THE RELATIONSHIP BETWEEN TOPOLOGY AND PROTOCOL PERFORMANCE: CASE STUDIES

by

Pavlin Ivanov Radoslavov

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(COMPUTER SCIENCE)

December 2001

Dedication

To my parents,

for all of the sacrifices.

Acknowledgments

Now that I am on the finishing line of my life as a graduate student, it is time to sit down, relax, collect my thoughts and think about those 5 years I have spent learning about networks, routing, multicast, protocol design and implementation, but most of all how to do research with the right sense for practicality. Nothing comes out of nowhere, and it would not be possible to gain the knowledge without the interaction with the large number of people I had the privilege to work with.

First of all, I would like to express my deepest gratitude to my advisors, Prof. Deborah Estrin and Dr. Ramesh Govindan who have been the primary source of knowledge and ideas, and have been extremely supportive through my study. I feel extremely lucky that I had the chance to work under their supervision. They gave me the right balance between the high-level picture and the low-level details. Indeed, it took more than two years for me to realize that my initial idealism for "optimality-first" is a very wrong approach. Thankfully, both of them were extremely patient with my stubbornness, and helped me learn by heart that "simplicity-first" is the right philosophy. Later in my study again and again I was able to verify that following this advice has a big payoff.

I would like to thank Dr. Sandeep Gupta for serving in my Qualifying Exam and Dissertation Committee, and for the valuable comments and feedbacks on this work. I am also thankful to Dr. Ashish Goel and Dr. Christos Papadopoulos, who were in my

Qualifying Exam Committee as well. The interesting discussions I had with Ashish on multicast state aggregation helped me look into the problem from a different angle. Christos was an extremely valuable source of information about reliable multicast, and he helped me to learn a lot about the subject. All the work about hierarchical reliable multicast in this dissertation was a direct result of his co-supervision.

I was very fortunate to have the chance to learn from many other people as well. Michael Speer helped me to look into the problems from engineering perspective during my Summer internship in Sun Microsystems in 1997. The numerous discussions with Mark Handley and Dave Thaler while working on MASC helped me to learn a lot about protocol design and specification. The 2-day dense discussion with Steve Hanna and his colleagues from Sun Microsystems Laboratory during the design of MASC was very important and fruitful. My colleagues from *dgroup* and *netgroup* at the USC Computer Networks and Distributed Systems Research Laboratory were the endless source of discussions, ideas and inspirations– Nirupama Bulusu, Alberto Cerpa, Xuan Chen, Jeremy Elson, Lewis Girod, Xinming He, Ahmed Helmy, Bau-Yi Polly Huang, Chalermek Intanagonwiwat, Amit Kumar, Satish Kanna Kumar, Ching-Gung Liu (Charley), Graham Phillips, Anoop Reddy, Reza Rejaie, Puneet Sharma, Hongsuda Tangmunarunkit, Kannan Varadhan, Haobo Yu, Yan Yu, Daniel Zappala, and Yonggang (Jerry) Zhao among others. Special thanks to Hongsuda who provided me with the data for most of the topologies that were used in this study, as well as for the valuable information about network topologies, and the insightful key-ideas and suggestions. The opportunity I had to work with Scott Shenker, Haobo, Hongsuda, Deborah and Ramesh on characterizing network topologies was very influential, and later part of that work became the foundation of my dissertation. I am very thankful to Paul

Francis, whose original work on Yoid was very inspirational, for opening me the door and giving me the direction to the new world of application-level multicast. My colleagues from the Yoid project in ISI, Yuri Pryadkin and Bob Lindell, were an invaluable source of information and enlightening discussions that were key-factors during the design of the Yoid mechanisms presented in this work.

Long time ago I would have taken a very different path, if it wasn't for my family–my parents Ivan and Maria, my sister Ionka and her two wonderful children Kristina and Ivailo, and my wife Kazuyo. Their love and support, but most of all their belief in me were the energy source I needed during this journey.

At different stages, this work was supported by the National Science Foundation through the Routing Arbiter project at ISI under Cooperative Agreement No. NCR-9321043, a gift-money from Sun Microsystems, and by the Defense Advanced Research Projects Agency through the Yoid/Yallcast project at ISI under Cooperative Agreement No. F30602-00-2-055.

Last, but not least, I would like to thank the people behind Internet sites such as the University of Oregon Route View Project (*http://www.routeviews.org/*) and the National Laboratory for Applied Network Research (NLANR) (*http://www.nlanr.net/*) that have provided the Internet research community with free access to extremely valuable network-related data and information.

Contents

List Of Tables

List Of Figures

Abstract

The self-organized growth of the Internet has resulted in Internet topology that is unplanned and completely decentralized. To the best of our knowledge, there is no study in the past that has considered how this affects (or can affect) network protocols. In this thesis we consider the impact of network topology on protocol performance. In particular, we ask the following three questions: What is the impact of the underlying topology on protocol performance? Can we use information about the underlying topology to improve protocol performance, and what gain can we expect? Can we use end-to-end mechanisms to design protocols that are adaptive to the underlying network topology?

In order to answer those questions, we study four specific problems that may be impacted by the underlying topology: multicast forwarding state aggregatability for network-layer multicast routing, Content Distribution Network replica placement, application-level and router-assisted hierarchical reliable multicast schemes, and application-level multicast routing.

Our findings are three-fold:

1. The underlying topology can make up to an order of magnitude difference in performance, but typically it causes at most a factor of two performance difference.

2. The performance of topology-sensitive protocols can be improved significantly even with limited knowledge about the underlying topology.

3. End-to-end mechanisms can be adaptive to the underlying network topology and participant placement; further, in many cases their performance can be comparable to the performance of topology-informed mechanisms.

In our study we use numerical simulations with a variety of network topologies and client placement, as well as real-world Internet experiments. As part of our study on application-level multicast routing we design and implement a set of algorithms for creation and management of application-level multicast data distribution trees. This implementation is now used in Yoid application-level multicast system.

Chapter 1

Introduction

One of the main reasons for the enormous success of Internet is its ability to support a large variety of applications with different requirements and characteristics: from email, file transfer, and remote telnet access to Web, real-time audio, video, and interactive games. Yet, it is decentralized in its organization and operation, and is self-evolving without explicit guidelines or central planning.

There are not many, if any, technological inventions that have been "left on their own" to grow, and have been as successful as the Internet. So, we ask, is there anything special about the Internet so it has accommodated, and continues to accommodate a large variety of applications, and at the same time it has sustained exponential growth without sacrificing performance or functionality.

One plausible answer for its success is its "openness" for everyone who wants to be a part of it. We, however, argue that this is necessary, but not sufficient to achieve the momentum it has.

Another answer is the *end-to-end* design principle of the Internet [105] that has been a guideline for designing new Internet protocols for more than two decades. The end-to-end

argument suggests that "specific application-level functions usually cannot, and preferably should not, be built into the lower levels of the system–the core of the network." The end-to-end principle explains the universality and adaptability of Internet, but does not explain why Internet has been able to grow exponentially and keep-up with the increased demand for performance. Indeed, as a result of the end-to-end principle, the network is not required to support any particular application, which not only simplifies the network management, but allows to build simpler and faster routers.

Faster routers and links are necessary to better performance, but there is no guarantee that all users will observe performance improvement, unless the whole Internet is upgraded or re-engineered to achieve the desired performance.

The work in this thesis is motivated by the following question:

Is there anything else beyond raw wire speed and router throughput that has impact on network performance?

In our search for an answer, we study the impact of network topology itself on protocol performance.[1] In particular, we address the following three questions:

1. What is the impact of the underlying topology on protocol performance? In other words, if the Internet topology were substantially different, should we expect similar performance?

[1]For the rest of this thesis we use the term "protocol performance" to denote the performance of either an abstract or a particular architecture. Also, we do not define explicitly the term "performance" in general, because it has to be associated with a particular metric that typically has meaning only within a particular architecture.

2. Can we use information about the underlying topology to improve protocol performance, and what gain can we expect?

3. Can we use end-to-end mechanisms to design protocols that are adaptive to the underlying network topology?

In order to answer these questions, we look into some specific problems that may be impacted by the underlying topology. Below are the four particular problems we consider, and a brief description of each of them.

1. *Multicast forwarding state aggregatability for network-layer multicast routing.* The multicast forwarding state is much more difficult to aggregate than the unicast forwarding state, because the outgoing interface set of each entry can change dynamically. Therefore, the amount of state in a router can be proportional to the number of existing multicast data distribution trees that use that router. The number of multicast trees a router belongs to depends not only on the number of active multicast groups and participants placement, but also the topology itself. E.g., the router in the center of a star topology will belong to all active multicast trees. Theoretically, this number is limited only by the size of the multicast address space (2^{28} and 2^{120} for IPv4 and IPv6 respectively), therefore the amount of state a router may need to keep can be beyond its capacity. By aggregating the multicast forwarding state in a router, that router can continue to operate and perform its functions.

2. *Content Distribution Network replica placement.* Server replication is used in Content Distribution Networks (CDNs) to improve the access latency, server availability, and to reduce network overhead. Because the number of replicas we can have is limited,

the question is, given some number of replicas, how to place them in the network to achieve maximum performance (*e.g.,* low access latency).

3. *Hierarchical reliable multicast schemes.* There is a variety of hierarchical schemes for reliable multicast. Some of them require explicit support from the network, while others do not make any assumption about the network (*router-assisted* and *application-level* multicast hierarchical schemes respectively). In our study we compare the performance of such two schemes.

4. *Application-level multicast routing.* Application-level multicast (also called *endsystem* multicast) is a truly end-to-end alternative to traditional, network-layer multicast, and has the advantage that it does not require special support from the network. Its drawback however is that the communication may be sub-efficient compared to network-layer multicast. Further, even if estimated performance difference may be acceptable, designing a practical scheme to minimize that difference may not be an easy task.

Question	Multicast state aggregation	Replica placement	Reliable multicast	Application-level multicast
Topology impact	X	X	X	X
Topology knowledge		X	X	
End-to-end mechanisms			X	X

Table 1.1: Relation between thesis questions and case-studies.

Table 1.1 summarizes the relation between each of the above examples we study and the questions we try to answer. For example, we use the the replica placement study, and

the hierarchical reliable multicast schemes study to understand better the impact of using knowledge about the underlying topology to improve protocol performance.

We should note that our interest of studying protocol performance is within the context of multi-party architectures. The impact of topology on protocols that involve only two participants can be simplified to studying single-chain topologies. We believe this is a problem that is also interesting to study, but is outside the primary focus of our work.

In our study we use a number of topologies, including a real-world router-level topology. As part of our evaluation, we consider the placement of the end-users participants (henceforth called *clients* or *receivers*) as well, because it is directly related to how a topology is "used."

How the findings from our case studies relate to the questions we ask in this thesis? Below we summarize those findings, which are also the main contribution of this thesis.

The underlying topology can make up to an order of magnitude difference in performance, but typically it causes at most a factor of two performance difference.

For example, the topology significantly affects multicast forwarding state aggregatability: for low receiver occupancy the aggregatability for some topologies can be on the order of 10 times and higher (*e.g.,* real-world topologies, random graph), but for other topologies it is on the order of 2 times (*e.g.,* tree, mesh, some generated topologies). In other cases, such as hierarchical reliable multicast and application-level multicast, the performance results do not vary significantly for different topologies (typically the difference is within a factor of 2). Based on the particular metrics we use in our evaluation, we believe that the

topology impact is smaller on protocol performance metrics that are based on relative values (*e.g.,* if the metric is the relative tree size difference between two protocols), because the topology factor that has impact on protocol performance will, in most cases, affect similarly all evaluated protocols. On the other hand, the topology impact is larger on protocol performance metrics that are based on absolute values (*e.g.,* a metric such as multicast distribution tree size in number of hops). The only exception among the topologies we have studied is the mesh topology for which even the relative performance difference is notably larger than the other topologies. Based on those findings we conclude that the topology indeed has impact on performance, but in many cases that impact is on the order of two.

The performance of topology-sensitive protocols can be improved significantly even with limited knowledge about the underlying topology.

A topology-sensitive protocol would benefit most from information about the underlying topology if it has knowledge about the complete topology. However, for large networks such as the Internet it not practical to require complete topology information for reasons such as lack of a scalable mechanism for collecting the information and keeping it up to date. Therefore, a protocol should not require complete knowledge about the topology in order to operate properly and efficiently. Instead, it should, whenever possible, use limited topology knowledge as hints to improve performance, as long as this knowledge is easily inferred. Our argument, supported by our findings, is that even small amount of information may be useful to improve protocol performance. For example, information about node fanout may be sufficient to place a limited number of replicas in the network and

achieve performance that usually requires much more complicated solutions. Similarly, the performance of application-level reliable multicast hierarchies can be improved significantly to a level that is comparable to the performance of solutions that require router support, if information such as end-to-end node distance is available to the participants. Therefore, a protocol should be allowed to use mechanisms to collect information about the topology, as long as those mechanisms are practical and scalable.

End-to-end mechanisms can be adaptive to the underlying network topology and participant placement; further, in many cases their performance can be comparable to the performance of topology-informed mechanisms.

If no topology-related information is available, it is possible to design protocols that use only end-to-end mechanisms and that are adaptive to the underlying network topology and participant placement. In many cases, their performance is comparable to the performance of protocols that use the help of topology information. For example, in case of application-level multicast we can use end-to-end mechanisms to improve significantly the end-to-end data propagation latency so it can be within a factor of 2.5–3.0 of the unicast latency. Similarly, application-level assisted hierarchies for reliable multicast data recovery can be modified to use traceroute mechanisms to compute the number of hops between participants, and then this information can be used to create an efficient hierarchy that has performance typically within a factor of two of router-assisted hierarchies.

We acknowledge that network topology is not the only factor that has impact on performance. It, however, is one of the major factors, or at least a factor that can explain

the less-than-expected impact that other factors such as client location can have on protocol performance. Our study is far from being completed. It is only a small step toward understanding the mechanisms behind complex systems such as the Internet.

The rest of the dissertation is organized as follows. Chapter 2 is our case-study of topology impact on protocol performance within the context of multicast forwarding state aggregatability. In that chapter we also describe some practical solutions of the problem. Chapter 3 is our case-study of the replica placement problem within the context of Content Distribution Networks (CDNs). In Chapter 4 we study the performance of hierarchical reliable multicast schemes. In Chapter 5 we present our work on application-level multicast and demonstrate its adaptability. Related work is in Chapter 6. Conclusions and future work are in Chapter 7. Appendix A contains our work on multicast address allocation, which is not directly related to the main topic of this thesis, but relates to our work on multicast forwarding state aggregation.

Chapter 2

Case Study: Multicast Forwarding State Aggregation

Scalable and efficient distribution of multicast data is achieved by multicast routing protocols. These protocols usually construct a multicast distribution tree, and create state in the intermediate routers. Due to the difficulties of aggregating the multicast state, the amount of state in a router can be proportional to the number of multicast distribution trees it belongs to, and theoretically this number can be as large as the multicast address space (*e.g.*, 2^{28} addresses for IPv4). Intuitively, the amount of state in all routers will be different; *e.g.,* the core routers will have more state than edge routers. In this chapter we investigate the topology sensitivity of multicast forwarding state aggregatability. Our study shows that the state is aggregatable, but the aggregatability is impacted by the underlying topology. However, for all topologies we investigated, the aggregation ratio is less than expected. In the second part of the chapter we describe one solution to the problem: a *leaky* aggregation mechanism that can be used to improve aggregation at the expense of small network bandwidth overhead.

2.1 Introduction

Today, an increasingly wide variety of applications are based on IP multicast [25]: audio and video transmissions [10] data replication [39], Web caching [135], and collaborative workspaces [36]. These applications owe their success to a key property of today's multicast infrastructure: mechanisms for scalable distribution of data flows to multiple receivers that avoid replicating data traffic on shared links of the distribution paths (see Figure 2.1). The design of these applications is greatly simplified by a key feature of the IP multicast service model: the level of indirection provided by logical group naming [77].

Scalable and efficient distribution of multicast data is achieved by multicast routing protocols. These protocols usually construct a multicast distribution tree. They contain mechanisms that enable receivers in a given multicast group to rendezvous with senders, and then establish the corresponding forwarding state in routers that achieves efficient distribution. Since Steve Deering's original work on multicast datagram service [25], various distribution tree building and rendezvous mechanisms have been proposed and deployed [94, 84, 26, 33, 4, 115]. Some of these mechanisms have limited scalability (within a single domain), while other have been designed to scale to the entire Internet.

Scalable routing mechanisms alone may not ensure the scalability of the Internet multicast architecture. Specifically, we believe there is an architectural argument for considering *multicast forwarding state aggregation.* This argument proceeds as follows. The logical naming feature of IP multicast necessitates per-group forwarding entries in routers. Each such entry implies a "cost" to the overall architecture. This cost is affected by the traffic bandwidth associated with the group. Creation of forwarding entries for high bandwidth

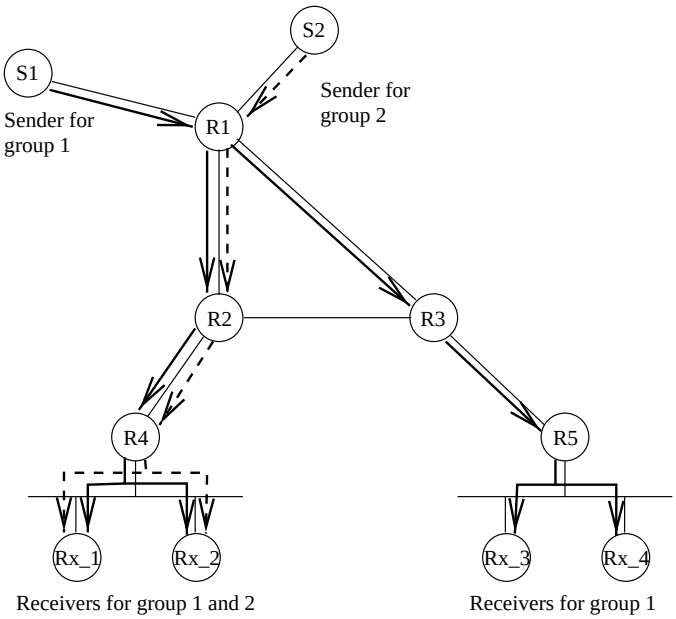


Figure 2.1: Example of multicast data distribution.

groups (*e.g.,* for example video and audio sessions) is justified by the bandwidth savings of using multicast. To justify supporting relatively low-bandwidth groups (*e.g.,* whiteboard-type applications [36] and event notification services [9]), however, we need mechanisms that amortize the cost of maintaining forwarding entries over many low-bandwidth groups. One way to do this is to *aggregate* multicast forwarding entries for low-bandwidth groups. To our knowledge, this problem has not been widely considered in the literature.

Even apart from this architectural argument, there is a practical reason for studying multicast forwarding state aggregation. Unicast forwarding state aggregation is well-understood, but, because multicast receivers are not—in general—topologically related, it is unclear whether these unicast techniques apply. Moreover, the multicast problem is potentially much greater compared to unicast, because the possible number of multicast groups grows combinatorially with the number of network nodes. For example, IPv4 [93]

allows up to 2^{28} multicast groups. Network capacity permitting, it is not inconceivable that some significant fraction (say 50%) of these groups are simultaneously in use. In this case, some routers may need more than 0.5 GB memory to store the forwarding entries. Under similar assumptions, an IPv6 [27] router might need several Terabytes for the forwarding entries!

In this chapter we look into the problem of multicast forwarding state aggregation for network-layer multicast routing. First we explain why aggregation is important, and we describe the problem in more details. Then, we investigate the impact of topology on the state aggregatability in general, without having in mind any particular aggregation scheme. Finally, we look into practical mechanisms for multicast forwarding state aggregation, and describe and evaluate one possible solution. Our solution targets the scaling of forwarding state to the number of high-bandwidth groups. It does this by aggregating, where possible, low-bandwidth groups. This aggregation allows for *leaks*[1]—traffic for a group may follow paths that do not lead to any receiver for that group. The aggregation strategy limits the bandwidth allocated for leaks to a fixed fraction of link capacity at each router.

The rest of the chapter is organized as follows. Section 2.2 describes IP multicast and presents the difficulty of the problem. Section 2.3 looks into the topology impact on state aggregatability. Section 2.4 describes several aggregation strategies, and works out the details of one practical solution to the problem, the leaky aggregation strategy. Section 2.5 shows that, for a wide variety of traffic mixes and receiver distributions, the leaky aggregation strategy manages to closely track the number of high bandwidth groups, even with dynamic data traffic and a large number of joins/leaves. Section 2.6 discusses

[1]To our knowledge, Van Jacobson was the first to suggest this tradeoff [56] for multicast forwarding.

the interplay between leaky forwarding state aggregation and existing multicast routing protocols. Section 2.7 contains a description of how to avoid some problems in case of multicast routing protocols such as BGMP [66] and CBT [4]. Finally, Section 2.8 contains the pseudo-code of the leaky aggregation algorithm.

