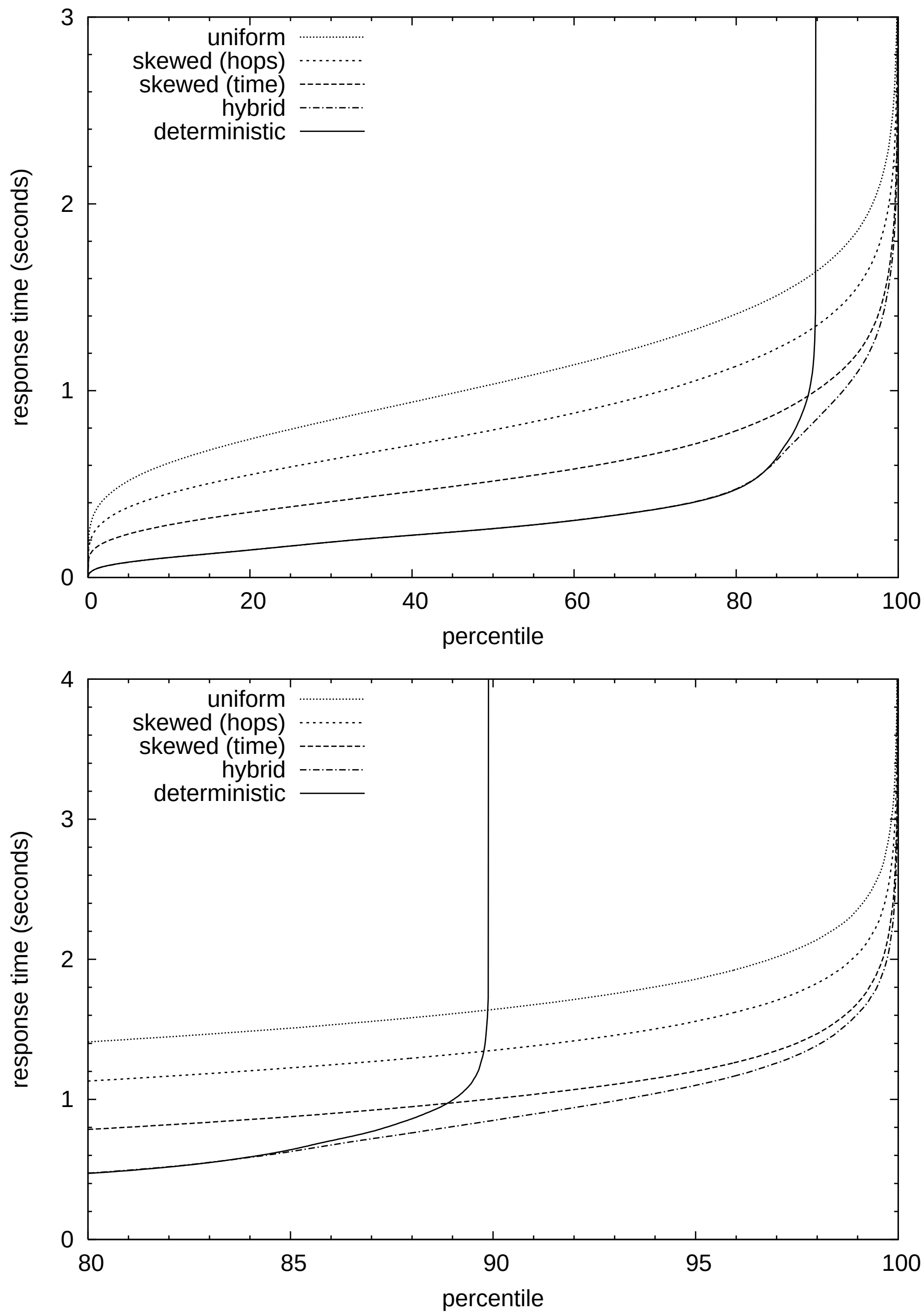

Figure 6.11: Distribution of response times with route losses concentrated at sources. 20% of all nodes are blind to (unable to reach) 95% of their peers.

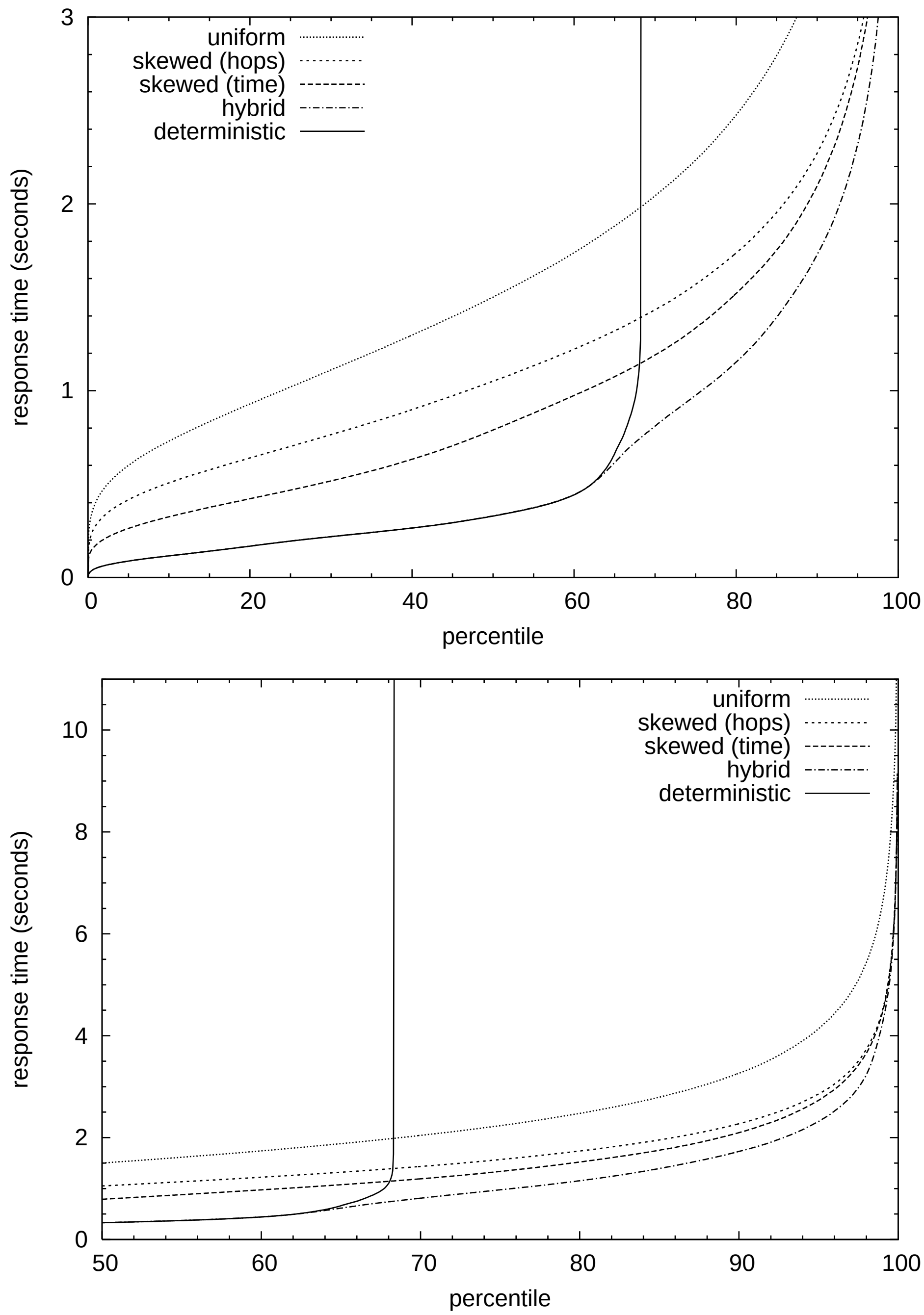

Figure 6.12: Distribution of response times with software bugs. 10% of all nodes have a bug that prevents forwarding but does not affect peer discovery.

6.3.5 Membership churn

Figure 6.13 shows the distribution of response times in the presence of membership churn (a large number of nodes continually joining and leaving the system). This fault has effects similar to those of route losses and software bugs, but to an even greater extent: it greatly increases the response time for all forwarding strategies, and greatly increases the failure rate for deterministic `send()` operations, to more than 1/2. Compared to deterministic forwarding, hybrid forwarding incurs no performance penalty and reduces the failure rate by more than two orders of magnitude.

6.3.6 Membership burst

Figure 6.14 shows the success rate before and after a membership burst (a large number of nodes suddenly joining or leaving the system). Unlike the previous graphs, these are not distributions (with percentile on the horizontal axis) but evolutions (with elapsed time on the horizontal axis), and the polarity of the vertical axis is reversed (a higher success rate is better, whereas a higher response time was worse). The success rate at time t (a multiple of 10 seconds) was computed over the interval $[t-10, t+10)$.

The mass leave is very disruptive for all forwarding strategies, with the deterministic strategy suffering the most. The robustness of the non-deterministic strategies increases with their degree of preference for shorter paths (because shorter paths have fewer opportunities for failure), so the hybrid strategy is the most robust. There is a 30–40% spread between the deterministic and hybrid strategies during most of the recovery.

The mass join is less disruptive than the mass leave, causing a failure rate of up to about 20% for the deterministic strategy, and much less for the non-deterministic strategies. The peak failure rate for the hybrid strategy is less than 0.2%.