2.2 Motivation

In the previous section, we argued that multicast forwarding state aggregation is necessary because forwarding state incurs a bandwidth-dependent architectural cost. Before we develop this argument further, we briefly describe multicast forwarding and forwarding state aggregation.

2.2.1 Multicast Routing and Forwarding

Multicast routing protocols achieve efficient data distribution from senders to receivers. They do this by setting up distribution trees that span all group members. The subject of scalable tree construction protocols has received much attention in the literature [94, 84, 26, 33, 4, 115]. A common feature of these protocols is that they all build multicast distribution trees rooted at some particular node.

The outcome of every tree construction protocol is a collection of *forwarding entries* at each router in the network. Together, these forwarding entries effect data distribution from sources to receivers. At a given router, the forwarding entry determines which neighboring routers receive copies of an incoming multicast packet. For example, Router R1 in Figure 2.1 has forwarding entries for group 1 and group 2. The forwarding entry for group 1 specifies that links R1-R2 and R1-R3 belong to the distribution tree of group 1. The

forwarding entry for group 2 specifies that link R1-R2 belongs to the distribution tree of group 2.

In its most general form, a multicast forwarding entry contains four pieces of information: a source address *prefix* (an initial bit-substring of an address—IPv4 prefixes are usually represented by, for example, by an IP address followed by the length of the initial bit substring `123.4.56.0/24`), a group address prefix, an incoming interface set, and a set of outgoing interfaces. For a given forwarding entry, its address prefixes represent the range of sources and groups whose distribution tree is represented by that entry. The incoming interface represents the router's "parent" in that distribution tree, and the outgoing interfaces represent the router's children. A forwarding entry *matches* a multicast data packet if its source address prefix is the longest initial bit-substring (among all other forwarding entries) that matches the packet's source address, its group address prefix is the longest initial bit-substring that matches the packet's group address, and the interface on which the packet was received is included in its incoming interface set. A copy of every such packet is forwarded on the outgoing interface set (Figure 2.2).

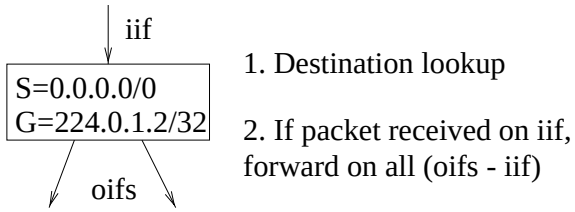


Figure 2.2: Forwarding a multicast packet for source = "any" and group = 224.0.1.2

Existing multicast routing protocols differ slightly in the types of forwarding entries they generate:

Group specific vs source-group specific: In group-specific entries, the source address prefix has zero length. That is, any source address matches that prefix. Usually, these entries are created by protocols that build distribution trees rooted at a group-specific *Rendezvous Point* [33, 4] or *Root Domain* [115]. Source-group specific entries contain non-zero source and group address prefix lengths. This type of forwarding entry is usually created for distribution trees rooted at the source [94, 84, 26, 33, 50, 91], but [4] is a notable exception.

Uni-directional vs bi-directional A given multicast routing protocol may define forwarding entries to be either *uni-directional* or *bi-directional*. Uni-directional forwarding entries result in multicast traffic that flows only "down" the distribution tree (from the root of the tree towards the receivers) [94, 84, 26, 33, 50]. Bi-directional forwarding entries result in distribution trees that allow traffic from a sender to reach its nearest router on the tree, then traverse the tree simultaneously towards the root and towards receivers [4, 115, 91]. To achieve bidirectional forwarding, an entry's incoming interface set trivially includes all interfaces of a router.

These semantic differences between forwarding entries is useful for understanding the design of forwarding state aggregation mechanisms.

2.2.2 Forwarding State Aggregation

But what do we mean by forwarding state aggregation? Consider Figure 2.3. This shows two forwarding entries with matching incoming interface and outgoing interfaces. The group address prefixes of these entries are adjacent—*i.e.,* there exists a single address prefix that includes both those group address prefixes. In such situation, these two multicast

forwarding entries can be *aggregated* (represented by a single entry) as shown. This is not the only way to aggregate multicast forwarding entries; Section 2.4 describes other approaches.

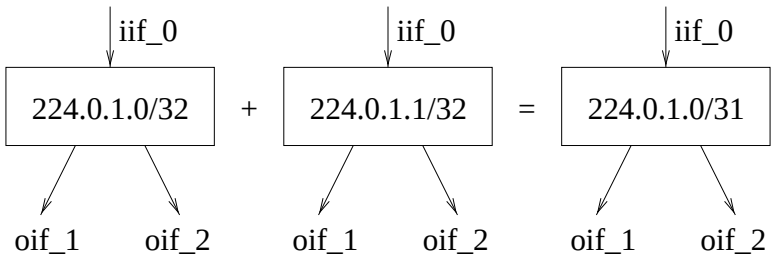


Figure 2.3: Example of aggregating multicast forwarding entries.

Notice that the example in Figure 2.3 omits the source prefix. In this work, we focus on *aggregating group-specific multicast forwarding entries*. Recall that such entries match any source address. Group-specific entries are created by multicast routing protocols that build *shared* (rather than per-source) distribution trees. It is generally acknowledged that scalable inter-domain multicast routing can only be achieved using shared trees [115].

The following subsections consider these questions: Why is it important to solve the problem of aggregating group-specific forwarding entries? Why is it hard? Is the problem solvable?

2.2.3 Why is aggregation important?

One way to argue for the importance of multicast forwarding state aggregation is to observe the following: there is no "natural" limit to the number of concurrently active multicast sessions in an internet. The capacity of the network bounds, in some loose sense, the number

of concurrent audio or video sessions. However, the same cannot be said of all multicast applications. One example is an event notification service [9]: event generators intermittently multicast events to interested clients. Another example is congestion-adaptive multicast applications (*e.g.,* file transfer, multicast distribution of software updates) that make use of whatever bandwidth is available to them. This problem of unbounded multicast forwarding state growth is particularly crucial in *core* or backbone routers (Figure 2.4).

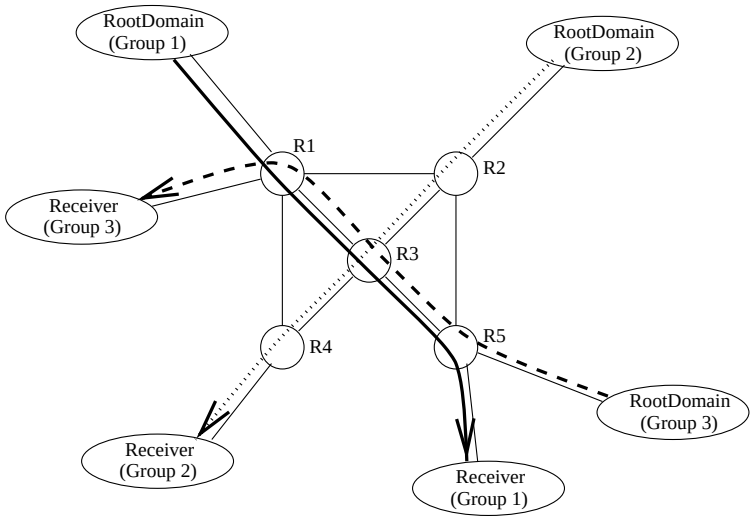


Figure 2.4: Example of multicast forwarding entries concentrations: the router at the center (R3) has the largest number of entries (three).

In the absence of a natural limit to the number of concurrently active multicast groups, the size of the multicast groups is limited only by the available address space: 2^{28} for IPv4 and 2^{120} for IPv6. Then, if a router has 32 interfaces, the memory required to store multicast forwarding entries (assuming a 50% address space utilization and one bit of information per interface per entry) is at least 512 MB for IPv4 and 10^{24} Terabytes of memory for IPv6! It is conceivable that, within 3-4 years, high-end routers will have enough

high-speed forwarding table memory to satisfy the needs of IPv4. However, forwarding state aggregation may be imperative for scaling IPv6.

Apart from this pragmatic argument based on technological trends, an architectural argument can be made for multicast forwarding state aggregation. This argument is based on the utility of multicast forwarding entries. Informally, the utility of a multicast forwarding entry to the infrastructure increases in proportion to the corresponding group's bandwidth. Thus, bursty low-bandwidth applications like event notification [9], or non-bursty low-bandwidth applications (like periodic Web document invalidations [104, 124]) generate the least utility; their infrequently used forwarding entries occupy expensive router memory. It is unclear whether such low bandwidth groups will predominate in the future. However, we believe that the architecture of an internet should not artificially constrain the numbers of such groups. For this reason, we argue the need for a forwarding table structure that provides higher overall utility with respect to forwarding table occupancy.

2.2.4 Why is it hard?

Aggregation of unicast forwarding entries is well understood [103]. This aggregation is achieved by careful unicast address assignment. Topologically contiguous networks are assigned adjacent address prefixes. In this way, an entire autonomous system (AS) can sometimes be represented in the forwarding table by a single forwarding entry in a backbone router.

These techniques do not apply to multicast forwarding. Today, multicast group addresses are largely assigned in topologically independent way. One proposal for inter-domain multicast address assignment [115] proposes to dynamically assign blocks of group

addresses to individual ASs.[2] Even if they were assigned using this method, the receiver distributions for adjacent groups could differ. That is, there is little likelihood of having many situations that correspond to Figure 2.3.[3]

In the next section we look into the state aggregatability in general, from information-theory perspective, to have some insensitive whether it is solvable. After that we look into one possible solution of the problem.

2.3 Topology Impact on Multicast Forwarding State Aggregatability

In this section, we investigate the topology sensitivity of multicast forwarding state aggregatability. Unicast forwarding tables aggregate well because addresses are, to some extent, assigned topologically [103]. Unfortunately, multicast forwarding information cannot be aggregated using similar mechanisms, because the outgoing interface set of a single entry changes dynamically and is determined by the existence of downstream receivers. Without any aggregation (and assuming a bidirectional shared tree routing protocol) the number of multicast forwarding entries in a router can be proportional to the number of active multicast groups.

Interface-centered multicast forwarding state aggregation technique is proposed in [116]. This technique essentially represents the entire multicast forwarding table as a collection of per-interface bit strings. These bit strings indicate if, for the corresponding groups, a

[2] See also Appendix A for details and evaluation of this architecture.

[3] Some have argued that, when multicast addresses are assigned topologically, there is some chance of receivership congruence. For example, the population interested in any group initiated within AS X is likely to be the same. This is an interesting, but yet unverified, hypothesis.

specific interface belongs to the outgoing interface set. The aggregation technique simply compresses these bit strings using run-length encoding for example. Such an aggregation technique is *strict*. By contrast, later in this chapter we consider *leaky* aggregation, where, for some groups, outgoing interface sets are not correctly preserved in the aggregation, resulting in traffic leakage in directions where there are no receivers.

Our focus in this section is on the topology dependence of strict aggregation. However, rather than evaluate a particular aggregation technique, we investigate the bounds of the aggregatability of the multicast forwarding state. To do this, we consider a simple information-theoretic approach. For a given group, let p be the probability that a particular interface of a router is an outgoing interface for that group. Then, the number of bits needed to represent this information is simply the *entropy* [21]:

$$H(Interface, Group) = H(p) = -p * \log_2(p) - (1-p) * \log_2(1-p) \qquad (2.1)$$

Our metric for aggregatability, *interface entropy*, is defined as the average entropy of an interface over all nodes.

To compute our aggregatability metric, we use the following methodology. We first uniformly select a node in the topology to be the root of the multicast distribution tree for a given group. Then, we assume that each node in the network has an identical probability of having an attached receiver (we call this the receiver probability). Tracing the reverse path from the receiver to the source, we can assign the corresponding probability to interfaces along the path. Where there exist multiple equal-cost paths, we assume that each of these paths has an equal probability of being chosen. In this manner, we can compute, for each

router, the average interface entropy for the group. For a given topology, we repeat this procedure for different receiver probabilities and average the results over different choices for tree root.

Topology	Nodes	Links	Diam.	Ave. dist.	Ave. fanout
Transit-stub	1008	1399	20	8.8	2.8
Tiers	5000	7084	39	15.3	2.8
Waxman	5000	18046	8	4.5	7.2
Mbone	4179	8549	26	10.1	4.1
AS	4830	9077	11	3.7	3.8
Internet core	54533	146419	23	7.6	5.4
Random	5000	24926	7	4.0	9.8
Mesh	10000	19800	198	66.7	4.0
Reduced mesh	10000	9999	199	83.2	2.0
K-ary tree (3-ary)	9841	9840	16	14.0	2.0

Table 2.1: Metrics of used topologies.

In this study we look into several topologies. Some of them are generated by topology generators, other are real-world topologies created by collecting various data. We use some canonical topologies as well. Below is a brief description of each topology. Some of the characteristics of those topologies are summarized in Table 2.1.

The generated topologies we use in our study are:

Transit-stub This is a topology generated by the Georgia Tech Internet Topology Modeler (GT-ITM) [8], a topology generator that has a number of parameters to define the generated topology. Typically, it is used to generate two-level topology. Initially it creates a number of top-level transit domains within which edges are assigned at random. Each transit domain has attached to it several stub domains that are

generated similarly. Additional stub-to-transit links are added at random as well, based on a parameter.

Tiers This is another generated topology, but by a different topology generator [31]. First, a number of top-level networks are created, and each of them has several intermediate tier networks attached to it. Similarly, several LANs are attached to each of the intermediate tier networks. Within each tier (except LANs which always have star topology), the generator connects all nodes by a minimum spanning tree, based on the Euclidean distance between every two nodes. Then, it adds more links in order of increasing inter-node Euclidean distance. Finally, a number of inter-tier links are added at random, based on a parameter.

Waxman This is another topology generated by GT-ITM, but is based on a different topology model [127].

First, the network nodes are randomly distributed over a rectangular coordinate grid and the links between every two nodes are added at random with probability based on the Euclidean distance between them:

$$P(x, y) = \beta e^{\frac{-d(x,y)}{\alpha L}},$$

where $d(x, y)$ is the distance between node x and node y, L is the maximum possible distance between any pair of nodes, and α and β are parameters used to modify the shape of the network ($1 \geq \alpha > 0, \beta > 0$).

The real-world topologies we use are:

Mbone This is a router-level topology of the Mbone [75] multicast network overlay. It is obtained by recursively querying each multicast router for its neighboring routers [123].

AS This domain-level topology of the inter-autonomous system (AS) connectivity is created based on the BGP AS path information carried by the backbone routers [122, 12]. Even though this is not a router-level topology, we include it for completeness.

Internet core The topology information was collected by using a large number of traceroute requests sent over the Internet [38, 123]. The resulting topology had 228263 nodes and 320149 links. Then we recursively removed all nodes that have a fanout of one to obtain a topology we call *Internet core*. The reason that we truncate the original topology is to remove the long, "skinny" branches that do not represent well the network connectivity at the edges, but are an artifact from the particular methodology used to obtain the topology information.

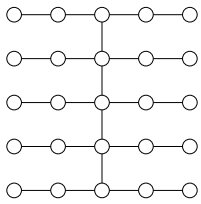


Figure 2.5: Example of a reduced mesh topology.

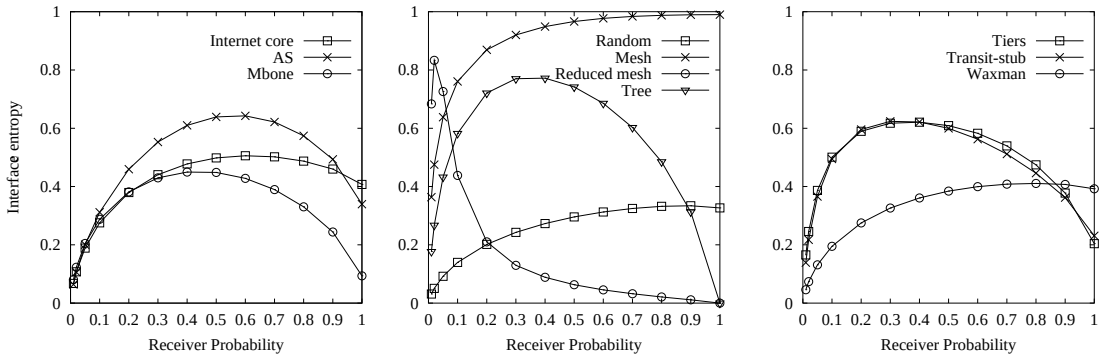
Finally, we consider the following canonical topologies as well:

Random All links in this graph are assigned at random, with uniform probability.

Mesh Also known as *grid*.

Reduced Mesh We start with a mesh, and the remove all vertical links, except the links of the column in the middle. See Figure 2.5 for an example.

K-ary tree An uniform tree where each node has k children, except the leaf nodes. In this study we use a 3-ary tree of depth 8.



(a) Real topologies (b) Canonical topologies (c) Generated topologies

Figure 2.6: Average interface entropy.

Interface Entropy Results

Figure 2.6 shows the average interface entropy, as a function of receiver probabilities, for all topologies described above (smaller entropy corresponds to better aggregatability).

The first observation we can make is that each of the canonical topologies shows very different behavior (Figure 2.6(b)). The entropy for both the mesh and the random graph increases monotonically with receiver probability, but for the mesh, unlike the random graph, the entropy approaches 1 in the limit of complete occupancy. This is because of the prevalence of equal-cost shortest paths in the mesh. On the other hand, the tree and the

reduced mesh have single-peaked, although completely distinguishable behavior, with zero entropy in the limit.

All real topologies (Figure 2.6(a)) are single-peaked with non-zero entropy in the limit. In this sense, their behavior represents some combination of a tree and a random graph. Of the three real topologies, the Mbone topology resembles a tree most. On the other hand, the Internet topology is closest to a random graph. Among the generated topologies (Figure 2.6(c)), the Waxman closely approximates a random graph. Tiers and Transit-stub qualitatively resemble the single-peaked, non-zero limiting behavior of real topologies.

We also investigated the worst-case interface entropy, defined as the maximum interface entropy averaged among a number of groups rooted at different nodes. With this metric, the distinctions between topologies disappear because in almost all cases the worst-case entropy is close to 1 (*i.e.,* the state is not aggregatable),

These results reveal that, for very low occupancies (about 3%), routing state for a single group can be compressed by up to a factor of 10. For receiver probabilities on the order of 10%, the aggregatability decreases to 3. This result is true for all real-world topologies. On the other hand, the aggregatability of random graph is notably better, than real-world topologies, and can be on the order of up to 20 times for very low occupancy. Those results qualitatively match the findings in [116] where the interface-centered technique achieves some, but not dramatic, aggregation. Finally, recall that we chose uniform receiver placement. For this reason, our results may not reflect reality, but lacking probabilistic models for receiver clustering, uniform placement is our only available choice.[4]

[4]The client placement models described in Section: 3.2.3 can be used to assign specific client placement, and cannot be used as client placement probabilistic models.

2.4 Leaky Aggregation of Multicast Forwarding State

2.4.1 Is there hope?

Before we answer this question, we describe a desirable goal for multicast forwarding state aggregation. This goal is motivated by an observation in Section 2.2.3, that the utility of a multicast forwarding entry is proportional to the bandwidth of the corresponding group(s). A desirable goal for forwarding state aggregation, then, is *to scale the worst-case forwarding table size approximately proportional to the number of high-bandwidth groups.* This goal is desirable because the number of high bandwidth groups is naturally limited by the available network capacity.

Our statement of the scaling goal is deliberately imprecise. A more precise statement is not essential to our argument. Furthermore, it is hard to quantify what exactly *high bandwidth* means. Intuitively, by this we mean audio and video applications that have fixed bandwidth requirements, or are rate-adaptive within some relatively small bandwidth range. By contrast, we use the term *low bandwidth* to refer to applications such as event notification [9] or multicast Web document invalidation [104, 124]. We also do not attempt to quantify our scaling goal. Rather than trying to achieve a fixed aggregation target (*e.g.*, a table size that is no more than 5% of the number of groups), we ask what aggregation is achievable with bounded resource utilization.

We would like to achieve this goal, as we've said before, for aggregating group-specific entries. Furthermore, our approach leverages the topological assignment of group addresses; the root of the shared tree for a group is, with high likelihood, topologically close to that of a group with an adjacent address. Several recent proposals for an inter-domain multicast

architecture share this property [115, 91, 50]. For consistency, we use the terminology introduced in [115]: all shared trees are rooted at a *root domain* and multicast *group prefixes* are assigned to individual domains. We believe that, with these assumptions, there exists a plausible hypothesis that the scaling goal described above can be achieved.

That hypothesis proceeds from a three step argument.

- With our assumptions listed in the previous paragraph, we can expect that forwarding entries for adjacent groups share the same incoming interface.