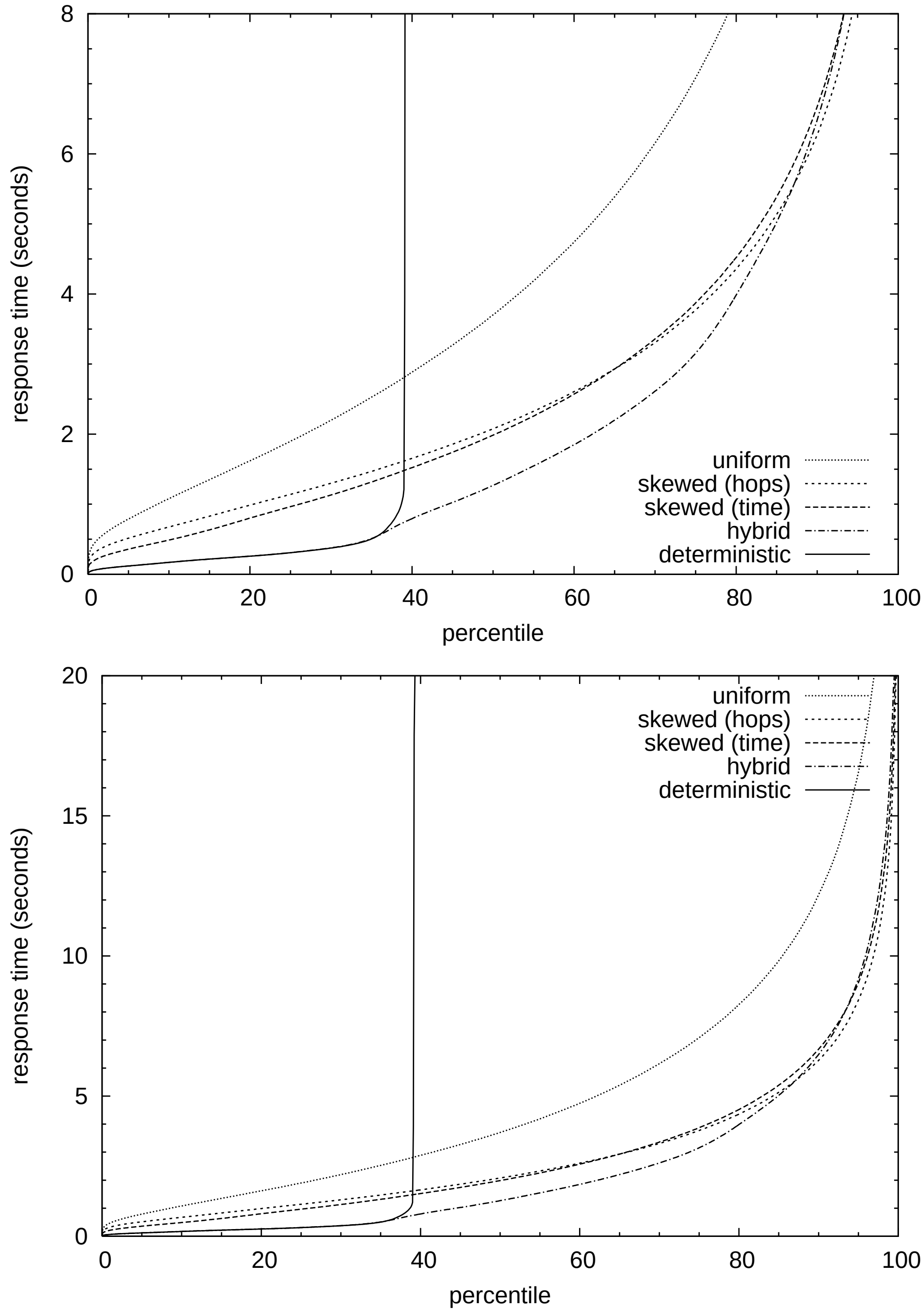

Figure 6.13: Distribution of response times with membership churn. A steady stream of transient nodes join, stay for about 500 seconds, and leave. 75% of nodes present are transient.

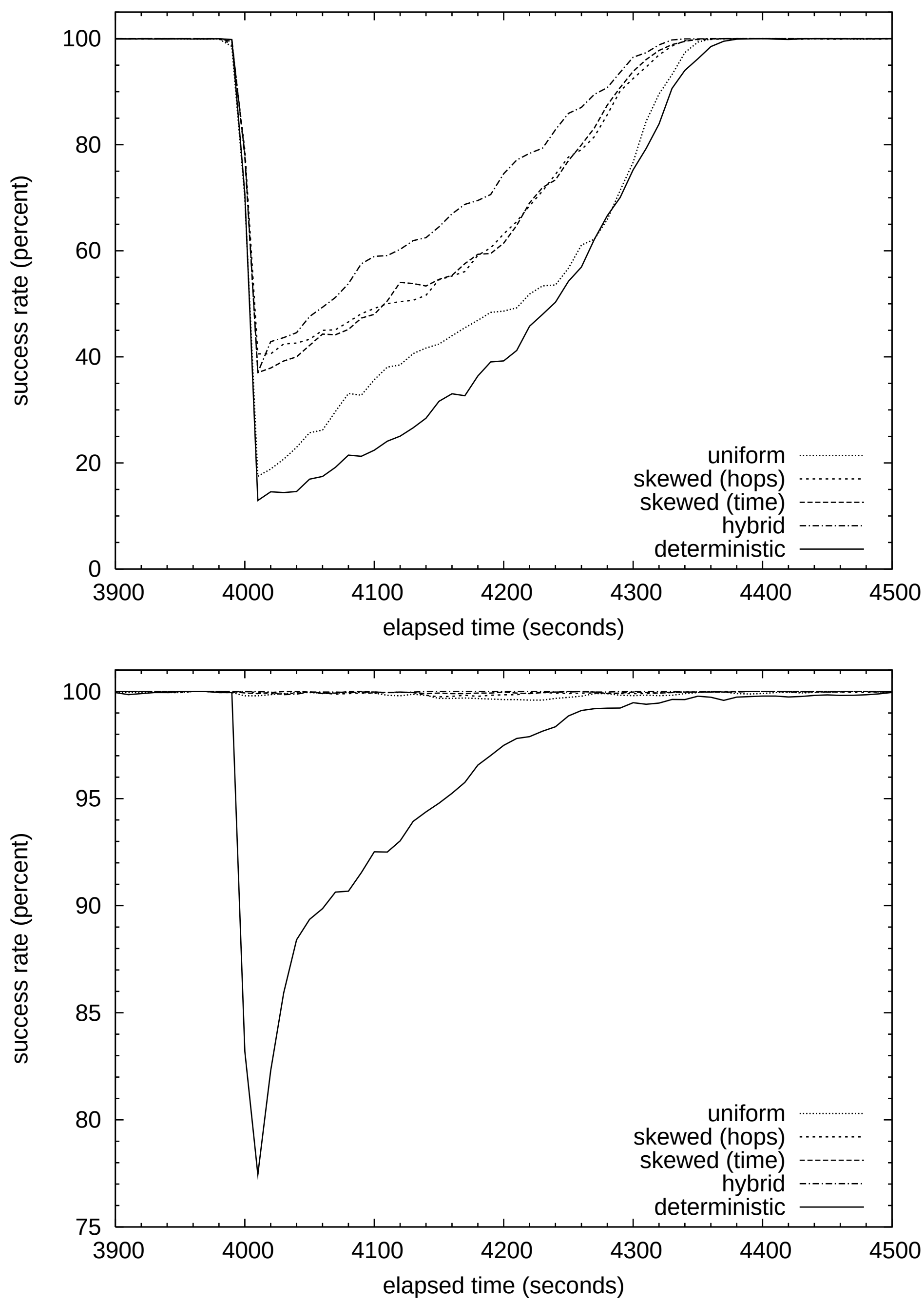

Figure 6.14: Success rate before and after a mass leave (top) or a mass join (bottom). 50% of the nodes leave/join at time 4000.