- Second, especially in core routers (Figure 2.4), we can expect that these forwarding entries' outgoing interface sets overlap significantly. This observation is based on the bounded fanout of routers vis-a-vis the receiver distribution. For example, if a group has more than a hundred receivers, a forwarding entry for that group in a core router can be expected to have, in its outgoing interface set, many of that router's interfaces.

- Finally, we can aggregate adjacent low-bandwidth groups by "leaking" one group's traffic down some of the other group's outgoing interfaces (see Section 2.4.4).

To complete our argument, we note that recent technological advances, particularly the deployment of dense Wavelength Division Multiplexing [64] technologies, can result in dramatically cheaper bandwidth. This trend will increase the disparity between switching and bandwidth costs, especially for backbone or core routers. For this reason, we believe that a "bandwidth-state" tradeoff will be more justifiable in the future than it has been until today. Our hypothesis, then, is that such leaky aggregation can achieve our scaling goal.

2.4.2 Design Space

Before we describe strategies for forwarding entry aggregation, we list the requirements that constrain the design space:

- Aggregation must *conserve joins*. Thus receivers who have joined a group must continue to receive traffic even if the corresponding forwarding entry has been aggregated.
- The aggregation strategy must be largely independent of the multicast routing protocol. In particular, it should not require additional routing protocol modification or control traffic to work. If a particular multicast routing protocol has to be modified, the modifications must be very simple, and the additional control traffic, if any, must not result in another scalability issue.
- Routers that aggregate forwarding entries must be able to interoperate with those that do not. Furthermore, the aggregation strategy must not require a particular deployment order (*e.g.,* core routers before leaf routers or vice versa).

In Section 2.4.1, we hypothesized that leaky aggregation can help us scale the size of the multicast forwarding table approximately proportional to the number of high-bandwidth groups. In this section, we present a leaky aggregation strategy. First, however, we present some *non-leaky* aggregation strategies. These strategies do *not* tradeoff bandwidth for reduced table size, and form a baseline for understanding the performance of leaky aggregation.

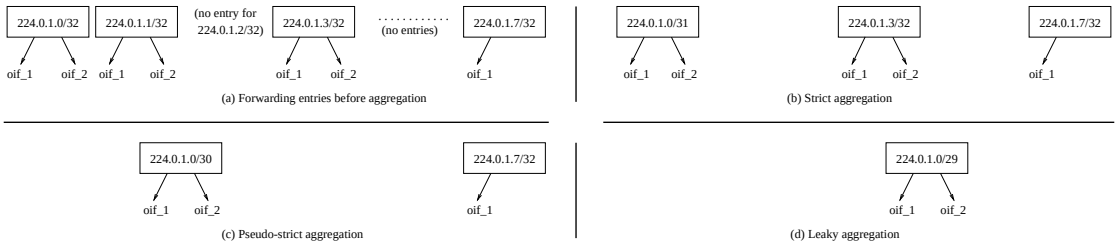


Figure 2.7: Example of prefix-based strict, pseudo-strict and leaky aggregation.

2.4.3 Non-Leaky Aggregation

The simplest non-leaky aggregation strategy is what we call *strict* aggregation (Figure 2.3 and Figure 2.7(b)). In this strategy, two adjacent group-specific forwarding entries are aggregated if and only if:

- Their incoming interfaces match.
- Their outgoing interface sets match.

Even if the root domains of adjacent groups are topologically aligned, we expect little likelihood of there being many group prefixes that satisfy these conditions.

A variant of strict aggregation is what we call *pseudo-strict* aggregation (Figure 2.7(c)). In this form of aggregation, two group prefixes are replaced by the longest covering prefix if and only if:

- Their incoming interfaces match.
- Their outgoing interface sets match.
- There is no intervening forwarding table entry whose group prefix matches the longest covering prefix.

Thus, in Figure 2.7(c), we can aggregate the entries for `224.0.1.0/31` and `224.0.1.3/32` *even though* these are not adjacent prefixes (prefix `224.0.1.2/32` lies between them). This is because that router has not received a join for group `224.0.1.2/32`. Note that this strategy does not "leak" traffic destined for `224.0.1.2/32`! If the router did not get a join for `224.0.1.2/32`, it could not have propagated a join upstream, and should not receive any traffic for that group.[5] Intuitively, we expect that in many cases pseudo-strict aggregation can compress better the routing table than strict aggregation.

2.4.4 Leaky Forwarding Entry Aggregation

The basic idea behind leaky forwarding state aggregation is simple. Leaky aggregation relaxes the requirement in pseudo-strict aggregation that the outgoing interface sets of the entries must match. A data packet that matches the resulting forwarding entry will be forwarded on all interfaces on which joins have been received, but it may be forwarded on some other interfaces as well (*i.e.,* those for which no Join message was received). Figure 2.7(d) shows an example of prefix-based leaky aggregation. Group `224.0.1.7/32` does not include `oif_2`, but `oif_2` is included in the aggregated entry `224.0.1.0/30`.

Obviously, leaky aggregation wastes some bandwidth. For this reason, such aggregation must be performed carefully, trading off as little increased bandwidth as possible for maximal reduction in forwarding table size. To do this, we first identify low-bandwidth groups, and only attempt to aggregate such entries. Such controlled leaking is not entirely free of problems. Section 2.6 describes some of the limitations of this approach.

[5] On shared LAN, however, a join message sent by one router to its upstream neighbor can cause traffic to "leak" through other downstream routers.

The design of a leaky aggregation strategy poses several challenges:

- To carefully tradeoff bandwidth and forwarding table size, we need to estimate the current bandwidth of a multicast group. Although techniques are known for estimating the rate of unicast traffic flows [35], such techniques react at congestion time scales. For leaky aggregation, it probably suffices to detect application-level phase changes (*e.g.,* a break in a video lecture).

- In our simplified description above, we have suggested only aggregating the "low-bandwidth" groups and not aggregating the "high-bandwidth" groups. In practice, our algorithm cannot assume the existence of a bi-modal bandwidth distribution, nor can it assume that there is any *a priori* bandwidth threshold for groups.

- As a corollary to the previous point, our strategy must limit the bandwidth attributable to leaks to some relatively small fraction of link capacities (*i.e.,* router administrators should be able to locally limit the bandwidth attributable to leaks, to, say 5% of link capacity). This is a key design challenge: achieving maximal reduction in table size while still limiting the amount of wasted bandwidth. In what follows, we use the term **leak budget** to refer to this limit.

- Finally, the scheme must take into account receiver joins and leaves. These may cause the outgoing interface set of forwarding entries to change.

Our leaky aggregation scheme involves two related components. First, a simple technique for estimating the rate of individual groups. Second, a heuristic that, given the rates of individual groups, and the link capacities, computes aggregated forwarding state in a manner that does not violate the leak budget. We now describe these two components.

2.4.5 Estimating the bandwidth of individual groups

Our basic approach to estimating the bandwidth of an individual group is to count the number of data packets that are forwarded using that group's forwarding entry over some time interval T. This coarse estimation suffices for us, and does not require a change to router forwarding engines—some major router vendors already support such counters. In this work, we do not suggest a value for T; further experimentation is needed to determine this value.

Clearly, we cannot estimate the bandwidth of all groups simultaneously. To do this, the router would need to install group-specific (unaggregated) forwarding table entries for every group for which it has received joins. This defeats the purposes of aggregation. There is an alternative solution, however. Recall that we have assumed topological assignment of group addresses. In BGMP for example, a block of group addresses is assigned to each root domain. Our solution, then, is to only concurrently estimate rates for (*i.e.,* install group-specific forwarding entries for) groups rooted at the same root domain. In this way, we "stagger" the bandwidth estimation across root domains, ensuring that the instantaneous forwarding table is never proportional to the total number of groups for which we have received joins. This approach, however, exhibits poor responsiveness to traffic pattern changes. This may not be a significant drawback, since our goal is only to detect application-level phase changes, and not to try and respond at congestion time scales.

In order for our aggregation scheme to work, we cannot only estimate the bandwidth of individual groups. *We also need to estimate the incoming traffic for groups that neighboring routers are leaking.* This estimation figures into each router's aggregation strategy

(Section 2.4.6). If a router's leak budget permits, it can choose to further propagate these *incoming leaks*. Otherwise, it can install forwarding entries with an empty outgoing interface set to prevent these leaks. To estimate the incoming leaks, we install prefixes corresponding to "holes" in the prefix associated with the root domain. For example, in Figure 2.8 the router would need to estimate the bandwidth not just on `224.0.1.0/32` and `224.0.1.7/32`, but also the leaks on the intervening four prefixes. For a given root domain, the number of prefixes needed to fill the holes is, in the worst case, proportional to $N \log_2 b$ where N is the number of groups rooted at that domain and b is the size of the prefix associated with the root domain. We believe this is still within acceptable bounds for the transient increase in forwarding table size.

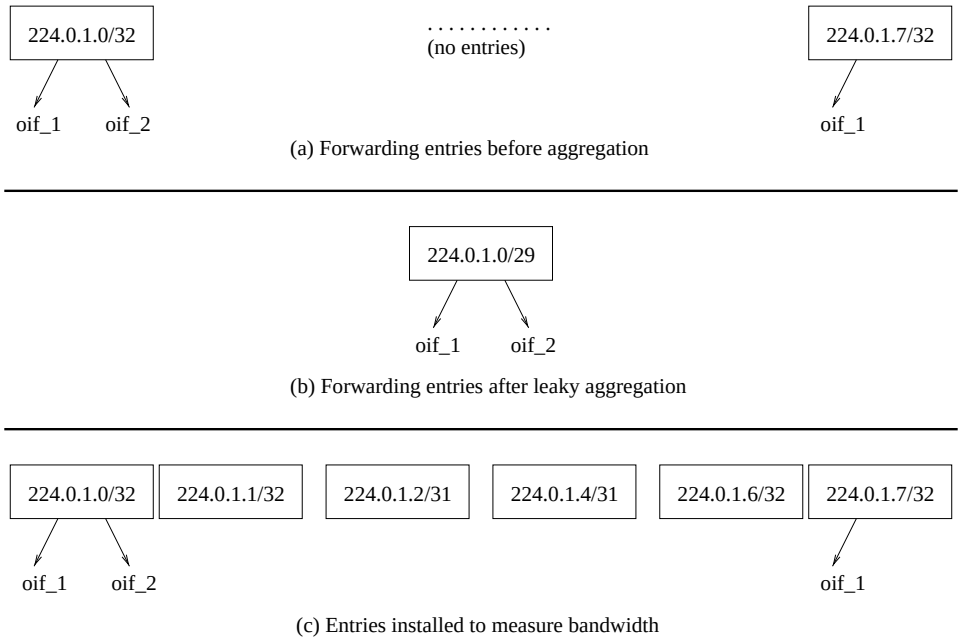


Figure 2.8: De-aggregation for bandwidth estimation.

In summary, a router periodically *de-aggregates* all prefixes allocated to a single root domain, and installs the group-specific forwarding table entries. After time T, it reads

the counters associated with each forwarding table entry and uses these as input to the algorithm described in Section 2.4.6. After computing the aggregates, it installs the corresponding forwarding entries, and deletes the group-specific entries. Because only one or few prefixes are de-aggregated at a time, and those prefixes would cover a fraction of all joined groups, the remaining entries (a larger fraction of all groups) can be aggregated, and therefore the aggregatability of those entries will contribute to the overall savings. A pseudo-code description of the algorithm is presented in Section 2.8.1. This algorithm need *not* be implemented in the forwarding fast path and should not affect router forwarding performance.

2.4.6 Aggregation Heuristic

Given the coarse grain bandwidth estimates for each forwarding entry (Figure 2.9(a)), the problem is to compute the smallest possible forwarding table size that fits within the leak budget of every link. This section describes how we compute the aggregated forwarding table.

First observe that, given bandwidth estimates for two adjacent groups, we can easily compute the leaks resulting from their aggregation. For example, consider the groups `224.0.1.0/32` and `224.0.1.1/32` in Figure 2.9(a). Knowing the rates of these two groups, we can easily see that the resulting aggregate `224.0.1.0/31` will have bandwidth of 1005 bps. Furthermore, that aggregate will leak 1000 bps on `oif_2` and 5 bps on `oif_1`. Having computed this, we can easily determine whether `224.0.1.0/31` fits within the leak budget on interfaces `oif_1` and `oif_2`. More generally, given n forwarding entries and an aggregate

A that covers those entries, we can determine whether A fits within the leak budget of all of its outgoing interfaces.

This basic building block suggests the following idealized algorithm for computing aggregates:

1. Given N forwarding entries, compute all possible aggregates of these N entries. There are at most $N-1$ of these (the internal nodes of the radix trie [20] built on top of the N forwarding entries).

2. Take these N forwarding entries, and the $N-1$ aggregates. Compute all possible subsets of these $2N-1$ entries. Among those subsets that a) cover the N forwarding entries and b) fit within the leak budgets of all interfaces, choose the subset with the smallest cardinality. Step b) is computed using the technique outlined in the previous paragraph.

Clearly, this algorithm will compute the optimal aggregation. However, considering all possible subsets of $2N-1$ forwarding entries might be compute-intensive simply because the number of subsets is $O(N!)$. Therefore, we choose instead a greedy heuristic that tradesoff optimality for computation time. We now briefly describe this heuristic: detailed pseudo-code for the heuristic can be found in Section 2.8.2.

1. Given the N forwarding entries for which we have estimated the rate, construct the corresponding radix trie. (Recall that these N entries include not just groups rooted at a given root domain, but also the corresponding "holes," Section 2.4.5). The leaves of this radix trie correspond to the N forwarding entries. **Mark** each leaf of the radix trie. Intuitively, a marked entry corresponds to one that will be inserted in

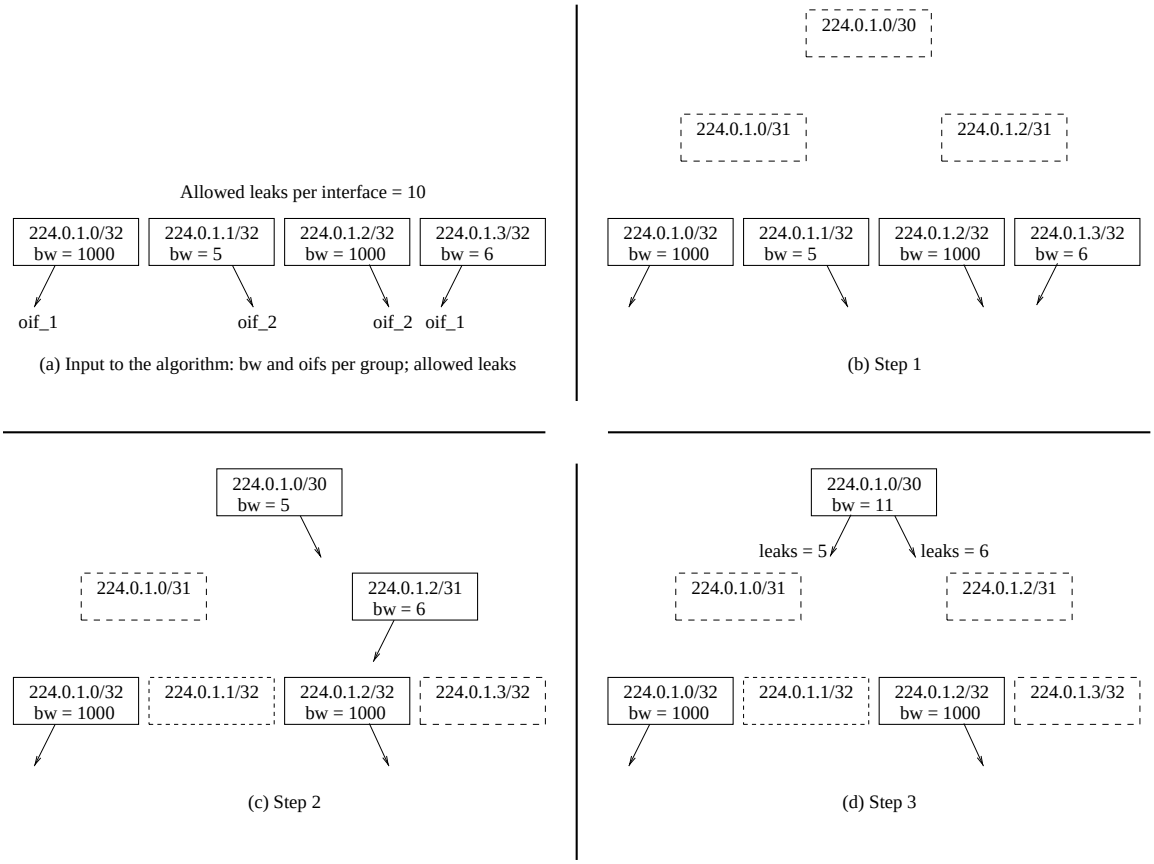


Figure 2.9: An example of the steps of the leaky aggregation algorithm.

the forwarding table. Subsequent steps will attempt to reduce the number of marked entries. Figure 2.9(b) shows the result of this step.

2. Starting from the lowest internal nodes, for each internal node

   (a) Mark the node, and remove the mark on the child of the node with the lower bandwidth.

   (b) Compute the bandwidth attributable to that internal node, and appropriately adjust its outgoing interface set to match the uncovered child.

   This step results in N marked nodes, but some of these nodes represent aggregates of the original N entries. Intuitively, step 2.1 "moves" up the marks on the low bandwidth entries. In Figure 2.9(c), for example, the mark on the lowest bandwidth entry, `224.0.1.1/32`, is replaced with a mark on the aggregate `224.0.1.0/30`. An important observation is that *this step does not introduce any leaks*, it simply changes the matching entry corresponding to a group.

3. For each marked entry A, in order of increasing bandwidth:

   (a) Consider the impact of unmarking A. Unmarking A will cause its nearest marked ancestor to leak more traffic. If this leakage does not violate the leak budget on any interface, we unmark A.

   This key step tries to greedily eliminate entries. For example, in Figure 2.9(d), we were able to remove the mark on `224.0.1.2/31`. This causes traffic to leak on `224.0.1.0/30`, but these leaks are within the budget. At the end of this step,

the remaining marked entries correspond to the aggregated forwarding table entries rooted at the domain under consideration.

The complexity of this heuristic is $O(N \lg N)$. Up to half of the nodes of the radix trie are considered exactly once, and for each node, we may need to traverse the path from that node to the root. Figure 2.9(d) shows the result of running this heuristic on the input shown in Figure 2.9(a). Notice how the heuristic aggregates the low bandwidth groups (of bandwidth 5 and 6 bps) but installs group-specific entries for the "high-bandwidth" entries.

Finally, running the heuristic for 64K entries for a 32-interface router takes about 200 milliseconds on a Pentium II 266 MHz PC. This represents an extremely unlikely worst-case scenario for our heuristic, where the leak budget and entry bandwidths were set so that all 64K entries could be aggregated to a single entry. Finally, because the de-aggregated entries need to be kept for some amount of time (e.g. one second) in the forwarding table to estimate the groups bandwidth, the speed of the algorithm will not be the bottleneck if the de-aggregation and the aggregation are performed on two blocks in parallel.

2.4.7 Dynamics of Groups Join/Leave and Multicast Data Bandwidth

In Section 2.4.4, we did not discuss the impact of receiver dynamics on the leak budget. Consider the following scenario. Suppose group entry G is represented in the forwarding table by its aggregate A. Clearly, A's outgoing interface set includes G's outgoing interface set. Now, if a receiver join results in a new interface added to G's outgoing interface set, then A's outgoing interface set must be updated to include this new interface. But doing so can overrun that some interface's leak budget, because other groups that match A might now start leaking traffic on this new interface. Notice that we can estimate how much this

overrun is, given the most recent bandwidth estimate for G and the estimates for all groups that are covered by A.

Hence, the router needs to make the binary decision: add another oif to A, or install a group-specific entry in the forwarding table. This decision can be based on whether at that moment the amount of leaks or the number are the dominant concern. If it was the first join for the group, the safer solution is to install first a a group-specific entry in the forwarding table, and later the leaky aggregation machinery will consider it for aggregation. Thus, short-living groups might never need be considered for aggregation. Similarly, when a receiver leave results in an interface being removed from G's outgoing interface set, this may reduce the leaks perpetrated by A. This is an opportunity for better aggregation. The join/leave dynamics will result in either adding/removing outgoing interfaces of existing forwarding entries, or installing short-living group-specific forwarding entries that will be eventually aggregated later, and therefore will not trigger immediate re-aggregation of the forwarding table.

	Low bandwidth	High bandwidth
Bursty	Events notification	Mirror update of files
Non-bursty	Heart-beating syncronization or updates	Audio and video streams

Figure 2.10: Examples of different classes of multicast applications: high, low bandwidth, bursty, and non-bursty.