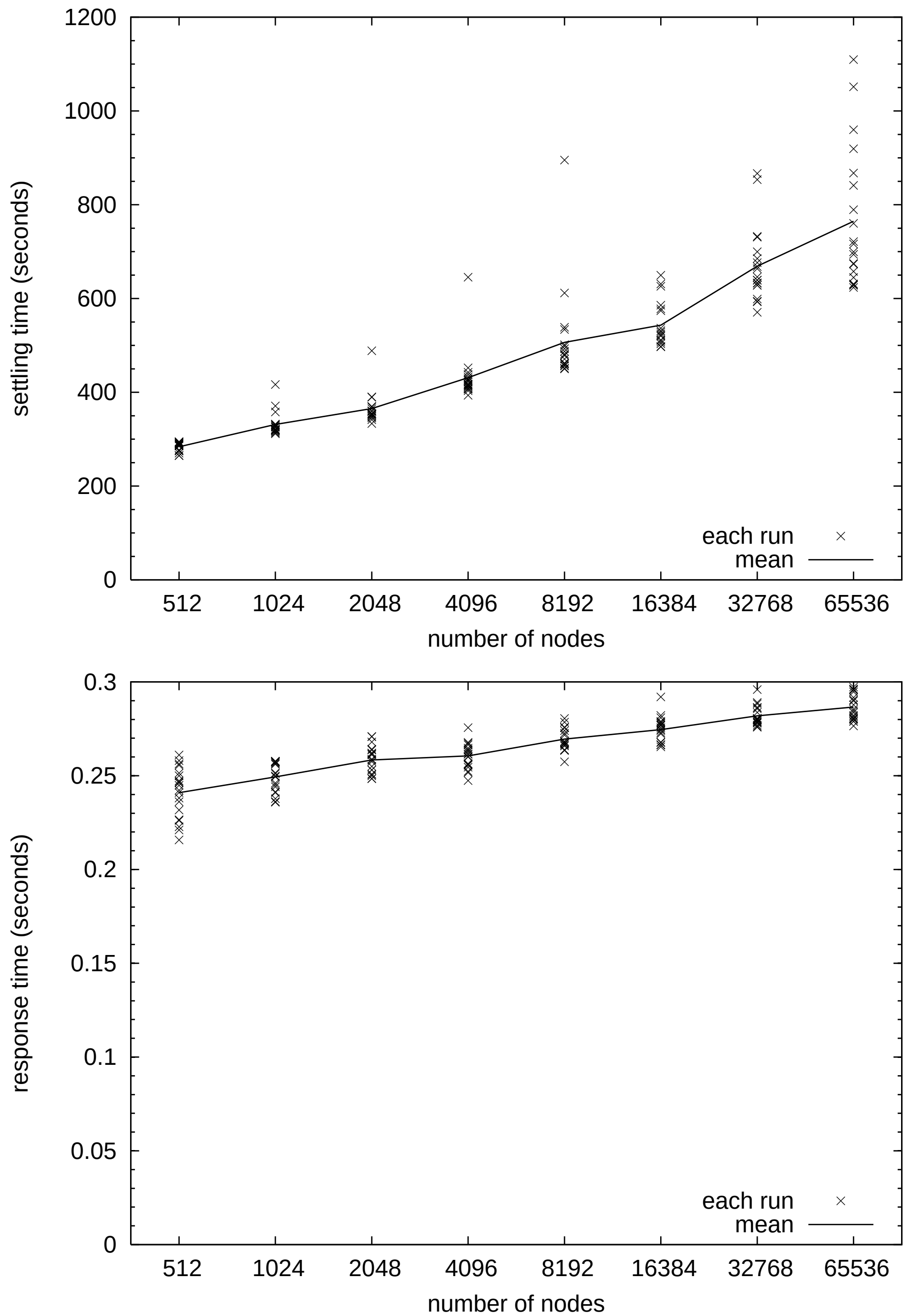

Figure 6.15: Settling time (top) and steady-state mean response time (bottom) versus number of nodes.

6.3.7 Scaling

Figure 6.15 shows how the population size N affects the settling time (the time to construct the routing tables from scratch) and the steady-state mean response time (the average time to complete a `send()` operation after the tables have been constructed). See Section 6.2.4 for more precise definitions of the metrics. The horizontal axis (population size) is on a log scale. The settling time increases at a scalable rate, slightly faster than $\log N$. The response time grows slowly with $\log N$, about 7 ms for each doubling of N.

6.4 Summary

The term *distributed hash table* originally referred literally to a hash table spread across many machines, but has since come to refer primarily to the routing layer that provides the essential *rendezvous* service, which allows two peers to discover each other using a shared key. Distributed hash table protocols are varied, but they all assign uniformly distributed *labels* to each node, they all forward messages toward the node that best matches a target label, and they all derive their scalability from the *small-world phenomenon*—the ability to find short paths in graphs having a suitable mix of local and long-range edges.

We have designed and built Tangle, a distributed hash table based on the use of randomized forwarding to provide redundant paths, so that the rendezvous service can tolerate faults that interfere with forwarding and with the construction of routing tables. The peer discovery sublayer uses a new multi-level gossip-style protocol to construct routing tables in which each next hop is not the single best next hop but rather a random sample of all suitable next hops. The forwarding sublayer forwards messages hop-by-hop, selecting each next hop from among the candidates using a method influenced by a forwarding hint contained in the message. The original message sender retransmits the message (which should be idempotent), possibly varying the hint from one instance to the next, until it determines success or failure (perhaps using end-to-end feedback and timeouts).

We have developed Kong, a tool that measures real end-to-end network delays in advance and uses the data to mimic those delays in a large simulated network. It is applicable to simulations that need to model end-to-end delays but not the network topology and individual router queues.

We have built a framework for running tens of thousands of instances of the Tangle implementation atop a simulated network (using Kong as a component), injecting faults and measuring their effects with and without randomized forwarding. A wide range of injected faults, including packet losses, route losses, software bugs, and membership changes, all yield similar results: the use of randomized forwarding reduces the failure rate by orders of magnitude at little or no cost in the common-case performance.

Chapter 7

Future Work and Conclusion

Section 7.1 describes some directions for further work on Search Party, Rumor Mill, and Tangle. Section 7.2 summarizes and concludes the dissertation.

7.1 Future Work

The simulations of Search Party and Rumor Mill used small groups of at most 64 nodes, with synthetic topologies. To experiment with realistic groups, real topologies must be obtained, and **ns** must be used more efficiently, or another simulator used, or experiments on real networks conducted.

We have focused on source-rooted multicast distribution trees, but in practice IP multicast will also use shared trees, in which the source is not the root of the tree, but is rather a leaf[1]. This is unlikely to matter for Rumor Mill, because it makes no use of the concepts of “up”, “down”, or “subtree”, but it could be a concern for Search Party. If we can arrange for the root of the tree, when it decides to forward a randomcast packet “upward”, to tunnel the packet to the source, then Search Party should still work. For

[1] This is the situation in core-based trees [1]. In PIM sparse-mode [11] the source still appears to be the root, because packets go from the source to the rendezvous point before entering the tree.

all edges except those on the path between the source and root, "toward the source" and "toward the root" are the same, so most recovery behavior will be unaffected. When a loss occurs on the way from the source to the root, there will not be a loss subtree, but rather a *non-loss subtree.* If the non-loss subtree is small compared to the whole tree, a request will soon be tunneled from the root to the source, who will send a response to the entire tree. If the non-loss subtree accounts for much of the tree, the recovery might involve multiple requests entering the non-loss subtree and generating responses that carry the data out to various separate subtrees. In either case, Search Party should work unmodified on shared trees, but simulations are needed to test this hypothesis.

Alternatively, if routers maintain positional labels as proposed in [29], they can "re-hang" the shared tree to make any source appear to be the root for packets carrying the positional label of the source.

Rumor Mill relies on fewer and simpler changes to the network than Search Party, but could it work with no changes to the network at all? Tangle uses an end-to-end protocol for discovering group members and exchanging packets with random nodes in the group. Perhaps a simplified variant of that protocol could be used as a substitute for trivial randomcast in Rumor Mill. It would not mimic the uniform distributions of trivial randomcast, but might work well enough.

The Tangle protocol has a number of parameters that were fixed during the experiments, including: the maximum size of each unconfirmed peer set (30) and confirmed peer set (20), the number of peers to include in a pong message (2), the rate at which each node sends ping messages (about $\sqrt{\log N}/5$ per second), the confirmed peer age limit (about $100\sqrt{\log N}$ seconds, just enough so that if a confirmed set's bound were lifted, its expected average occupancy would equal the former bound, if all ping messages succeed), the rate at which each node adds a peer to its historic list (12 per hour), the weighting of the coin used to select the historic peer to be removed (1:1), and the retransmission rate for the originating node of a `send()` operation (2 per second). One could investigate the consequences of varying these parameters, the sensitivities and tradeoffs.

Section 6.1.1 contains a conjecture that the Tangle peer discovery protocol causes the peer sets to "quickly become random". This calls for a formalization of what it means to "become random", and an analysis of how long it takes.

The Tangle experiments all used a simulated network, but one could perform future experiments over a true wide-area network, such as PlanetLab [4]. The same Tangle implementation could be used if the glue code were written to tie its abstract network interface to a real network interface (like the socket layer), and if code were written to coordinate the setup and data collection for widely dispersed nodes. The main challenge might be the logistics of gaining access to enough Internet hosts to be able to conduct experiments of an interesting scale.

Another direction for future work is the serverless web application that Tangle was designed to sit under, which was briefly described near the beginning of Section 6.1. Much of the design has been done, but none of the implementation, simply because the higher application layers were not directly relevant to our randomized forwarding research.

One could consider generalizing the rendezvous operation at the heart of all distributed hash tables. Currently two nodes can rendezvous based on a label, but the labels used by the nodes must match exactly. A much harder problem to solve in a scalable decentralized way would be to allow nodes to rendezvous based on some sort of pattern matching, rather than exact matching. This more powerful primitive might enable a scalable decentralized search application.

7.2 Conclusion

Peer-to-peer systems enable scalable reliable services to be built from many small unreliable components, using multi-hop forwarding to allow peers to find each other. To cope with the faulty components, the forwarding service can be designed more toward fault tolerance rather than performance, using randomization. We have examined the use of randomized forwarding at both the network layer and the application layer, in peer-to-peer protocols for two diverse problems: multicast loss recovery, and distributed hash tables.

Recovery from multicast packet losses is challenging because the unknown loss subtree can be much larger than a single member and/or much smaller than the entire group, so the recovery must be localized in order to be scalable. The existing LMS protocol showed that new forwarding services defined in terms of the multicast distribution tree, parentcast and subcast, can be used to solve the local recovery problem, but it suffered from sensitivity to node fault patterns and packet loss patterns, because it organizes the membership into a hierarchy in which the many nodes near the bottom depend on the few nodes near the top, and the former are unnecessarily bothered when packet losses affect only the latter. Chapter 4 presented a new loss recovery protocol, Search Party, that uses a new randomcast service rather than parentcast, thereby overcoming the fault tolerance problems of LMS at the expense of a factor of two in speed and efficiency, as we showed via proofs, analytic approximations, and small-scale simulations (up to 64 peers). Concern about the practicality of the network-layer support needed by both LMS and Search Party led us to develop Rumor Mill, described in Chapter 5, which solves the same problem using a simpler randomcast variant, and no subcast, and the cost of a factor of $O(\log N)$ in speed and request traffic.

Distributed hash table protocols like CAN, Chord, Tapestry, and Pastry show various ways to exploit the small-world phenomenon to allow peers to rendezvous using shared keys and deterministic application-layer forwarding. Chapter 6 introduced a new protocol, Tangle, that uses randomized forwarding based on multiple-next-hop routing tables constructed by a new gossip-style peer discovery protocol. The fault tolerance benefits of the randomized forwarding were examined via large-scale experiments (up to 32,768 peers) running a real implementation of Tangle on top of a simulated network using Kong, a new method for generating realistic network delays based on measurements collected in advance. By measuring the effects of a wide variety of injected faults, we determined that randomized forwarding reduced failure rates by orders of magnitude at little or no cost in performance.

Taken together, this work demonstrates that randomized forwarding is an effective, practical, and general technique for improving the fault tolerance of peer-to-peer protocols, and provides illustrative examples across multiple communication layers and problem domains.

Bibliography

[1] T. Ballardie, P. Francis, and J. Crowcroft. Core based trees (CBT): An architecture for scalable inter-domain multicast routing. In *SIGCOMM'93: Conference proceedings on Communications architectures, protocols and applications*, pages 85–95. ACM Press, September 1993.