Clearly, leaky aggregation is targeted towards long-lifetime groups; the architectural cost of carefully aggregating short-lifetime groups may not be worth the effort. Figure 2.10

illustrates another source of dynamics that can affect leaky aggregation. Bursty multicast applications (event notification, mirroring software updates) may skew the bandwidth attributable to their aggregates, and hence may affect the leak budget. We are not concerned with short-term increases and do not want to control the leaks with granularity similar to the congestion control. Instead, we need to control the leaks with granularity similar to the activity of the multicast groups. If a high bandwidth group becomes idle, this is an opportunity for better aggregation. On contrary, if an idle group that was eventually aggregated becomes suddenly a high bandwidth group, a group-specific entry should be installed to reduce the potential leaks. By reading periodically the forwarded bandwidth of each entry in the forwarding table, the idle groups that have group specific entry can be easily identified; an aggregated entry that has suddenly increased its aggregated bandwidth should be considered as a source of increased leaks. The block of addresses that has the largest increase of the total forwarded bandwidth should be re-aggregated (i.e. de-aggregated and then aggregated) next to control the amount of leaks. Re-aggregating the block of addresses with the largest decrease of the total forwarded bandwidth has the potentials for better aggregation.

2.5 Simulation Results

We have studied, through simulation, the performance of leaky aggregation and have compared it to the non-leaky methods described in Section 2.4.3. First, we evaluate how the leaky aggregation works among all routers in the network. Then, we evaluate the dynamics of the leaky aggregation by considering a single router only. In this section, we present the results of this study.

2.5.1 Network-wide Simulations

Our first analysis of leaky aggregation was driven by two questions: Compared to the non-leaky schemes, how much additional aggregation does the leaky scheme offer? How sensitive is the leaky scheme to choices of various parameters (the leak budget, the traffic bandwidth mix, and so on)?

The first set of simulations do not consider the impact of dynamic group formation and removal or receiver joins and leaves; this is left to the second set of simulations.

2.5.1.1 Methodology

Our simulations are based on the Mbone topology described in Section 2.3. We also verified that our simulation results were not skewed by our particular choice of topology. To do this, we used automatically generated transit-stub topologies [8], but do not present the results here.

On the input graph, we labeled as *stub* nodes those that were connected to the rest of the network by only one interface. The remaining nodes were labeled as *transit* nodes. In our Mbone topology, there were 2300 stub nodes. This distinction separates leaf routers from interior routers. A subset (32) of the stub routers were randomly chosen to be the root domains for multicast group prefixes. This models our expectation that group origination will primarily be confined to "leaf" networks.

In our simulation, we restricted the multicast address space to 2^{12} (4096 addresses). Because our evaluations are comparative, the size of the address space does not affect our simulations. In the absence of data indicating otherwise, we divided these addresses into 32 equal sized blocks and assigned each block to one root domain.

In each block, we assumed a 75% utilization of the address space. Lower utilizations would have resulted in fewer overall forwarding table entries, a regime where aggregation itself is less important and easier. A higher utilization may be somewhat unrealistic, even when multicast addresses are dynamically allocated [115]. Of the allocated multicast groups, we randomly marked some groups to be *low bandwidth* and some others to be *high bandwidth*. The ratio of the numbers of low and high bandwidth groups, as well as the bandwidth ratios are parameters to our simulations. Clearly, they significantly affect how much bandwidth we can trade off for reduced space.

Lacking data for group size distributions, we chose the receivers for each group randomly among the stub nodes. This random placement of receivers stresses forwarding state aggregation, reducing the likelihood that adjacent groups have identical outgoing interface sets. The number of receivers per group was a parameter to our simulation: this number affects the outgoing interface set of the forwarding entries, and hence the leaks. In our simulations, we varied the number of receivers between 1 and 1500. For a given run of the simulation, every group was allocated the same number of receivers. Finally, we simulated bi-directional distribution trees (a choice of many proposed multicast routing protocols [115, 91, 45]), but the results should hold for uni-directional trees as well. All multicast data was originated either by the root domain for the group, or by some of the receivers. Since few of today's existing applications have them, we did not simulate non-member senders.

How did we select the capacities of links in the network? We could have assigned every link the same capacity, but that would not have stressed the leaky aggregation scheme enough. That is because we define the leak budget to be some fraction of the link

capacity (this fraction is yet another parameter to our simulation). So, if all links are the same capacity, then, even if a link carries little multicast traffic, it may be used to leak a disproportionate amount of multicast traffic. This allows greater aggregation.

To obtain more realistic capacity distribution, we ran our simulation with one set of parameters from the space we intended to explore (in our case, this was 512 high bandwidth groups with bandwidth of 200 units, 2560 low bandwidth groups with bandwidth of 1 unit, and each group had 400 receivers). We performed 100 iterations, and for each iteration we placed at random the root domains and the receivers, and computed the amount of multicast traffic over each link. The capacity of each link was defined as twice the amount of maximum observed multicast traffic on that link during any of our simulation runs (the factor "twice" assumes that the 50% of the network is provisioned for unicast). This heuristic ensured a heterogeneous distribution of link capacities. However, because of the way we generated the link capacities, it sometimes resulted in situations where some routers had attached links that varied by more than an order of magnitude in capacity. To limit this skew and model reality more closely, we adjusted the link capacities so that no router had interfaces whose link capacities varied by more than a factor of 10.

2.5.1.2 Results

Our first set of simulations compared leaky against non-leaky aggregation schemes. In all runs of the simulation, the number of groups was fixed at 3072. Of these, we assumed that 512 were high-bandwidth groups, and 2560 low-bandwidth groups. In the absence of data indicating otherwise, this choice assumes close to an 80-20 split between low and high bandwidth groups. Why this choice? If far less than 80% of the groups were low bandwidth,

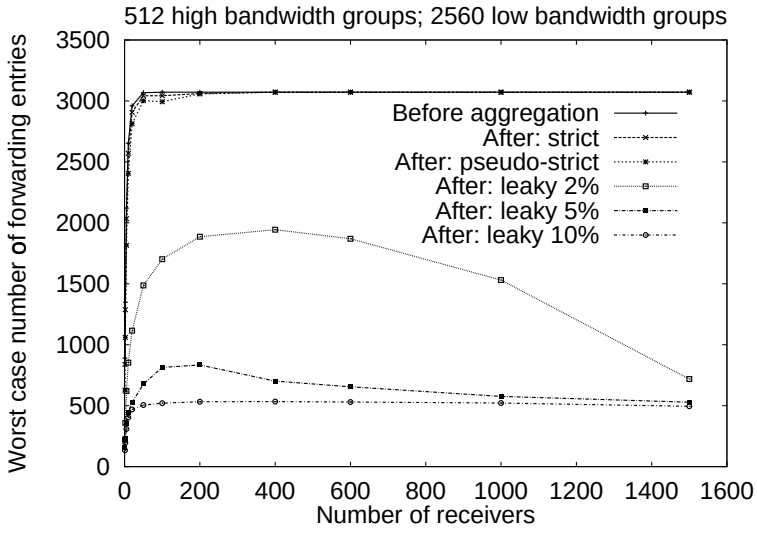


Figure 2.11: Prefix-based strict, pseudo-strict and leaky aggregation.

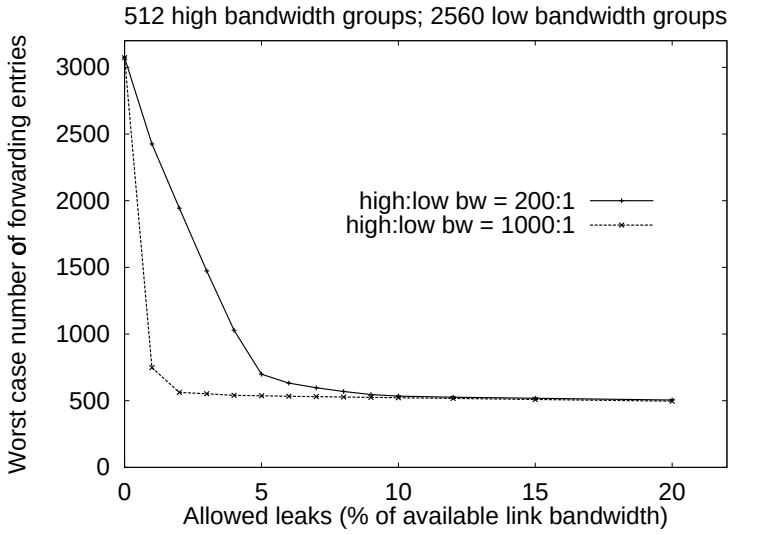


Figure 2.12: Effect of bandwidth ratio on leaky aggregation.

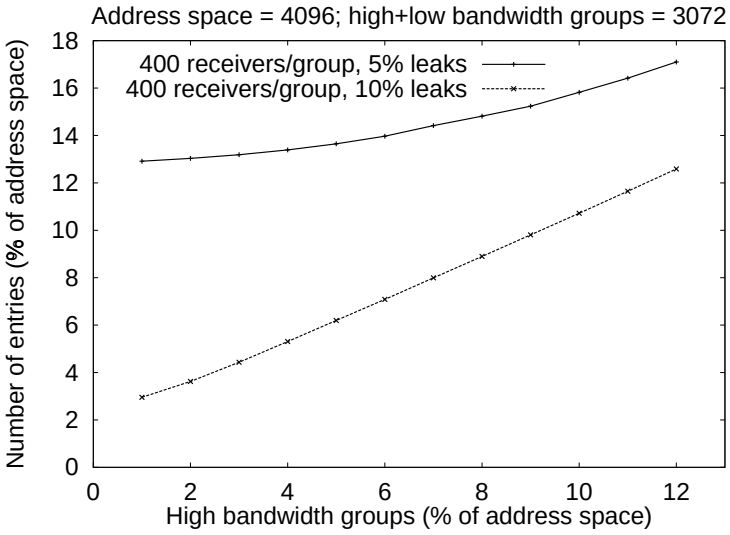


Figure 2.13: Leaky aggregation for different percentage number of high bandwidth groups.

we might be in a regime where aggregation is not an important problem. For example, a split of 50-50 instead of 80-20 will be primarily not because of a larger number of high bandwidth groups (a number naturally limited by available network capacity), but because of the much smaller number of low bandwidth groups. If the fraction of low bandwidth groups was much larger, it might not realistically represent a future internet. We also fixed the high to low bandwidth ratio at 200:1. We believe this to be a very conservative choice: realistically, a high bandwidth video stream may have a bandwidth of hundreds of kilobits per second, while an event notification group may only have a few packets per hour. Below we explore the sensitivity of our results to variations in these ratios.

For these choice of parameters, Figure 2.11 plots the variation of the *worst-case* number of forwarding entries with increasing number of receivers. Clearly, non-leaky techniques do not reduce the worst case forwarding table size. Even with a large number of receivers, there appear to be few instances where adjacent groups' outgoing interface sets overlap.

This can be attributed to our random placement of receivers. If there is indeed a correlation between receivers and the originating root domain (Section 2.2.4), we might expect non-leaky methods to perform well. We might also expect that non-leaky methods perform well with lower router fanouts. In some of our simulations, non-leaky methods reduced the forwarding table by a factor of 2 in routers with 2-4 interfaces.

Figure 2.11 also plots, for three different values of the leak budget, the variation in worst case forwarding table size with number of receivers. Ideally, we expect the worst-case forwarding table size to match the number of high bandwidth groups, 512 in our simulations. Consider the curve for a leak budget of 2%—*i.e.,* where the leaks on every link are limited to 2% of the link's capacity. With our choice of simulation parameters, the low-bandwidth groups account for about 2.5% of the overall bandwidth requirement. So, it comes as no surprise that a 2% leak budget does not match our expected forwarding table size. Despite this, leaky aggregation performs impressively, reducing the worst case table size to less than 60% of the unaggregated size. *Furthermore, with increasing the number of receivers, leaky aggregation approaches our scaling goal.* With a large number of receivers, the likelihood of outgoing interface set overlap increases. In that event, aggregating two entries introduces fewer leaks, so that more aggregates fit into the leak budget.

With a large leak budget, 10%, our heuristic closely matches our expected table size. However, the curve for a 5% leak budget reveals an interesting "hump" for relatively small numbers of receivers. With our choice of parameters, a 5% leak budget should be sufficient to leak all low-bandwidth groups. However, the leak budget is defined as a fraction of link capacity. So, if a router has a low capacity link, 5% of its capacity may not suffice to leak

all low bandwidth groups incident on that router. Such routers account for the hump in the 5% curve.

Figure 2.11 validates our design of the leaky aggregation mechanism, demonstrates its ability to achieve our scaling goal, and highlights its graceful behavior even with a limited leak budget. How would our results differ with different choices for simulation parameters? In particular, our results seem to depend on the total bandwidth requirement of low bandwidth groups, and its relationship to the leak budget.

One way to study this relationship is to vary the bandwidth ratio. We mentioned earlier that our choice of this ratio is conservative. With a more liberal choice (Figure 2.12), we see that our scaling goal is reached for a much smaller leak budget. This is as expected, since the low-bandwidth groups now account for a much smaller fraction of overall multicast bandwidth.[6]

Another way to study this relationship is to vary the *number* of high-bandwidth groups, keeping the total number of groups fixed (Figure 2.13). This figure shows a largely linear growth in the table size as a function of the number of high-bandwidth groups. This rather dramatically illustrates the ability of our heuristic to achieve the scaling goal described in Section 2.4.1. However, with lower numbers of high-bandwidth groups, notice that there is a deviation from the linear. In this region, the leak budget is insufficient to achieve expected aggregation.

[6]For the 1000:1 curve, we recomputed the link capacities, using the technique described in Section 2.5.1.1.

2.5.2 Join/Leave and Multicast Data Traffic Dynamics Simulations

In our second set of simulations we evaluated how well our re-aggregation strategy contains the size of the forwarding table, as well as the volume of traffic leakage, in the face of varying traffic and receiver dynamics.

2.5.2.1 Methodology

We focused on a single router with 32 interfaces. Our simulation methodology was designed to stress the re-aggregation heuristic by employing more than realistic receiver dynamics and widely varying traffic patterns. The multicast address space had total of 128K groups. We defined three types of groups: low, medium, and high bandwidth, with bandwidth ratio of 1:33:1000. Of all groups, 50% were low, 30% were medium, and 10% were high bandwidth; the remaining 10% were considered idle all the time. To define the amount of bandwidth on each interface that is available for multicast, we assumed that the router and its links had the capacity to carry the traffic of 32K groups (i.e. 25% of the address space), when each group had, on average, 8 randomly chosen outgoing interfaces. The leak budget on each interface was fixed at 5% of the maximum multicast traffic on that interface. If the router had the capacity to carry the traffic of more groups, or if the average number of oifs per group was higher, then the absolute amount of allowed leaks on each interface would be higher, and therefore the aggregation results shown below would be better (i.e. the number of entries after the aggregation will be smaller and/or the percentage of leaks will be smaller).

The address space was divided into 32 blocks of addresses of size 4096 addresses each. Each second a block was chosen to be de-aggregated, and the corresponding group specific

entries were installed in the forwarding table (including the prefixes that cover the holes between the groups). The forwarded bandwidth was measured after one second, the entries were aggregated using the algorithm described in Section 2.4.6, and the result aggregated entries were installed in the forwarding table. Then the same process was repeated for another block of addresses, chosen among all blocks that had the largest momentary increase of the forwarded traffic by the corresponding aggregated entries.

We considered three types of bandwidth distribution of a group, which we call Uniform, Exp, and Pareto. For a given interval of time, the bandwidth of a Uniform group had a random value between zero and twice its average bandwidth. Exp and Pareto were ON/OFF type of traffic. The average length of the ON periods was equal to the average length of the OFF periods, and the length of each period had exponential and Pareto distribution respectively. During each ON period, the bandwidth was fixed, and its value also had exponential or Pareto distribution. Similar to [59], the shape parameter of the Pareto distribution was 1.1 for the ON/OFF period and 1.5 for the bandwidth. During a simulation, all traffic had the same distribution type.

In the simulations we chose to install a group-specific entry right after a new group was joined, and later eventually aggregate it; the alternative option was to just add a new interface to the appropriate aggregated entry. The former solution reduces the amount of leaks, but increases the number of entries; the latter is just the opposite. Finally, we should note that we kept track of each group measured bandwidth, and used the following formula to estimate a group bandwidth from the de-aggregated block:

$$EstimBW = MAX(MeasuredBW, 0.7 * EstimBW + 0.3 * MeasuredBW)$$

This formula allows us to capture the bursty bandwidth groups, and at the same time does not underestimate long-term high bandwidth groups if they are idle for a short interval.

In each simulation, we initially populated the router with a number of initial groups (10% of all multicast groups), and on average each group had 2 oifs. After 1000 seconds we caused a number of receivers to join or leave groups every second. The number of receivers joining or leaving was uniformly distributed between 0 and 200; the receivers themselves were randomly selected.

Finally, we also assumed that the router is receiving leaky traffic from its neighbors. The number of low bandwidth leaky groups was 70% of the number of low bandwidth joined groups; the percentage for the middle and high bandwidth leaky groups was 30% and 10% respectively. It is realistic to assume that the higher the bandwidth of a group, lower the probability a neighbor will be leaking it.

2.5.2.2 Results

Given the particular number of groups, their bandwidth and the allowed leaks, we computed that the leaks are enough to aggregate all low bandwidth groups (joined and leaking from the neighbors), and approximately 1000 of the middle bandwidth groups. In the beginning of the simulation there were approximately 6600 low bandwidth groups, 4000 middle bandwidth groups, and 1300 high bandwidth groups, therefore we would expect

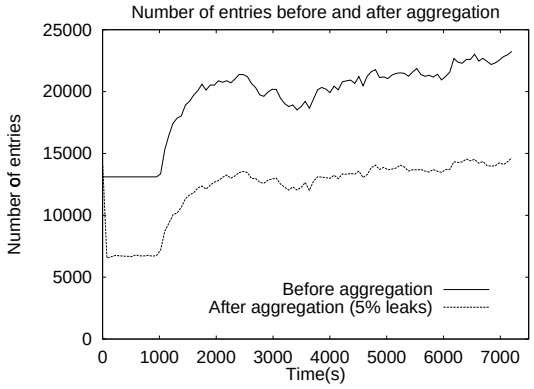


Figure 2.14: Number of entries (Uniform traffic).

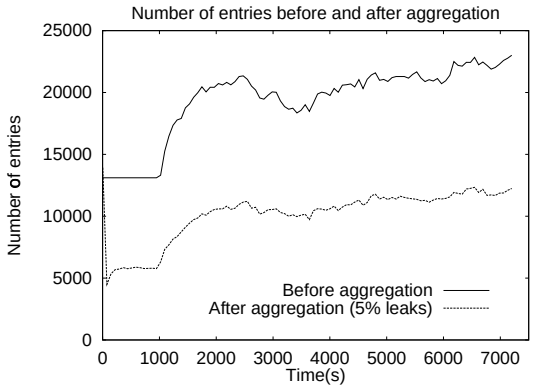


Figure 2.15: Number of entries (Exp ON/OFF traffic).

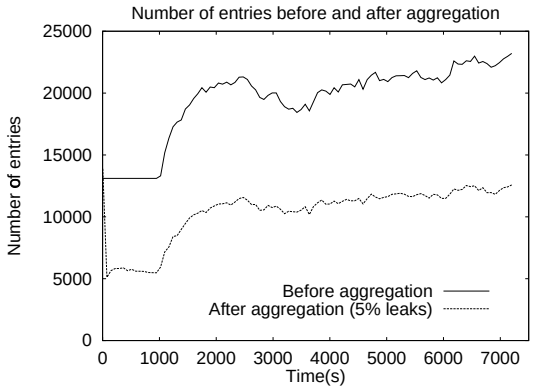


Figure 2.16: Number of entries (Pareto ON/OFF traffic).

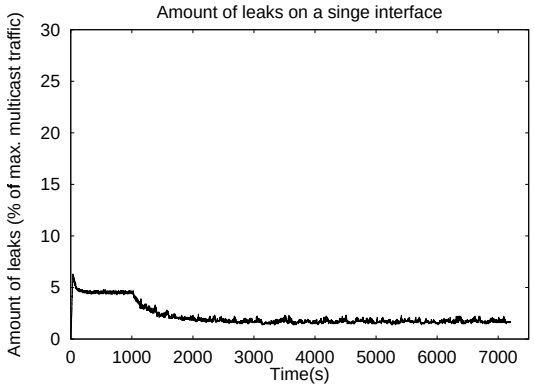


Figure 2.17: Amount of leaks (Uniform traffic).