[2] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. The World-Wide Web. *Communications of the ACM*, 37(8):76–82, 1994.

[3] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, volume 36 of *ACM SIGOPS Operating Systems Review*, pages 299–314. USENIX Association, ACM Press, December 2002.

[4] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: An overlay testbed for broad-coverage services. *ACM Computer Communication Review*, 33(3):3–12, 2003.

[5] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity*

and Unobservability, Proceedings, volume 2009 of *Lecture Notes in Computer Science*, page 46. Springer, July 2000.

[6] Bram Cohen. Incentives build robustness in bittorrent. In *Proceedings of the First Workshop on the Economics of Peer-to-Peer Systems*, June 2003.

[7] Adam M. Costello. Kong: The network delay ape. http://www.nicemice.net/amc/research/kong/.

[8] Adam M. Costello. Tangle: A distributed hash table based on randomized forwarding. http://www.nicemice.net/amc/research/tangle/.

[9] Adam M. Costello and Steven McCanne. Search party: Using randomcast for reliable multicast with local recovery. In INFOCOM'99 [24], pages 1256–1264. http://www.nicemice.net/amc/research/search-party/.

[10] S. Deering and D. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems (TOCS)*, 8(2):85–110, May 1990.

[11] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C.-G. Liu, and L. Wei. An architecture for wide-area multicast routing. In *SIGCOMM'94: Proceedings of the conference on Communications architectures, protocols and applications*. ACM Press, August 1994.

[12] C. Anthony DellaFera, Mark W. Eichin, Robert S. French, David C. Jedlinksy, John T. Kohl, and William E. Sommerfield. The Zephyr notification service. In *Proceedings of the USENIX Winter Conference*. USENIX Association, December 1988.

[13] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM Press, 1987.

[14] Hans Eriksson. Mbone: The multicast backbone. *Communications of the ACM*, 37(8):54–60, 1994.

[15] Kevin Fall and Sally Floyd. Simulation-based comparisons of Tahoe, Reno and SACK TCP. *ACM Computer Communication Review*, 26(3):5–21, July 1996.

[16] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, December 1997.

[17] Sally Floyd and Van Jacobson. The synchronization of periodic routing messages. *IEEE/ACM Transactions on Networking*, 2(2):122–136, April 1994.

[18] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *Proceedings of the 2nd Usenix / ACM SIGCOMM Internet Measurement Workshop (IMW)*, November 2002.

[19] Mor Harchol-Balter, Tom Leighton, and Daniel Lewin. Resource discovery in distributed networks. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 229–237. ACM Press, April 1999.

[20] Markus Hofmann. A generic concept for large-scale multicast. In *Proceedings of IZS'96: International Zurich Seminar on Digital Communications*, volume 1044 of *Lecture Notes in Computer Science*, pages 95–106. Springer, February 1996.

[21] Hugh W. Holbrook and David R. Cheriton. IP multicast channels: EXPRESS support for large-scale single-source applications. In *SIGCOMM'99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 65–78. ACM Press, August 1999.

[22] ICQ Inc. The ICQ story. http://www.icq.com/info/icqstory.html.

[23] *INFOCOM'96, Fifteenth Annual Joint Conference of the IEEE Computer Societies, Networking the Next Generation, Proceedings*, March 1996.

[24] *INFOCOM'99, Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies, Proceedings*, March 1999.

[25] Phil Karn and Craig Partridge. Improving round-trip time estimates in reliable transport protocols. *ACM Transactions on Computer Systems (TOCS)*, 9(4):364–373, November 1991.

[26] Jon Kleinberg. Small-world phenomena and the dynamics of information. In *Advances in Neural Information Processing Systems 14 (Proceedings of NIPS 2001)*, pages 431–438. MIT Press, December 2001.

[27] Satish Kumar, Pavlin Radoslavov, David Thaler, Cengiz Alaettinoglu, Deborah Estrin, and Mark Handley. The MASC/BGMP architecture for inter-domain multicast routing. In *Proceedings of the ACM SIGCOMM'98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 93–104. ACM Press, August 1998.

[28] LBL/PARC/UCB/ISI/ICIR. Network simulator **ns**-2. http://www.isi.edu/nsnam/ns/.

[29] B. Levine, D. Lavo, and J.J. Garcia-Luna-Aceves. The case for concurrent multicast using shared ack trees. In *Proceedings of the fourth ACM international conference on Multimedia*, pages 365–376. ACM Press, November 1996.

[30] Brian Neil Levine and J.J. Garcia-Luna-Aceves. Internet multicast based on group-relative addressing. http://www.cse.ucsc.edu/research/ccrg/publications/brian.aim98.pdf, November 1998.

[31] Brian Neil Levine, Sanjoy Paul, and J.J. Garcia-Luna-Aceves. Organizing multicast receivers deterministically according to packet-loss correlation. In *Proceedings of the sixth ACM international conference on Multimedia*, pages 201–210. ACM Press, September 1998.

[32] Brian Niel Levine and J.J. Garcia-Luna-Aceves. Improving internet multicast with routing labels. In *Proceedings of the Fifth International Conference on Network Protocols (ICNP'97)*, pages 241–250, October 1997.

[33] Dan Li and David R. Cheriton. OTERS (On-Tree Efficient Recovery using Subcasting): A reliable multicast protocol. In *Proceedings of the Sixth International Conference on Network Protocols (ICNP'98)*, pages 237–245, October 1998.

[34] John C. Lin and Sanjoy Paul. RMTP: A reliable multicast transport protocol. In INFOCOM'96 [23], pages 1414–1425.

[35] A. Mankin, A. Romanow, S. Bradner, and V. Paxson. *RFC 2357: IETF Criteria for Evaluating Reliable Multicast Transport and Application Protocols*, June 1998. http://www.rfc-editor.org/.

[36] Steven McCanne, Van Jacobson, and Martin Vetterli. Receiver-driven layered multicast. In *SIGCOMM'96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 117–130. ACM Press, August 1996.

[37] David L. Mills. Improved algorithms for synchronizing computer network clocks. *IEEE/ACM Transactions on Networking*, 3(3):245–254, June 1995. See also RFC 1305.

[38] Moni Naor and Udi Wieder. Novel architectures for p2p applications: the continuous-discrete approach. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 50–59. ACM Press, June 2003.

[39] Moni Naor and Udi Wieder. A simple fault tolerant distributed hash table. In *Proceedings of Peer-to-Peer Systems II: Second International Workshop, IPTPS 2003*, volume 2735 of *Lecture Notes in Computer Science*, pages 88–97. Springer, February 2003.

[40] Andy Oram, editor. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* O'Reilly, February 2001.

[41] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, 1994.

[42] Christos Papadopoulos, Guru Parulkar, and George Varghese. An error control scheme for large-scale multicast applications. In *INFOCOM'98, Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies, Proceedings*, pages 1188–1196, March 1998.

[43] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 311–320. ACM Press, June 1997.

[44] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In SIGCOMM'01 [54], pages 161–172.

[45] Sylvia Ratnasamy and Steven McCanne. Inference of multicast routing tree topologies and bottleneck bandwidths using end-to-end measurements. In INFOCOM'99 [24], pages 353–360.

[46] RFC Editor. *RFC 793: Transmission Control Protocol*, September 1981. http://www.rfc-editor.org/.

[47] RFC Editor. *RFC 1034: Domain names – concepts and facilities*, August 1982. http://www.rfc-editor.org/.

[48] RFC Editor. *RFC 821: Simple Mail Transfer Protocol*, August 1982. http://www.rfc-editor.org/.

[49] RFC Editor. *RFC 759: File Transfer Protocol*, October 1985. http://www.rfc-editor.org/.

[50] RFC Editor. *RFC 977: Network News Transfer Protocol*, February 1986. http://www.rfc-editor.org/.

[51] RFC Editor. *RFC 1459: Internet Relay Chat Protocol*, May 1993. http://www.rfc-editor.org/.

[52] RFC Editor. *RFC 3920: Extensible Messaging and Presence Protocol (XMPP): Core*, October 2004. http://www.rfc-editor.org/.

[53] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, volume 2218 of *Lecture Notes in Computer Science*, page 329. Springer, November 2001.

[54] *SIGCOMM'01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM Press, August 2001.

[55] Tony Speakman, Jon Crowcroft, Jim Gemmell, Dino Farinacci, Steven Lin, Dan Leshchiner, Michael Luby, Todd L. Montgomery, Luigi Rizzo, Alex Tweedly,

Nidhi Bhaskar, Richard Edmonstone, Rajitha Sumanasekera, and Lorenzo Vicisano. *RFC 3208: PGM Reliable Transport Protocol Specification*, December 2001. http://www.rfc-editor.org/.

[56] Ion Stoica, Rorbert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In SIGCOMM'01 [54], pages 149–160.

[57] Peggy Tang Strait. *A First Course in Probability and Statistics.* Harcourt Brace Javanovich, second edition, 1989.

[58] U.S. Department of Commerce. *Secure Hash Standard, FIPS PUB 180-1*, April 1995.

[59] David Wetherall and Christopher J. Lindblad. Extending Tcl for dynamic object-oriented programming. In *Proceedings of the Third Annual Tcl/Tk Workshop*, pages 173–181. USENIX Association, July 1995.

[60] Wikipedia, the free encyclopedia. Gnutella. http://en.wikipedia.org/wiki/Gnutella.

[61] Wikipedia, the free encyclopedia. Napster. http://en.wikipedia.org/wiki/Napster.

[62] Koichi Yano and Steven McCanne. The breadcrumb forwarding service: A synthesis of PGM and EXPRESS to improve and simplify global IP multicast. *ACM Computer Communication Review*, 30(2):41–49, 2000.

[63] Rajendra Yavatkar, James Griffioen, and Madhu Sudan. A reliable dissemination protocol for interactive collaborative applications. In *Proceedings of the third ACM international conference on Multimedia*, pages 333–344. ACM Press, November 1995.

[64] Ben Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1), January 2004.