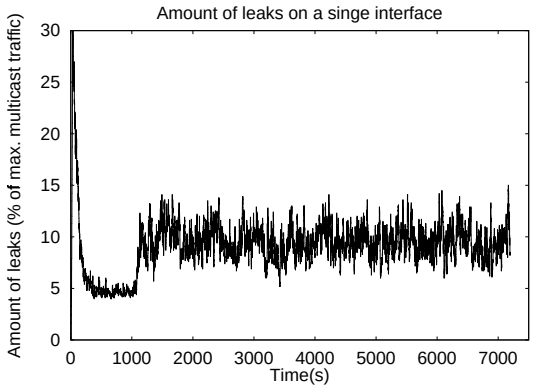


Figure 2.18: Amount of leaks (Exp ON/OFF traffic).

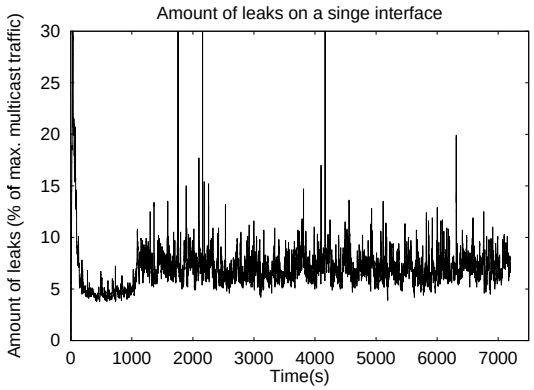


Figure 2.19: Amount of leaks (Pareto ON/OFF traffic).

that the leaky aggregation would save approximately 7600 groups. Similarly, when the total number of groups grows to 22000, we would expect that the leaky aggregation would save approximately 12000 entries.

Figure 2.14, Figure 2.15, and Figure 2.16 show the number of entries in the forwarding table with and without aggregation for Uniform, Exp, and Pareto traffic respectively. The average length of the ON and OFF periods for Exp was 1 minute. After the initial joins at time 0, there were no other joins or leaves during the first 1000 seconds, hence the number of total groups did not change. The reason that the number of total groups increase after time 1000 is because we did not try to keep the average oifs per group to its initial value of 2: over time the average number of oifs became smaller than 2, and the number of joined groups increased. After taking into account that the de-aggregation of a prefix adds up to 2048 more entries (to measure the incoming leaks), we can see that, despite the high variability in traffic, and the significant traffic dynamics, leaky aggregation performs as expected. In particular, the size of the forwarding table does not vary widely. This validates our re-aggregation heuristic 2.4.5.

The reason that the Uniform aggregation was slightly worse than Exp and Pareto was that the formula we used to estimate a group bandwidth was more likely to overestimate the bandwidth of a Uniform group. The observed leaks confirm that. Figure 2.17, Figure 2.18, and Figure 2.19 show the amount of leaks on a single interface (the results for all interfaces were similar). Despite the fact that the leak budget was 5%, because of the group bandwidth overestimation Uniform did not use all leaky budget. Exp and Pareto leaks however were beyond the target leak budget, simply because their momentary bandwidth is less predictable. This is especially true for Pareto which has a much larger variance. We

Modified Parameters	Aggreg (Uni)	Aggreg (Exp)	Aggreg (Par)
32 blocks, 2 oifs, 1 min.	14200/22500	12000/22500	12200/22500
10 min. ON/OFF	—	8800/22500	11000/22500
5 sec. ON/OFF	—	13800/22500	13300/22500
128 blocks	20000/22500	18000/22500	18000/22500
8 blocks	16800/22500	13700/22500	14600/22500
8 oifs	30500/57000	29500/57000	26400/57000

Table 2.2: Simulation results summary: aggregation.

Modified Parameters	Leaks (Uni)	Leaks (Exp)	Leaks (Par)
32 blocks, 2 oifs, 1 min.	1.6–2.0%	7–13%	5–11%
10 min. ON/OFF	—	4–8%	5–11%
5 sec. ON/OFF	—	4–9%	4–10%
128 blocks	1.1–2.0%	7–13%	4–7%
8 blocks	3.0–3.5%	5–11%	4–10%
8 oifs	3–4 %	8–12%	7–15%

Table 2.3: Simulation results summary: leaks.

believe that those middle bandwidth groups that were aggregated account for most of the leaks beyond the leak budget. As future work, we propose to investigate whether a better aggregation heuristic and a better group bandwidth estimation mechanism can reduce this excessive leakage.

We ran the same simulations, but with different ON/OFF mean interval (10 minutes, 5 seconds), different number of blocks used to split the address space (128 blocks, 8 blocks), and with larger initial fanout (8 oifs on average). In Table 2.2 and Table 2.3 we summarize the results. From those results we can make the following observations.

- Increasing the average ON/OFF interval from 1 to 10 minutes improves the aggregation. The reason is that a group with long OFF interval is practically idle, and can be aggregated.

- Decreasing the average ON/OFF interval from 1 minute to 5 seconds did not change significantly the results.

- A much larger number of blocks (128) did not work well. The reason is that, because of the join/leave dynamics, it would take much longer to process all blocks once, and aggregate the low bandwidth groups; similarly, it would take longer time to re-aggregate all blocks and reduce promptly the eventually increased leaks.

- A small number of blocks (8) has a shorter processing cycle, but because each block size now is larger, the number of holes to install to measure the neighbors leaks is larger too, and therefore the total number of forwarding entries is larger.

- Increasing the initial average number of interfaces per group from 2 to 8 increased the number of total groups in the long run, because we did not try to keep the same average number of oifs per group. Despite the larger number of groups, the number of entries after aggregation was as expected.

2.6 Discussion of Leaky Aggregation

We have, so far, postponed the discussion of several interesting questions: What are the limitations of leaky aggregation? Does leaky aggregation have undesirable traffic effects, *e.g.,* loops? Are there other aggregation strategies? Does leaky aggregation address all

multicast routing related scaling issues? The following paragraphs address some of these questions.

Clearly, leaky aggregation cannot achieve the same levels of table compression when low-bandwidth groups dominate overall multicast bandwidth (Figure 2.11). While we have no data to refute this, we believe that, by analogy with TCP traffic [23] on the Internet, a few high-bandwidth multicast applications will dominate the overall multicast bandwidth. Leaky aggregation uses some approximated estimation of the bandwidth of each group, and tries to keep the leaks on each interface within some reasonable limit. However, an extremely bursty traffic might result in larger than expected leaks, and, worse, it might not be always possible to quickly identify the bursty group.

Leaky aggregation can create traffic loops in multicast routing protocols whose forwarding entries are bi-directional. Intuitively, this happens because traffic leaking causes a router to receive traffic for which it never sent a join. Fortunately, this might happen only in some rare cases, which are easy to identify in advance. The solution is simple and needs to be applied only after those cases are identified. Section 2.7 describes the problem and the solution in details with reference to a particular multicast protocol [115].

An alternative approach for multicast group lookup and non-leaky aggregation has been suggested in [116]. This approach reorganizes the forwarding table structure so that for every packet, a per-interface decision is made whether to forward the packet out that interface or not. This alternative organization promises greater non-leaky aggregation. There is more likelihood of adjacent groups sharing one interface on their outgoing interface sets than there is the likelihood of having identical sets. We expect that this alternative structure will equally benefit from leaky aggregation.

Even with leaky aggregation, routers will need to maintain group-specific state in their routing tables. Unlike forwarding state, scaling the routing table may be less critical. First, a router needs to have only a single copy of this table.[7] Second, because routing state is not accessed in the forwarding path, routing tables can be stored in cheaper and slower memory thereby alleviating the problem somewhat. If the amount of routing state becomes an issue, the leaky approach can be applied at the routing table level in a network architecture where the edge routers carefully aggregate the join/prune messages (i.e. the edge routers have the control over the overall network leaks and aggregation). The future will show whether the multicast-capable networks will need this kind of engineering.

In soft-state multicast protocols such as PIM-SM [33], control traffic can increase linearly with the number of concurrent groups. Leaky aggregation does not solve this problem. Approaches that adapt the refresh rate of explicit joins [106] have been proposed to deal with this.

In theory, leaky aggregation is applicable to source-group-specific forwarding entries as well. Because source unicast addresses are already topology assigned, our leaky aggregation strategy will work on prefixes which consider the source and the group address together. It is less clear if there is a problem to solve here: increasingly, inter-domain multicast protocols [115, 91] instantiate mostly group-specific state. Source-based leaky aggregation can, however, be applied to broadcast-and-prune multicast routing protocols such as DVMRP [94] and PIM-DM [26], when the number of low-bandwidth entries is too large.[8]

[7] High-end routers, such as [80] and [89], use a number of routing engines with a copy of the forwarding table at each engine.

[8] We should note that if a PIM-DM or DVMRP router receives leaky (i.e. unwanted) traffic, typically it would send a Prune message to the upstream router. Usually "forwarding table cache miss" is used to

Finally, the meta-question that someone might ask is: do we need aggregation at all, and if we do, would not be possible to achieve the desired result with strict aggregation only? If most of the time the forwarding table size was within the capacity of the forwarding table memory, but occasionally the forwarding table would overflow, it is quite likely that the router performance will be affected during the overflow. Taking the router off-line for hardware upgrade might not be the most appropriate or flexible solution. Automatically triggering leaky aggregation before the overflow of the forwarding table will eliminate the need for unnecessary hardware upgrade, and will reduce the management cost. Indeed, it is possible that using pseudo-strict or strict aggregation might be good enough, and we might not need leaky aggregation at all. In fact, the leaky aggregation algorithm we described in Section 2.4.6 (see Section 2.8.2 for the pseudo-code) could actually be used for pseudo-strict aggregation when the allowed leaks per interface are zero, or even for strict aggregation (after a simple modification of the aggregation rule), simply because strict and pseudo-strict aggregations are more restricted versions of leaky aggregation. If the non-leaky aggregation was not good enough, then the leaky aggregation might be used as a last resource.

2.7 BGMP-specific Loop Problem and its Prevention

For protocols that use bi-directional trees traffic leaks can result in loops. Figure 2.20 shows such a loop in BGMP [115]. Router A installs an aggregate forwarding entry for `224.1/16`.

identify the unwanted group traffic and trigger the Prune. If there was a leaky entry in the downstream router too, then it will not send a Prune message. If the downstream router did not have any entry for that unwanted group (including an entry with oifs=NULL), the rate of the Prune messages it sends might be used by the upstream router for quick identification of the high bandwidth groups that should not leak anymore on that interface.

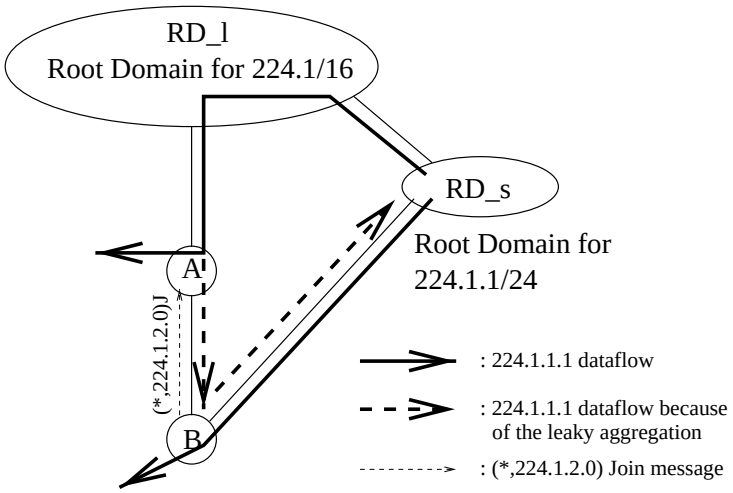


Figure 2.20: BGMP-specific loop potential problem because of the leaky entries aggregation.

This entry is leaky—that is, the A–B interface is included in the entry's outgoing interface list *even though B has only sent an explicit join for one group* `224.1.2.0` *in that prefix.* B installs a forwarding entry for `224.1.1/24`. Traffic for any group in this prefix can reach B from two directions: RD_s–B and RD_s–RD_l–A–B. The BGMP forwarding rules in B will accept the data over link A–B and will forward it toward RD_s. The data will be forwarded by RD_s to RD_l, then A down to B again, *i.e.*, the data will start looping.

The reason for the loop problem is that B has Group Routing Information Base (G-RIB) for two overlapping prefixes (`224.1/16` and `224.1.1/24`) with different next-hop routers toward the corresponding Root Domains. Furthermore, both A and B implement leaky aggregation. Based on the G-RIB information B has, and the join messages that were sent to the neighbor routers, B can detect potential loops for some group prefixes (`224.1.1/24` in our example). One possible solution is B to instruct A, via BGMP's optional attribute negotiation mechanism, not to send any leaky traffic for `224.1.1/24` over link A–B. A

can prevent the leaks by simply installing a single entry for `224.1.1/24`, and this entry would never be passed to the leaky aggregation machinery. We do expect that mostly the routers that are close to the edges (i.e. routers with less entries to aggregate) will have such overlapping and pointing in different directions G-RIB entries, but even if it is quite common among backbone routers, in the worst case a router will need to install a number of entries (each entry will represent a prefix size of at least 256 groups), and those entries would never be deleted. This solution retains the incremental deployability of BGMP. A similar solution may be applied for CBT [4].

2.8 Pseudo Code for Leaky Aggregation

2.8.1 Leaky Aggregation Main Procedure

The main procedure used to perform leaky aggregation can be implemented in software, and does not require any explicit hardware support.

```
while (TRUE) {
  /* Select next block to de-aggregate,
   * measure its groups bandwidth, and aggregate.
   */
  /* Check first if some prefix is a better candidate:
   * it may be creating too much bandwidth overhead, or
   * may have better aggregatability.
   */
  curr_block = HEAD(priority_queue);
  if (curr_block == NULL)
```

```
    curr_block = HEAD(round_robin_queue);

  /* De-aggregate. Install in the forwarding table entries
   * for all groups and for the holes between them.
   */
  de_aggregate(curr_block);

  /* Continue below after some amount of time */
  goto_label_later(READ_BW, T);

READ_BW:
  read_entries_bw(curr_block);  /* Read estimated bandwidth */

  /* Compute the entries. See the pseudo code below */
  compute_leaky_entries(curr_block);

  /* Install in the forwarding table the computed
   * aggregated entries
   */
  install_leaky_entries(curr_block);
}
```

The management of the priority queue can be performed by the same process before the execution continues from label "READ_BW."

2.8.2 Greedy Heuristic Aggregation Algorithm

Below is the pseudo-code of the greedy approximation algorithm we used to compute the leaky aggregated entries in our simulations. The algorithm time complexity is $O(N \lg N)$; the space complexity is $O(N)$.

```
compute_leaky_entries(prefix_address_block)
{
  /* Assign the allowed leaks per interface for this address block */
  for (all_ifs)
    allowed_leaks[if_id] = leaks_budget(if_id, prefix_address_block);

  /* For each group, mark a node in the Radix trie as INSTALL */
  /* Lower the bandwidth, the corresponding node will
   * be higher in the trie.
   */
  for (parent = addr_block_size-1; parent >= 1; parent--) {
    left = LEFT_CHILD(parent); right = RIGHT_CHILD(parent);
    if (ENTRY(left).bw <= ENTRY(right).bw)
      low = left; high = right;
    else
      low = right; high = left;
    ENTRY(high).flag |= INSTALL;
    ENTRY(parent).bw = ENTRY(low).bw;
    for (all_ifs)
      ENTRY(parent).join[if_id] = ENTRY(low).join[if_id];
  }
```

```
  /* Sort all entries except ROOT */
#define ROOT 1
  ENTRY(ROOT).flag &= ~INSTALL;
  /* Can use approximated hash buckets to sort in O(N),
   * or any standard O(N*lg(N)) sorting algorithm.
   */
  SORT_ALL_INSTALLED_ENTRIES_BY_THEIR_BW();
  remaining_nodes = total_installed_entries;

  /* Always install the root of the prefix (the default
   * entry toward the Root Domain for that prefix).
   */
  ENTRY(ROOT).flag |= INSTALL;

  while(remaining_nodes--) {
    node = POP_SMALLEST_BW_ENTRY();
    /* Find first installed parent node up in the Radix trie */
    parent = FIND_INSTALLED_PARENT(node);          /* O(lg(N)) */
    for (all_ifs) {
      bw_overhead[if_id] =
          compute_leak_after_aggreg(parent, node, if_id);
      if (bw_overhead[if_id] > allowed_leaks[if_id]) {
        keep_entry = TRUE;
        break;
      }
```

```
    }
    if (keep_entry == TRUE)
      continue;
    ENTRY(node).flag &= ~INSTALL;
    for (all_ifs) {
      allowed_leaks[if_id] -=
          compute_leak_after_aggreg(parent, node, if_id);
      ENTRY(parent).join[if_id] |= ENTRY(node).join[if_id];
    }
    ENTRY(parent).bw += ENTRY(node).bw;
    REINSERT_INTO_SORT_LIST(parent);    /* O(lg(N)) or           */
  }                                     /* O(1) for hash buckets */
}
```

2.9 Conclusions

In this chapter we have studied the topology impact on multicast forwarding state aggregatability. In our study we use a variety of topologies (real-world, generated and canonical). The results show that the underlying topology can have notable impact on performance. For example, for low receiver occupancy, the aggregation ratio for the Internet core topology can be on the order of 10 times, for random graphs the aggregation can be on the order of 20 times, but for tree, mesh and some generated topologies it is on the order of 2 times.

Aggregation ratio on the order of 10 or 20 times may seems high, but for this particular problem it may not be sufficient if multicast services become widely popular and there are tens or hundreds of million active groups at a time; further, designing practical algorithms

to achieve this level of aggregation may be challenging. In the second part of the chapter we describe a *leaky aggregation* that can be used to reduce further the amount of state. This is achieved by allowing some imperfection of the state aggregation: we allow a small, controllable amount of data traffic to "leak" on interfaces without downstream members. By carefully aggregating the entries, we can reduce the amount of forwarding state in the routers to the order of the number of high bandwidth groups whose data traffic is forwarded by that router. Therefore, by trading-off small network bandwidth overhead for memory in the forwarding engines, we can achieve higher level of aggregation.

In the next chapter we study another problem: efficient replica placement for Content Distribution Networks. However, instead of using a very radical solution such as the leaky aggregation, we consider using partial knowledge about the network topology to improve protocol performance.

Chapter 3

Case Study: Replica Placement for Content Distribution Networks

In this chapter we study efficient replica placement for Content Distribution Networks. In general, this is an NP-complete problem, and existing approximation algorithms are either computationally expensive, or require detailed knowledge about the underlying network topology and expected client location. We consider a very simple solution to the problem that uses limited amount of information about the underlying topology. Our solution places replicas among nodes with the largest fanout. Surprisingly, the performance of this solution is within a factor of 1.1–1.2 of more complicated solutions that have been shown to perform within a factor of 1.1–1.5 of the optimal solution. We study the problem for a variety of topologies and client placement, and we discover that in most cases the results are qualitatively similar. These results demonstrate that sometimes a small amount of information about the underlying topology can be useful to improve significantly the protocol performance.

3.1 Introduction

Content Distribution Networks (CDNs) [2, 53, 29] replicate Web content in an effort to reduce client access latency. This kind of replication can also reduce network overhead. However, the efficacy of content distribution can crucially depend on the placement of these replicas, and on the relative location of the client population.

In the past, there have been several studies that have addressed the problem of replica placement on the network and its impact on network performance [65, 95, 60, 70]. A number of replica placement methods have been proposed and studied. Two of the studies [95, 60] have considered a greedy placement strategy[1] which, compared to the computationally expensive optimal solution, performs remarkably well in practice (within a factor of 1.1–1.5), and is relatively insensitive to imperfect input data. Unfortunately, this greedy placement requires knowledge about the client locations in the network, and all pairwise inter-node distances, which information in many cases may not be available.

One of the previous studies [60] considers also topology-informed replica placement, where nodes are selected as replicas in decreasing order of their node degree.[2] Their results suggest that this method can perform almost as good as the greedy placement. However, due to lack of more detailed network topology, this particular study uses only Autonomous Systems (AS) topologies (real-world and generated) where each node represents a single AS, and a node link corresponds to AS-level BGP peering.

[1] The particular greedy placement is also very similar to the one in [65].

[2] From now on we will use interchangeably the terms *node degree* and *node fanout* to represent the number of links connecting a node with its neighbors. Also, we will use the term *well-connected node* to indicate a node that has a large fanout.

In this work we extend their evaluation of fanout-based replica placement in several ways. First, instead of using a coarse-grained AS topology derived from BGP AS paths information [122], we have in our possession an approximate router-level Internet topology [38] which we use to obtain more detailed and accurate results. Second, instead of shortest-path routing, we generate router-level paths using approximate models of inter-AS routing policy [113]. With their technique, each router from the router-level topology is mapped to the AS it belongs to (based on that router IP address), and then AS-level shortest-path routing is combined with router-level shortest-path towards the next-hop AS. Finally, we look into results sensitivity by considering various client placement models, and some other topologies.

Our main findings are:

- In most cases the router-level fanout placement is almost as good as the greedy placement (within a factor of 1.1–1.2).
- A fanout-based replica placement method needs to be carefully designed to be efficient. For example, if we select first a well-connected AS and then we select a router within that AS, we must be very careful which particular router is selected.

Our conclusions do not depend on client locations. Only if the number of clients is very small, then there is a significant performance difference between the fanout-based replica placement and the greedy placement. The results are true also for random graphs, generated and real-world AS topologies, but do not apply for overlay topologies such as Mbone [75].

The rest of the chapter is organized as follows. In Section 3.2 we describe the particular replica and client placement models we consider in this work. Section 3.3 contains the performance evaluation results. Section 3.4 presents a possible explanation when and why those results may hold. Conclusions are in Section 3.5.

3.2 Replica and Client Placement Models

In this section we describe the replica placement models we are interested at, and are evaluated later in Section 3.3. We use each of those models to place a number of replicas on the topology, so we could eventually reduce the client access latency and the overall network overhead (compared to a single-server solution). We also describe the client placement models that we use to select a number of nodes as clients. Those models are used in Section 3.3.3.2 to perform the client-impact sensitivity evaluation. Before presenting the replica and client placement models, we describe the client-replica assignment we assume.

3.2.1 Client-Replica Assignment

In this work we assume that each client selects the closest (in number of hops) replica. Indeed, it is possible to consider a more sophisticated scheme where a client selects in real-time the replica that offers the lowest latency, but for simplicity we ignore such schemes. The second assumption we make is that we do not limit the number of a clients that can be assigned to a replica. Both assumptions are similar to those in some of the previous work [95, 65, 60]. One of the arguments to support the latter assumption is that typically it is much easier to increase the capacity of a particular replica (*e.g.,* by creating a cluster of replicas at the same location), than deploying a new replica at different location for the sake

of reducing other replicas' load. The latter assumption does not impact our conclusions even in the presence of flash crowds, because, as we demonstrate later in the chapter, our results are robust to variations in client locations and client population size.

3.2.2 Replica Placement Models

In this work we consider the following replica placement methods. The first method has been proposed in some of the previous work [65, 95, 60, 70]. In our study we use it as a base for comparison.

- *Greedy placement.* The greedy placement we choose is same as the greedy algorithm described in [95] and [60]. The basic idea is to choose the replicas one-by-one, a subject to a greedy selection: at each step we evaluate all nodes in the topology and choose the one that, if we place a replica there, the resulting network overhead will be minimized. The process is repeated until all replicas have been chosen. The input to this method is all pairwise inter-node distances, and the client placement locations.

- *Max-router fanout placement.* Given a network topology and the fanout of each node, we choose the replicas one-by-one in decreasing order of their node degree until all replicas have been chosen. The intuition behind this method is that the nodes with large fanout are eventually the closest (on average) to all other nodes, and therefore they are a better choice for replica location.

- *Max-AS/max-router fanout placement.* This method assumes that each node/router has been assigned to some AS, and that all ASs have been connected into an AS-level topology. If R is the number of replicas to select, first we select the R ASs that have

the largest fanout (on the AS-level topology). Then, within each selected AS, we choose the router that has the largest router-level fanout. Similar to the *max-router fanout* placement, the intuition is that the selected nodes will be closer to the rest of the nodes.

- *Max-AS/min-router fanout placement.* This method is similar to the *max-AS/max-router fanout* placement, except that instead of selecting the router with largest fanout within each of the chosen ASs, we select the router with the smallest fanout. This placement may not make sense for practical purposes, but we need to consider it to evaluate the sensitivity of network performance to replica placement within an AS. Note that for the rest of this chapter, when we use the term *fanout-based placement*, we do not include the *max-AS/min-router* fanout placement, unless stated otherwise.

- *Random placement.* In this method the replicas are chosen at random with uniform probability among all nodes in the topology. We consider it as an "upper-bound" placement method in a sense that an efficient replica placement method should always be better than the random placement.

Unlike previous work [95, 65, 60], we do not consider some of the existing optimal solutions that have been proven to be always within a small factor of the most optimal solution.

Indeed, Qiu et al. [95] have found that the particular greedy algorithm described above performs very well in practice (typically within a factor of 1.1–1.5 of the computationally intensive optimal solutions). Further, its performance is relatively insensitive to imperfect

input data such as client locations and network topology information. Therefore it is a reasonable choice for our needs and we can use it as a base of comparison.

3.2.3 Client Placement Models

To investigate the sensitivity of replica placement performance to client locations, we look into several client placement models. Our goal is not to explore all possible client placements, but to consider the extreme cases, along with the random case, because the extreme cases can give us the boundary of expected performance.

The first model we look into is the *random client placement*, where the client nodes are selected at random with uniform probability.

We also look into the extreme client placement as defined in [92], namely *extreme affinity* and *extreme disaffinity*. The extreme affinity model places the clients as close as possible to each other; the extreme disaffinity model places the clients as far as possible from each other. The particular algorithm we use to place a number of clients on a graph according to the affinity/disaffinity model is described in [130]. Below is a brief summary of that algorithm. The first client is selected at random among all nodes. Then, we assign to each node n_i that is not selected yet the probability $p_i = \frac{\alpha}{w_i^\beta}$, where w_i is the closest distance between node n_i and a node that is already selected as a client, α is calculated such that $\sum_{n_i} p_i = 1$, and β is the parameter that defines the degree of affinity or disaffinity. After a node is chosen to be a client, the probabilities of the remaining nodes are recomputed and the process is repeated until the desired number of clients is selected. Similar to [130], in our experiments we use $\beta = 15$ and $\beta = -15$ for extreme affinity and disaffinity respectively.

We look into yet another extreme client placement: *extreme clustering*. This placement can be considered as a hybrid between extreme affinity and extreme disaffinity. With this placement, clients are "grouped" into a number of *clusters*, such that the clients that belong to the same cluster are as close as possible to each other (*i.e.*, extreme affinity placement). Then, all clusters are placed as far as possible from each other (*i.e.*, extreme disaffinity placement). A two-step version of the extreme affinity/extreme disaffinity algorithm described above can be used to create the extreme clustering as well. In the first step we place $\sqrt{C}$ clients with extreme disaffinity with parameter $-\beta$, where C is the total number of clients. Each of those clients are considered as a center of a cluster of size $\sqrt{C}$ clients. In the second step, we add $C-1$ clients to each cluster by using the extreme affinity algorithm with parameter β.

To verify our results with real-world data, we use Web server access logs to create the population of clients. In particular, we collect the unique IP addresses of all clients that have accessed the same Web server within some period of time. Then, we run the *traceroute* tool [55] to each of the client addresses. Finally, we intersect each of the traceroute paths with the Internet map to find the last-hop router toward a Web client that is on that map. The set of all last-hop routers is our *web clients set* that can be used to represent the population of the real-world Web clients.

3.3 Performance Evaluation

In this section we present the main results from our evaluation. In particular, we use numerical simulations to compute the relative network performance. As part of our evaluation we look into the impact of various factors on performance: replica and client placement,

client number, network topology. First we describe the metric space, and then we present the results when we vary each of the input factors we consider.

3.3.1 Metric Space

The two particular metrics we are interested at are *average client latency* and *overall network overhead*. For simplicity, we assume that the latency between two nodes is proportional to the number of link-hops between them. A similar assumption has been used in a previous work [95]. Indeed, [41] shows that router-level hops correlate well with observed latency. On the other hand, that work points out that the number of ASs in the path to a destination has a higher correlation to latency than the number of router-level hops. However, that study is several years old and Internet has evolved since then. Further, a more recent study has measured 50–70% correlation between network hop and round-trip time [86], and its authors claim that the router-level number of hops is more meaningful as a latency metric.[3] Finally, due to lack of information, we assume that the bandwidth capacity of all links is same. Obviously, those assumptions are not perfect, but without detailed network measurements this is the best we can do. Hence, the average client latency across all clients c can be computed as:

$$AveClientLatency = \frac{\sum_{clients(c)} Dist(c, Replica(c))}{NumberOfClients}$$

where $Replica(c)$ is the replica node for client c, and $Dist(c, Replica(c))$ is the distance between them in number of hops.

[3]We also show, in Section 3.3.3.4, that if we consider latency defined in terms of AS-level hops, our findings hold even stronger.

For similar reasons as above, we also assume that the overall network overhead is proportional to the number of link-hops used to disseminate the data from the replicas to all clients. At the same time, we ignore the network overhead to distribute the data from its original location to each of the replicas, because this overhead may be a small fraction of the network overhead to distribute the data from the replicas to a large number of clients. Hence, the overall network overhead for all clients can be computed using the following formula:

$$NetworkOverhead = \sum_{clients(c)} Dist(c, Replica(c))$$

In our evaluation, we are not interested in the absolute client latency or absolute network overhead metrics. Instead, we are interested in the *relative client latency* or *relative network overhead* of each replica placement method versus the greedy placement. Based on our assumptions, we have $AveClientLatency = \frac{NetworkOverhead}{NumberOfClients}$, therefore it is easy to see that when we perform *relative* comparison between two replica placement methods using the same set of clients, then the relative average client latency will be same as the relative network overhead. Therefore, for the rest of this chapter we use a single metric we call *efficiency ratio* to compute the relative performance between two replica placement methods. We always use the greedy placement as a base for comparison, hence the efficiency ratio of method M_i can be computed as:

$$EffRatio(M_i) = \frac{NetworkOverhead(M_i)}{NetworkOverhead(Greedy)}$$

3.3.2 Simulation Setup

For most of our simulations (except for those described in Section 3.3.3.4 where we look at the topology impact factor), we use a real-world router-level topology. The topology information was collected by using a large number of traceroute requests sent over the Internet [38, 123]. The resulting topology had 102639 nodes and 142303 links. Then we recursively removed all nodes that have a fanout of one to obtain a topology we call *Internet core*. The reason that we truncate the original topology is to remove the long, "skinny" branches that do not represent well the network connectivity at the edges, but are an artifact from the particular methodology used to obtain the topology information.

To obtain more realistic results, instead of using shortest-path routing, we use AS-level hierarchical routing as described in [113]. With their technique, first each router from the router-level topology is mapped to the AS it belongs to, based on that router IP address and the AS-level topology [12] at the time the router-level topology data was collected. After that, the AS-level shortest-path routing is computed. Finally, to compute the router-level path between two nodes, the AS-level path is followed, and within each AS the router-level shortest path is used to reach the closest node that belongs to the next-hop AS.

Topology	Nodes	Links	Diam.	Ave. dist.	Ave. fanout
Internet core	27646	67310	26	8.3	4.9
Random graph	19596	40094	16	7.2	4.1
Power-law graph	10091	23253	9	3.2	4.6
AS	4830	9077	11	3.7	3.8
Mbone	4179	8549	26	10.1	4.1

Table 3.1: Metrics of used topologies.

Within each set of simulations we fix the number of clients and vary the number of replicas, or vice-versa. The number of replicas varies between 1 and 50; the client population size varies as a fraction of the number of all nodes between 0.005 and 0.2. The replica and the client placement methods, as described in Section 3.2, are the other input to the simulations. In all simulations we use 100 different sets of algorithmically chosen clients (except for the Web-derived clients when we have 3 sets), and we average the results among all trials. The results we show are for the 95% confidence interval (note that in most cases this interval is very small and can be seen as a single dot).[4] To create the set of Web-derived clients, we use the access logs of a busy Web server for three consecutive days, and we apply the technique described in Section 3.2.3 to compute the nodes on the router-level topology that represent the Web clients. The number of unique client addresses for each of the three days is 37401, 40833, 43558 respectively. Those clients, after the intersection of the traceroute paths with the router-level topology, are represented by 4015, 4158 and 4264 unique nodes on the Internet-core map (approximately 15% of all nodes).

We tried also some generated topologies, and some other real-world maps (see Section 3.3.3.4). Table 3.1 summarizes some of the metrics of all topologies.

3.3.3 Network Efficiency Results

First, we present the results for different replica placement methods, which are of most interest to us. Then we look at how the client placement may have impact on performance. Finally, we look into other factors such as client number and network topology.

[4]We looked also into the min-max interval, and it was almost unnoticeable for the *max-router fanout* and *max-AS/max-router* placement methods.

3.3.3.1 Replica Placement Impact

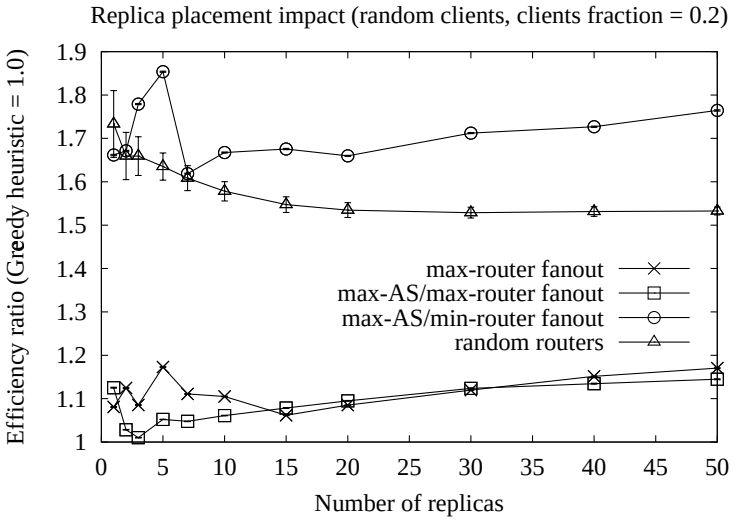


Figure 3.1: Internet core: replica placement impact (random clients).

To evaluate the replica placement impact, we assume a fixed number of randomly placed clients (20% of all nodes), and the number of replicas varies between 1 and 50. Then we compute the relative network efficiency for different replica placement methods (as described in Section 3.2.2), by using the greedy algorithm results as base for comparison (1.0).

The results from this simulation are in Figure 3.1. The first observation we can make is that both *max-router fanout* and *max-AS/max-router fanout* placement methods perform very well, within a factor of 1.1–1.2 of the greedy placement, regardless of the number of replicas. This result is our first confirmation that the fanout-based placement methods perform well even on Internet router-level topology. On the other hand, the *max-AS/min-router fanout* placement performs even worse than random replica placement. This, to some extent, is a surprising result, because we expected that the AS fanout is the major

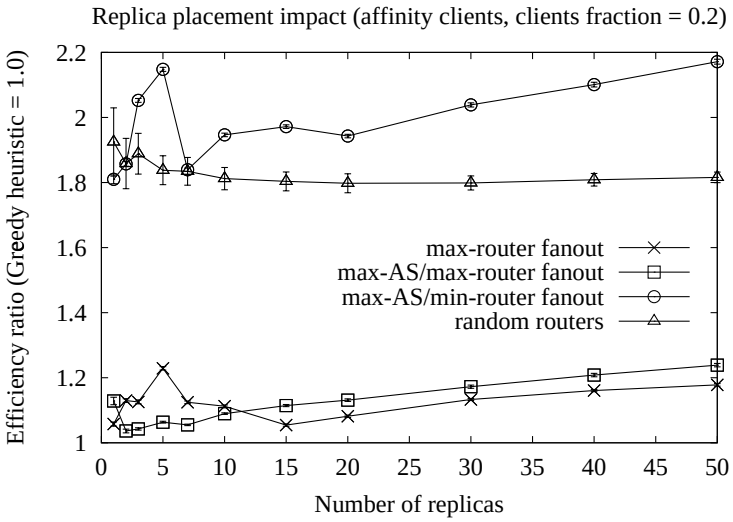


Figure 3.2: Internet core: replica placement impact (extreme affinity clients).

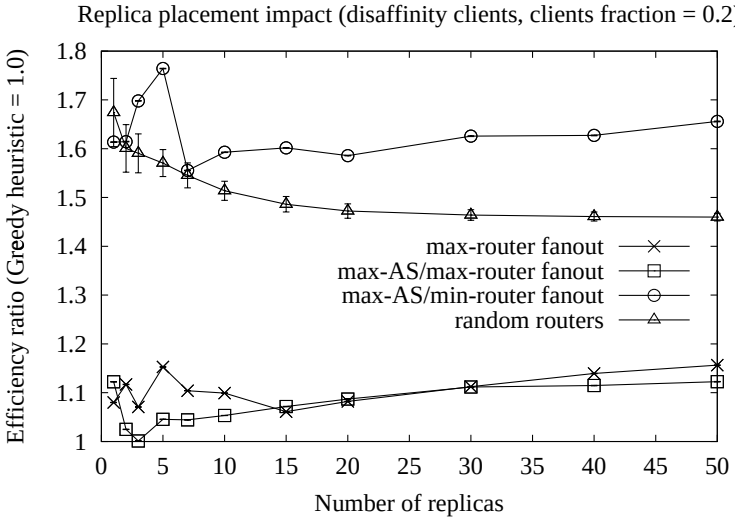


Figure 3.3: Internet core: replica placement impact (extreme disaffinity clients).

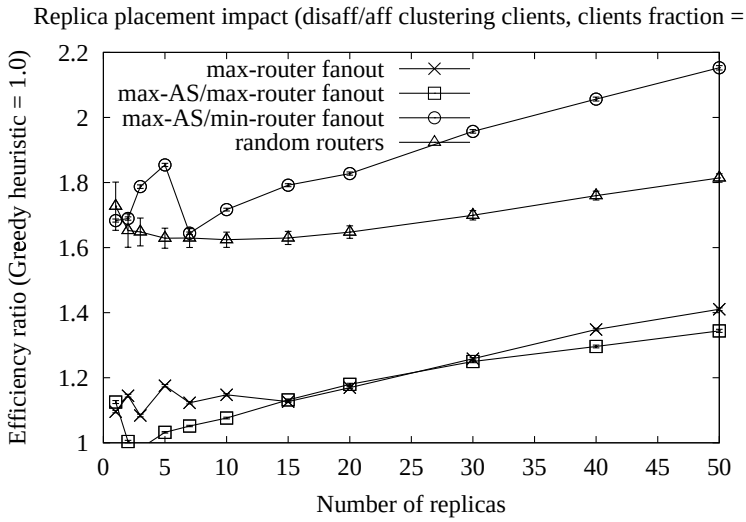


Figure 3.4: Internet core: replica placement impact (extreme clustering clients).

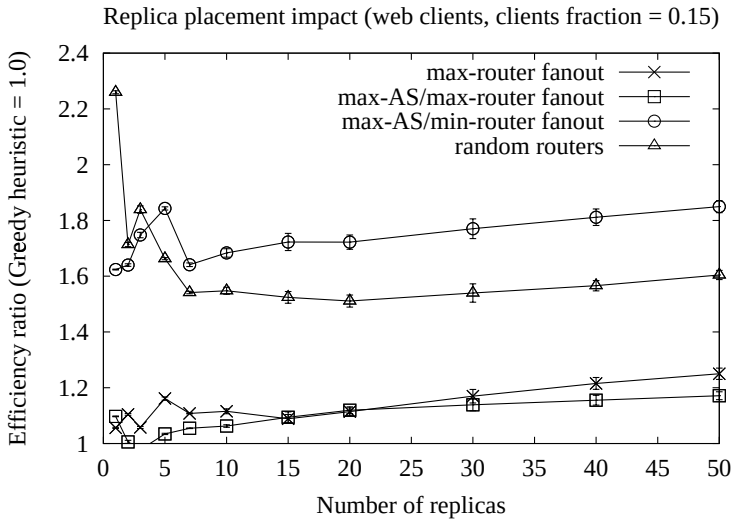


Figure 3.5: Internet core: replica placement impact (web clients).

factor that has impact on performance. Clearly, placement is very sensitive to the actual selection of routers within ASs–selecting the highest fanout AS alone for placing a replica is not sufficient.

3.3.3.2 Client Placement Impact

To evaluate the client placement impact on the results, first we consider the extreme cases of *affinity*, *disaffinity* and *clustering*. The particular model we use was described already in Section 3.2.3. Figure 3.2, Figure 3.3 and Figure 3.4 show the results for extreme affinity, extreme disaffinity and extreme clustering respectively (the rest of the setup is same as in the case of random client placement in Section 3.3.3.1). Here again we can see that in case of extreme affinity and extreme disaffinity client placement, the *max-router* and *max-AS/max-router* fanout-based placement methods perform remarkably well within a factor of 1.1–1.2 of the greedy placement. However, the results for extreme clustering reveal notable difference in performance: the fanout-based replica placement method can be on the order of 1.4 worse compared to the greedy placement. Even though, quantitively, the difference is not significant, we consider this finding an important evidence of the impact of client placement may have on performance.

The results with the *web-clients* are in Figure 3.5. Similar to the extreme affinity and extreme disaffinity client placement, with web-clients the fanout-based placement methods perform equally well.

It is interesting to note that, unlike the greedy placement, the fanout-based replica placement methods do not take client locations into account, yet they can perform very well over a wide range of client placements (including realistic placements).

3.3.3.3 Client Number Impact

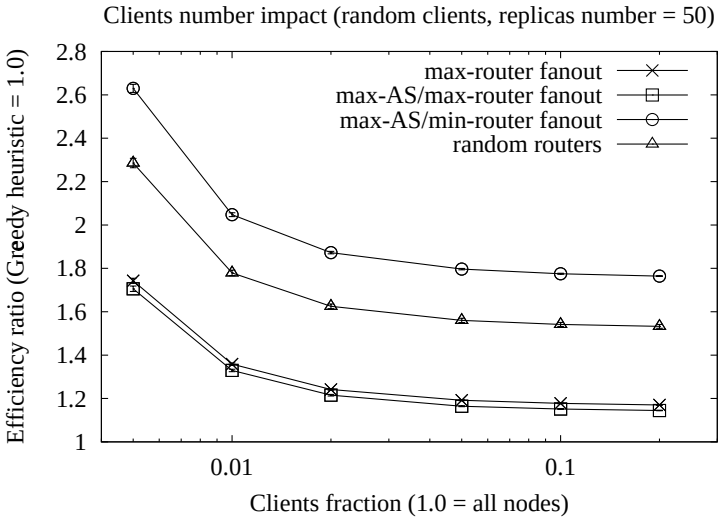


Figure 3.6: Internet core: client number impact (random clients).

The next question we want to answer is how the client population size impacts the performance. In this set of simulations we fix the number of replicas to 50, and then we vary the fraction of nodes that are clients in the range 0.005–0.2. Figure 3.6, Figure 3.7, Figure 3.8, and Figure 3.9 show the results for various client placement: random, extreme affinity, extreme disaffinity, and extreme clustering respectively.

We observe that when the number of clients is small, the fanout-based placement methods do not perform very well. This is especially true for extreme affinity and extreme clustering client placement. For moderate and large client number, the fanout-based placements perform much better, as expected.[5] Another observation we can make is that the

[5]The *random* and *max-AS/min-routers* placement performance for extreme disaffinity of the clients as a function of the client population size may seems a little bit unusual because it is not monotonically increasing or decreasing. This behavior can be explained by the fact that the results can be influenced significantly by various factors if the number of clients is very small (on the order of the number of replicas).

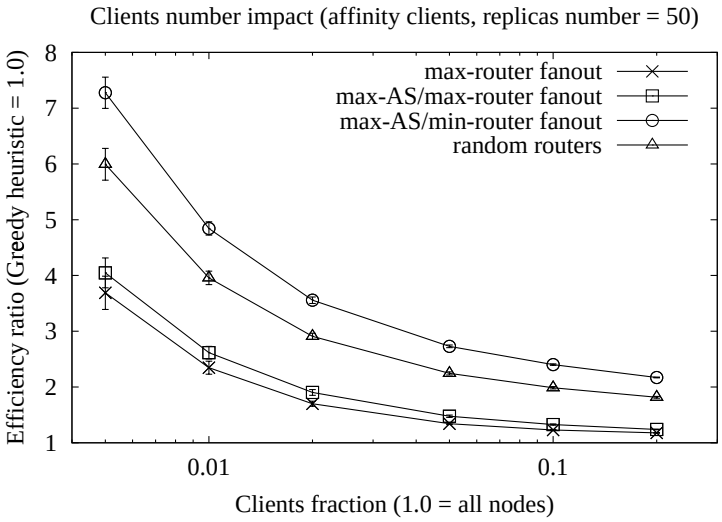


Figure 3.7: Internet core: client number impact (extreme affinity clients).

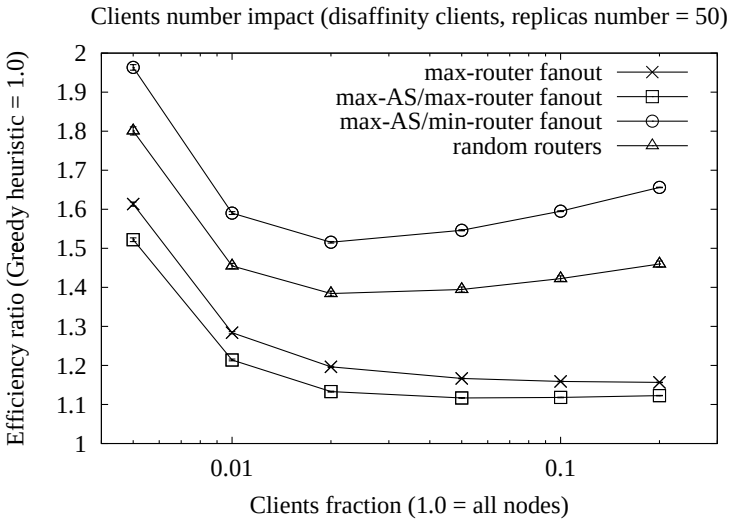


Figure 3.8: Internet core: client number impact (extreme disaffinity clients).

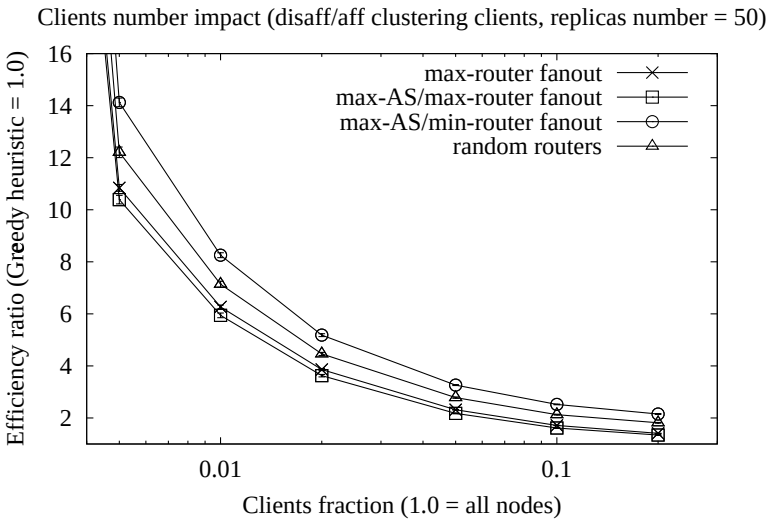


Figure 3.9: Internet core: client number impact (extreme clustering clients).

performance difference is larger with extreme affinity or extreme clustering of the clients. This result is not unexpected, and here is a possible explanation. When all clients are placed together in some part of the network (in case of extreme affinity), there is relatively low probability that there will be a node with large fanout in their proximity that will be selected as a replica (unless the number of clustered clients is very large and altogether they cover a notable fraction of the network). When all clients are clustered in various parts of the network (in case of extreme clustering), each of those clusters has a relatively small size, therefore it is even less likely there is a replica in a proximity of each cluster; yet, given the relatively small number of clusters, eventually most clusters are "pushed" to the edge of the network where is less likely there is a node with a large fanout. This also explains the notably worse performance of extreme clustering compared to extreme affinity and extreme disaffinity.

From the above results we can see that the client population size has impact on performance only when the number of clients is small. Only then the fanout-based placements perform notably worse than the greedy placement.

3.3.3.4 Network Topology Impact

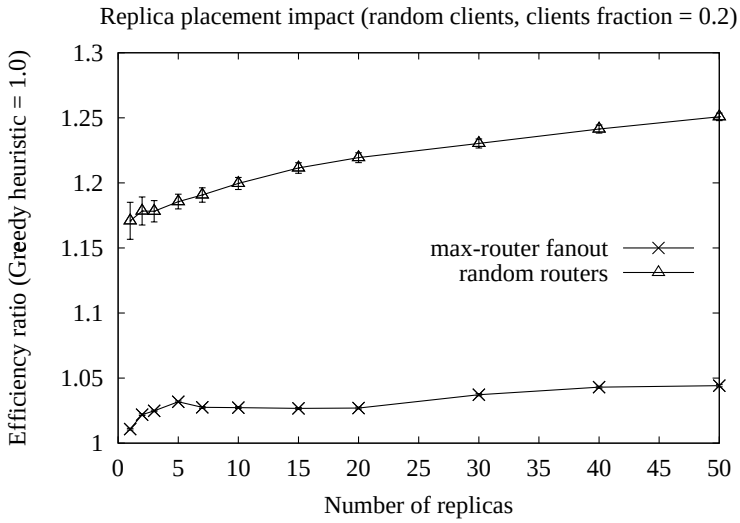


Figure 3.10: Random graph: replica placement impact (random clients).

The next factor we consider that may have impact on performance is the type of topology. First, we repeat the same simulations with two different types of generated topologies. The first one is a random graph (see Table 3.1 for some of its metrics, as well for the metrics of the other topologies), generated by the GT-ITM topology generator [8]. The second one is a power-law graph[6] created by a generator based on the algorithm described in [1]. A recent study shows that this topology qualitatively resembles both the AS and the router-level topologies [114]. Obviously, we do not have ASs over the generated

[6]One of the characteristics of the power-law graphs is that the node fanout distribution can be described by a power law: $f_d \propto d^{\beta}$ where f_d is the frequency of out-degree d, and β is a constant.

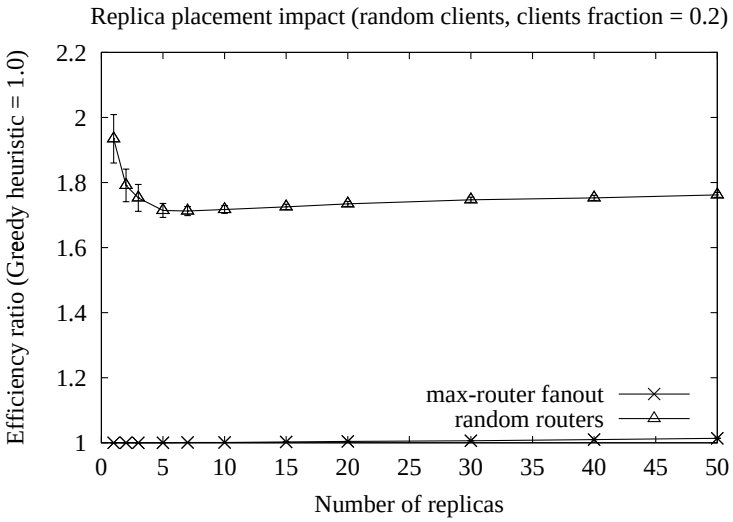


Figure 3.11: Power-law graph: replica placement impact (random clients).

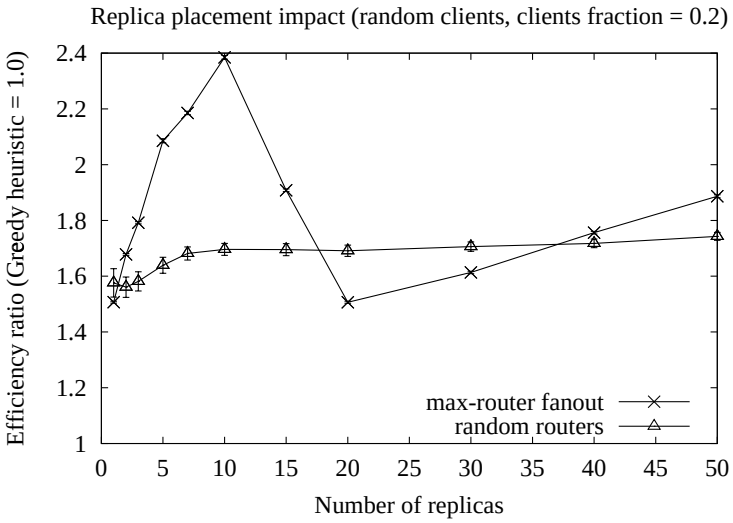


Figure 3.12: Mbone topology: replica placement impact (random clients).

topologies, therefore we have to use shortest-path routing; for the same reason we cannot apply the *max-AS/max-router* or the *max-AS/min-router* replica placement methods.

The results for the random graph with randomly placed clients (20% of all nodes) when we vary the number of replicas are in Figure 3.10. We can see that the difference between the greedy placement and the *max-router fanout* placement is even smaller (within a factor of 1.05). On the other hand, even random replica placement can perform within a factor of 1.25 which is much lower compared to the Internet-core results.

The results for the the power-law graph are in Figure 3.11. As expected, the fanout-based placement performs very well, while the random placement performs notably worse. We tried also a real-world AS-level topology itself [12], and the results were very similar to the results for the generated power-law graph.

However, we should note that this result is not universal. When we tried the Mbone [75] overlay topology [123] we found that the *max-router fanout* replica placement is not better than the random placement. In fact, in most cases it was even worse (see Figure 3.12). We believe the reason is that the connectivity of the Mbone topology is very sparse compared to the other topologies we have considered (compare the topology size, topology diameter and average inter-node distances of all topologies on Table 3.1). As a result, the average distance from a well-connected node to the rest of the nodes is relatively large, therefore such node is not a good choice to be a replica.

Finally, we should note that when we repeated the Internet-core simulations with using shortest-path routing instead of the approximated AS-policy routing, we did not see any notable difference. From this we can conclude that the impact of the routing on relative performance is insignificant.

3.4 Results Discussion

In this section we discuss our findings, and try to explain the reasons behind some of them.

Our main finding is that the fanout-based placement methods can perform remarkably well. Unlike more sophisticated methods such as greedy placement that takes into account the client locations to compute the appropriate replica placement, the fanout-based placement does not require any knowledge about expected client locations. This is a very significant simplification, because it basically suggests that we do not need a dynamic and adaptive replica placement that requires knowledge about expected client locations. In other words, as long as the replicas are placed in some "key" locations in the network, then the expected performance would be reasonably good. The only notable exception is when we use extreme clustering to place the clients, when the results are worse compared to other client placement methods, but even then the difference is not significant.

If we abstract from the particular replica placement methods and ask the question "if we had to select a single node as a replica, what would be the best node to select," the answer would be the node that is as close as possible to all clients. Typically, if a node has a large fanout, it means that it is a one-hop away from a large number of nodes, and therefore probabilistically it is close to a large number of clients as well. The AS and the Internet topologies have the characteristics of power-law graphs [34, 98]. One of the characteristics of those topologies is that they have a small number of nodes with very large fanout which nodes are apparently just few hops away from all other nodes. In other words, the high-fanout nodes are the "key-location" nodes for the power-law graphs that in most cases are very close to the rest of the nodes.

Here someone may ask the question why did we get similar results for the random graph which is not a power-law graph? The answer to that question may be in the fact that the majority of the nodes in the random graphs have similar fanout, while overall random graphs have very high *topology expansion* (defined as the growth of neighborhood size as a function of distance) [98]. The combination of these two factors eventually means that all nodes are just few hops away from each other, and then choosing any node to be a replica will be a good solution. This observation also explains why the random replica selection performs much better for random graph compared to power-law graph (see Figure 3.10 and 3.11).

Another observation is that a two-level fanout-based placement such as the *max-AS/max-router* placement method can perform very well. One possible speculation here may be that the AS fanout is the factor that matters, *i.e.,* that choosing any node within an AS with a large fanout will be a good solution. However, the results for the *max-AS/min-router* replica placement show that router selection based on the AS-level fanout only is not sufficient: the router-level fanout must be considered as well. One possible explanation to this is as follows. Typically, an AS with large fanout has inside a large number of nodes, and some of those nodes may not be very well connected.[7] Therefore, choosing such not well connected node as a replica may not be beneficial at all to reduce client latency or network overhead.

Finally, we should note that in most cases the *max-AS/min-router fanout* replica placement performs slightly better compared to the *max-router fanout* placement. The reason

[7]Indeed, when we compared the number of nodes assigned to each AS versus the AS fanout, on average the relation can be approximated with a straight line on the log-log scale, which means that each of the few ASs with the largest fanout contains a large number of nodes inside.

for this, we believe, is that the *max-AS/min-router fanout* placement spreads the replicas among a number of ASs, by placing no more than one replica inside each AS. On the other hand, the *max-router fanout* placement does not have this property, therefore it may place a number of replicas very close to each other without adding significant benefit to the clients.

3.5 Conclusions

In this chapter we studied the efficient replica placement for Content Distribution Networks. We have demonstrated that by using a small amount of information about the underlying topology such as node fanout, we can improve significantly the protocol performance. In our case, node fanout-based replica placement can be used to achieve results that are within a factor of 1.1–1.2 of solutions that are much more complicated and computationally expensive. The results are not affected by the client placement. However, they are affected by the underlying topology: the results can be applied to power-law and random graphs, but they do not apply for overlay topologies such as Mbone, or canonical topologies such as tree and mesh. This case-study suggests that in some cases it may be worth collecting some topology-related information that can be used to improve significantly protocol performance.

In the next chapter we study another problem where again we can use the help of partial knowledge about the underlying network to improve performance. In particular, we compare hierarchical router-assisted and application-level reliable multicast schemes, and we use partial topology knowledge to improve the creation of the application-level hierarchy.

Chapter 4

Case Study: Hierarchical Reliable Multicast

In this chapter we study another problem where we can benefit from using partial knowledge about the underlying network to improve protocol performance. We compare the performance of hierarchical router-assisted and application-level reliable multicast schemes. The router-assisted schemes use the help of the intermediate routers to create the data recovery hierarchy. Intuitively, such hierarchy would be more congruent with the underlying physical topology, therefore its performance should be significantly better compared to hierarchies that were created without the help of the intermediate routers. However, we find that if participants have information about all-pairs end-to-end distances, that partial information about the topology can be used to create an efficient application-level data recovery hierarchy. The performance of such hierarchy is comparable to the performance of router-assisted hierarchies.

4.1 Introduction

Reliable multicast has received significant attention recently in the research literature [36, 88, 72, 49, 16, 71, 68, 132]. The key design challenge for reliable multicast is scalable recovery of losses. The two main impediments to scale are *implosion* and *exposure*. Implosion occurs when, in the absence of coordination, the loss of a packet triggers simultaneous redundant messages (requests and/or retransmissions) from many receivers. In large multicast groups, these messages may swamp the sender, the network, or even other receivers. Exposure wastes resources by delivering a retransmitted message to receivers which have not experienced loss. Another challenge that arises in the design of reliable multicast is long *recovery latency*, which may result from suppression mechanisms to solve the implosion problem. Latency can have significant effect on application utility and on the amount of buffering required for retransmissions. Finally, highly dynamic groups may result in a loss of efficiency because they break assumptions about group constituency and structure.

One popular class of solutions is *hierarchical data recovery*. In these schemes, participants are organized into a hierarchy. By limiting the scope of recovery data and control messages between parents and children in the hierarchy, both implosion and exposure can be substantially reduced. Hierarchies introduce a latency penalty, but that only grows proportional to the depth of the hierarchy. The biggest challenge with hierarchical solutions is the construction and maintenance of the hierarchy, especially for dynamic groups. For optimal efficiency, the recovery hierarchy must be *congruent* with the actual underlying

multicast tree.[1] Divergence of these structures can lead to inefficiencies when children select parents who are located downstream in the multicast tree.

One approach, exemplified by RMTP [72], is to use manual configuration or application-level mechanisms to construct and maintain the hierarchy. Manual hierarchy construction techniques rely either on complete or partial (*e.g.,* where the border routers are) knowledge of the topology. Automated hierarchy construction techniques rely on dynamically discovering tree structure, either explicitly by tracing tree paths [71], or implicitly by using techniques based on expanding ring search. Once a hierarchy is formed, children recursively recover losses from their parents in the hierarchy by sending explicit negative acknowledgments.

Another approach, exemplified by LMS [88], proposes to use minimal router support not only to make informed parent/child allocation, but also to adapt the hierarchy under dynamic conditions. In some of these router-assisted schemes, hierarchy construction is achieved by routers keeping minimal information about parents for downstream receivers, then carefully forwarding loss recovery control and data messages to minimize implosion and exposure. In these schemes, hierarchy construction requires little explicit mechanism at the application-level at the expense of adding router functionality. Because of this, one would expect these *router-assisted hierarchies* (Section 4.2.2) to differ from the *application-level hierarchies* (Section 4.2.1) in two different ways: a) router-assisted hierarchies are *finer-grained*; that is, have many more "internal nodes" in the hierarchy; and b) they are more congruent to the underlying multicast tree.

[1] Congruency is achieved when the virtual hierarchy and the underlying multicast tree coincide.

Then, it is natural to ask, as we do in this work: Is the performance of application-level hierarchies qualitatively different than that of router-assisted hierarchies? To our knowledge, this question has not been addressed before. We study this question by evaluating two specific schemes: LMS and an RMTP-like schemes which use two specific hierarchy construction techniques. For our comparison we used four metrics: recovery latency, exposure, data traffic network overhead, and control traffic network overhead. We approach the question from two angles: first, we use analysis (Section 4.3) to determine the asymptotic behavior of the two schemes for regular trees, and second, we employ simulation (Section 4.4) to study the performance of large irregular multicast trees. These irregular multicast trees are randomly generated on real-world topologies such as the Internet [38], and the Mbone [75] topology [123].

Before doing this performance comparison, our expectation was that router-assisted hierarchies would significantly outperform application-level hierarchies. Our finding was surprising: that, with careful hierarchy construction, the performance of application-level hierarchies *is comparable* to that of router-assisted hierarchies, even though the former have a coarse-grained recovery structure. However, as we show, there exist pathological hierarchy construction techniques for which application-level hierarchies perform qualitatively worse than router-assisted hierarchies. Thus, the congruence of the hierarchy to the underlying multicast tree seems to be more important to performance that having a fine-grain recovery structure.

We should emphasize that we model only the essential features of the two schemes, and while our conclusions may be colored by the specific schemes we chose, we believe our results have a bearing on the larger issue of how router-assisted hierarchies compare to

application-level hierarchies. Furthermore, our conclusions inform but do not necessarily close the debate regarding the appropriate approach to hierarchical data recovery. Our evaluation metrics do not capture the complexity and cost of hierarchy construction, or the complexity of adding router-assistance for hierarchical recovery to the network.

The rest of the chapter is organized as follows. In Section 4.2 we present in details the application-level and router-assisted schemes we consider in this paper, and describe the evaluation metrics. Section 4.3 presents the k-ary tree analytical results for both schemes. Section 4.4 present and discusses the simulation results for real-network topologies. Conclusions are in Section 4.5.

4.2 Hierarchical Multicast Data Recovery Schemes

As the name implies, hierarchical reliable multicast schemes solve the scalability problem by structuring the group into a hierarchy. Because a hierarchy explicitly enforces scope on the data recovery, it is a natural approach to address many of the problems described earlier, including implosion, exposure and latency. Based on the mechanisms used to create and maintain the hierarchy, we can distinguish between two classes of hierarchical schemes. The first class, *application-level hierarchical schemes (ALH)*, uses only end-to-end mechanisms assisted by the end-systems (the receivers) to create and maintain the hierarchy. The second class, *router-assisted hierarchical schemes (RAH)*, uses assistance from the routers in the creation and maintenance of the hierarchy.

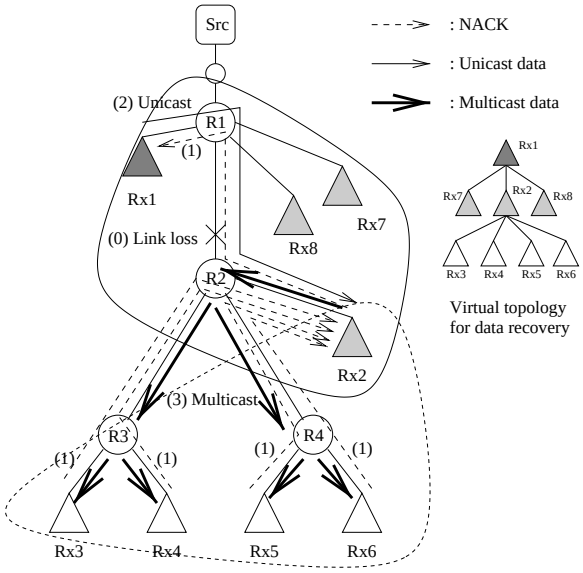


Figure 4.1: ALH example: optimal hierarchy organization.

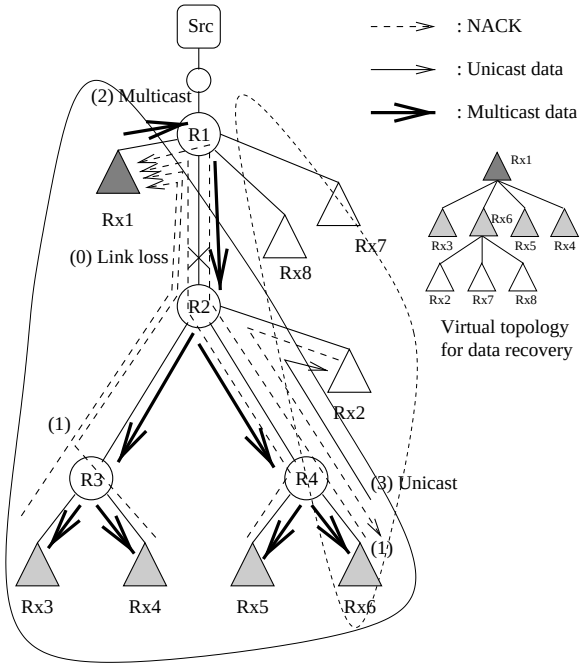


Figure 4.2: ALH example: sub-optimal hierarchy organization.

4.2.1 Application-Level Hierarchical Schemes

ALH schemes create and maintain the data recovery hierarchy by using only end-to-end mechanisms. Typical mechanisms include manual (static) configuration and expanding ring search to locate the nearest candidates. More sophisticated schemes employ heuristics such as "loss fingerprinting" where receivers compare their loss fingerprints with those of potential parents and select the most appropriate. Both types, however, tend to be slow to adapt to dynamic conditions and are not always accurate in maintaining congruency.

The Reliable Multicast Transport Protocol (RMTP) [72] is an example of an ALH scheme, and forms the basis of our ALH model. RMTP employs a combination of positive and negative acknowledgments (ACK and NACK) for data recovery. Because the focus of this work is not on modeling and evaluating protocol details, but rather understanding the underlying mechanisms, we do not follow exactly the RMTP protocol; instead, we adopt the hierarchical approach in RMTP and model only NACKs and retransmissions.

Briefly, our ALH scheme works as follows. In Figure 4.1 $Rx1$ is a parent for $Rx2$, $Rx7$, and $Rx8$, while $Rx2$ itself is a parent for $Rx3$, $Rx4$, $Rx5$ and $Rx6$. Upon detecting loss (link $R1 - R2$), children unicast NACKs to their parents ($Rx3$, $Rx4$, $Rx5$, $Rx6$ to $Rx2$, and $Rx2$ to $Rx1$.) If the parent has the data, the parent sends it to its children by either unicast or multicast. A multicast response is sent to a local multicast group where only the children and the parent are members of that group. To select between unicast and multicast, a parent collects NACKs and uses multicast if at least 50% of the children requested data retransmission; otherwise the parent uses unicast (parent $Rx1$ to $Rx2$.) If a parent does not have the requested data, its own parent also detects the loss from missing

acknowledgments (and so on until we reach the root). After receiving the data each parent sends it to its children.

RMTP does not explicitly specify how the hierarchy is created; rather, in its current incarnation it assumes a manually configured static hierarchy. In order to explore the potential of ALH schemes, we introduce a rather powerful heuristic: we hypothesize that all participants have somehow obtained information about the distance to each other, and use that information in a heuristic algorithm to create the hierarchy. The algorithm creates the hierarchy in a bottom-up fashion as follows: among a group of participants, the node with the smallest sum of distances to all other nodes becomes a parent. Initially, all receivers are eligible to become parents and thus the lowest level of the hierarchy is formed by selecting parents among all receivers. Each of the receivers which was not elected as a parent chooses the closest parent node as its parent. The same heuristic is recursively applied at the next level among all the nodes that were selected as parents in the previous iteration, until we are left with a small number of nodes which become children of the root of the tree (*i.e.,* the sender). The depth of the hierarchy is defined by the fraction of nodes to choose at each iteration, which is a number in the interval $(0.0, 0.5)$. A value of 0.1 for example means that among all nodes at level i in the data recovery hierarchy, 10% of them will become parents of the remaining 90%.

4.2.2 Router-Assisted Hierarchical Schemes

Router-assisted hierarchical schemes (RAH) use network assistance to achieve congruency between the hierarchy and the multicast tree. By eliminating the need to maintain the hierarchy through potentially expensive and complicated endsystem-based mechanisms, RAH

schemes reduce application complexity and enable the development of large-scale reliable multicast applications. For our evaluation of RAH schemes we chose Light-weight Multicast Services (LMS) [88] as our model. LMS employs router assistance to create a dynamic hierarchy which continuously tracks the underlying multicast routing tree regardless of membership changes. The network assistance required by LMS is in the form of new forwarding services at the routers, and thus has no impact at the transport level. With LMS each router marks a downstream link as belonging to a path leading to a *replier*. A replier is simply a group member willing to assist with error recovery by acting as a parent for that router's immediate downstream nodes. Because they are selected by routers, parents are always upstream and close to their children. The forwarding services introduced by LMS allow routers to steer control messages to their replier, and allow repliers to request limited scope multicast from routers. More specifically, LMS adds the following three new services to routers:

- *Replier selection:* potential repliers advertise their willingness to serve as repliers for a particular *(Source, Group)* pair with their local router. Routers propagate these advertisements upstream. Before propagating the message upstream, a router selects one of its downstream interfaces (based on an application-defined metric) as the replier interface. When all routers have received advertisements the replier state is established. Replier state is soft state which provides robustness and guards against replier and link failures.

- *NACK forwarding:* LMS routers forward NACKs hop-by-hop according to the following rules: a NACK from the replier interface is forwarded upstream; a NACK

from a non-replier interface (including the upstream interface) is forwarded on the replier interface. However, a NACK from a non-replier downstream interface marks this router as the "turning point" of that NACK. Note that by definition, there can be only one turning point for each NACK but the same turning point may be shared by multiple NACKs. Before forwarding a NACK, the turning point router inserts in the packet the addresses of the incoming and outgoing interfaces, which we call the "turning point information" of the NACK. This information is carried by the NACK to the replier.

- *Directed multicast (DMCAST):* DMCAST is used by repliers to perform fine-grain multicast. A replier creates a multicast packet containing the requested data and addresses it to the group. The multicast packet is encapsulated into an unicast packet and sent to the turning point router (whose address was part of the turning point information) along with the address of the interface the NACK originally arrived at the turning point router. When the turning point router receives the packet, it decapsulates and multicasts it on the specified interface. An enhanced version of DMCAST may allow repliers to specify more than one interface that the packet should be directed to send on.

LMS works well in most cases to deliver the requested packet with minimum latency and only to receivers that need it. Figure 4.3 shows such an example. The loss on link $R1 - R2$ is recovered from replier $R1$ by sending a DMCAST to $R1$. However, in some cases LMS may expose receivers to retransmissions that do not need it. This occurs when loss happens on the replier path, as shown in Figure 4.4. The resulting exposure does not

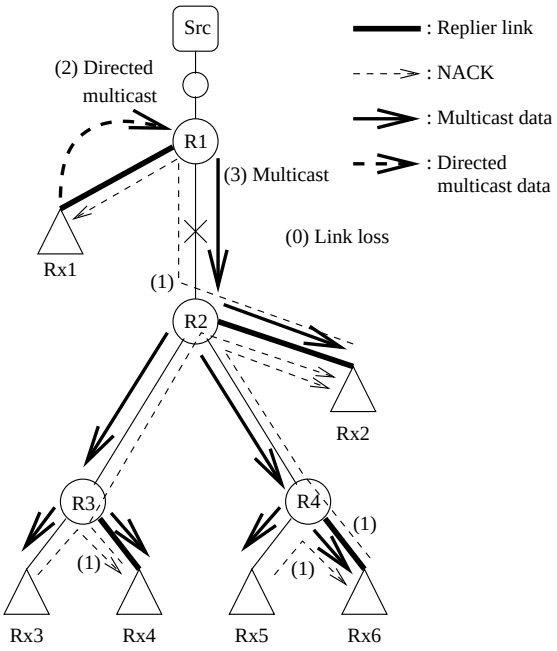


Figure 4.3: LMS vanilla example: data loss and recovery.

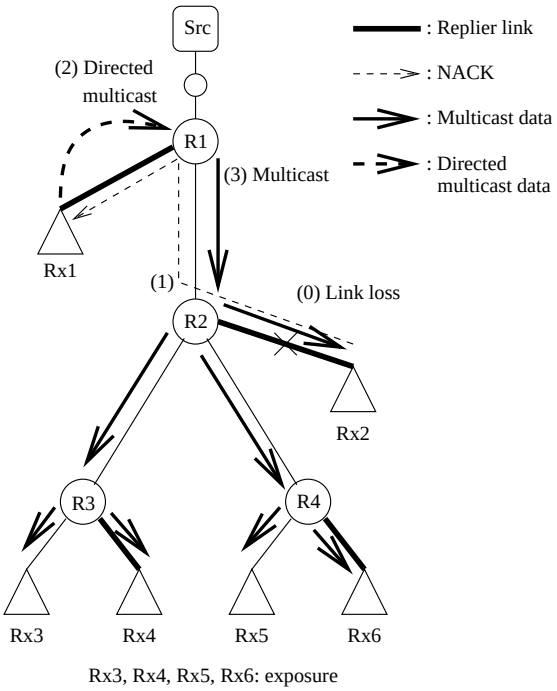


Figure 4.4: LMS vanilla example: data loss by replier only and exposure to other receivers.

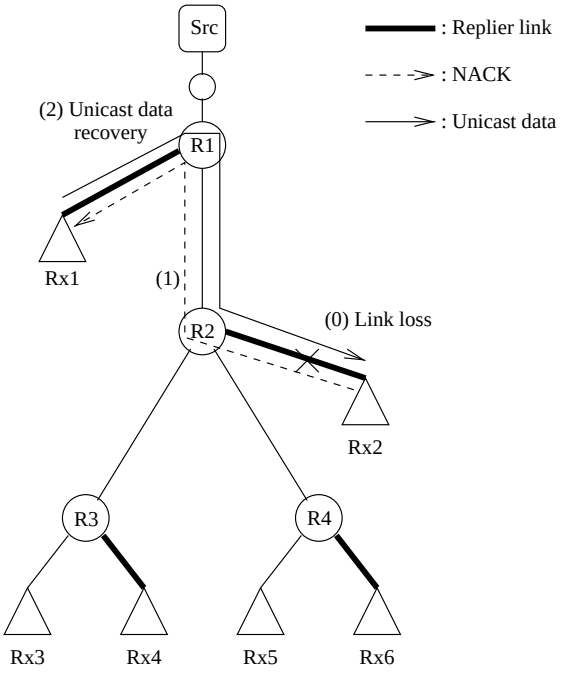


Figure 4.5: LMS enhanced example: data loss by replier only and unicast recovery.

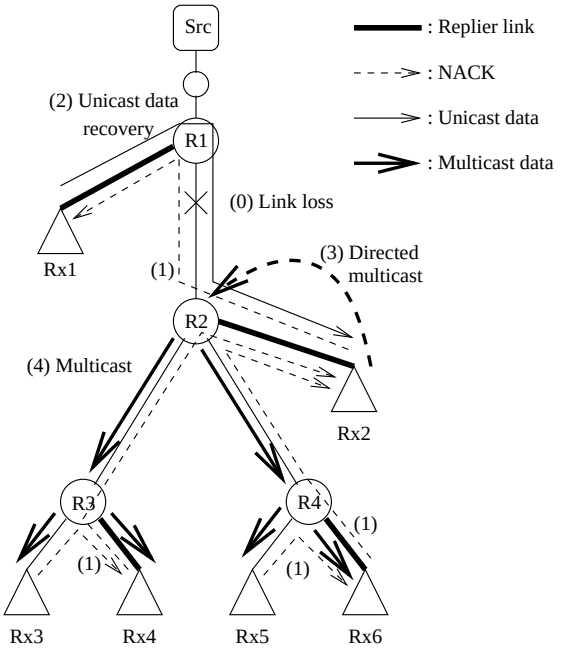


Figure 4.6: LMS enhanced example: two-step data recovery (unicast followed by direct multicast).

affect correctness but may lead to wasted resources if a replier branch (the link between $R2$ and $Rx2$ in our example) is particularly lossy. LMS addresses this problem by selecting the replier branch that advertises the least loss. However, determining path loss characteristics can be hard, and thus LMS employs another method to eliminate exposure, which comes at the cost of eventually adding an extra hop to the retransmission. With this enhancement, a NACK by a downstream replier specifies that the reply should be unicast to the requestor itself rather than the turning point. For example, in Figure 4.5 (the same loss scenario as in Figure 4.4) $Rx1$ will directly unicast the reply to $Rx2$ and therefore there will be no exposure on the subtrees rooted at $R3$ and $R4$. The extra hop of retransmission can be illustrated by the example in Figure 4.6 where the packet loss occurs on the link between $R1$ and $R2$. Similar to the previous example, the request by downstream replier $Rx2$ will reach $Rx1$ and the reply will be unicast back to $Rx2$. However, because in the mean time $Rx2$ has received NACKs from the downstream parts of the tree ($Rx4$ and $Rx6$), now it just needs to send a single enhanced directed multicast to $R2$ specifying that the reply should be multicast on links $R2-R3$ and $R2-R4$. Only if at some later time the requestor $Rx2$ receives more requests, direct multicasts are sent to the remaining part of the subtree. Note that this two-step process occurs only once, between the replier above the loss and the first requestor. We distinguish the previous version (which we call vanilla LMS) from this version, which we call "enhanced LMS." Preliminary experiments have shown that for large groups the increase in latency in enhanced LMS is negligible but there is a significant reduction in exposure. Therefore, in this work we use the enhanced version of LMS.

We note that LMS is not the most aggressive router-assisted recovery scheme. Finer grain recovery schemes, in which routers themselves respond to loss recovery requests from

downstream neighbors, can, perhaps perform better than LMS. While such schemes are conceivable, we believe they are impractical in that they require significant router state.

4.2.3 Metric Space

We did not model the overhead of creating a hierarchy with ALH schemes, because this depends strongly on the application and network characteristics. For example, in an application where membership is static, parents can be deployed manually and can yield excellent performance. At the other extreme, applications with mobile receivers may impose many restrictions on the type of the hierarchy creation algorithm and the parent–child associations.

If we ignore the overhead for creating and maintaining the hierarchy, the main source of inefficiencies in ALH schemes is the lack of congruency between the possibly fine grain hierarchy and the underlying multicast tree. Divergence of the two structures results in problems when children inadvertently join parents that are located downstream or are too far away, which results in an increase in recovery latency and network overhead. For example, in Figure 4.2 $Rx6$ is at the very bottom of the multicast delivery tree, but is inadequately a parent for the upstream $Rx2$, $Rx7$ and $Rx8$. RAH schemes do not suffer from these problems because they continuously track the multicast topology (although the cost varies among different schemes).

We have defined a set of metrics to evaluate the two schemes which represent the impact of the data recovery mechanism on the application and the network. These metrics are defined below; in section 4.2.4 we will present some examples of those metrics computed for different data recovery schemes.

- **Data recovery latency**. The recovery latency is defined as the ratio of the data recovery time observed by a receiver and the round-trip time from that receiver to the sender. A smaller value means that the receivers will wait shorter time to receive the missing data. For example, latency of 0.5 means that the time it will take for the receiver to recover the data is half of the round-trip time to the root/sender. The formula we use to compute the average data recovery latency across all receivers and across all links being lossy is:

$$NormLat = \frac{\sum_{receivers(r)} \sum_{links(l)} \frac{Lat(r,l)}{RTT(r)}}{NumberOfLossRcvs * NumberOfLinks}$$

where $Lat(r, l)$ is the receiver r latency when the packet loss is on link l, $RTT(r)$ is the round-trip time from receiver r to the root of the tree, $NumberOfLossRcvs$ is the total number of receivers that have observed any loss, and $NumberOfLinks$ is the total number of links in the topology.

In ALH schemes, sources of latency include longer recovery paths due to lack of congruency between the hierarchy and the multicast tree, and the latency due to multiple hops (parents). RAH schemes typically do not suffer from these problems because they (a) almost always recover from the nearest replier, and (b) have the capability of sending the multicast data to only one branch of the tree.

- **Receiver exposure**. The exposure is defined as the ratio of the extra amount of packets that have been received by a receiver (and eventually discarded), and the total number of packets sent by the sender. Ideally, this metric should be 0 (*i.e.,* no extra packets are received and no extra processing by the receivers). The formula we

use to compute the average exposure across all receivers and across all links being lossy is:

$$NormExp = \frac{\sum_{receivers(r)} \sum_{links(l)} Exposure(r,l)}{NumberOfExpRcvs * NumberOfLinks}$$

where $Exposure(r,l)$ is the exposure for receiver r (in term of number of extra packets) when the packet loss is on link l, $NumberOfExpRcvs$ is the total number of receivers that have observed any exposure, and $NumberOfLinks$ is the total number of links in the topology.

ALH schemes suffer from exposure when more than half (but not all) children lose a packet which result into a local multicast. With RAH schemes the problem is limited to only few specific cases due to the ability to use subcast.

- **Data traffic network overhead**. The data traffic network overhead is defined as the ratio of the amount of used network resources because of the retransmitted multicast data (in term of total number of data packets sent over any link in the network), and the size of the subtree (in number of links) that did not receive the data. In the ideal case the data network overhead will be 1.0 (*e.g.*, when the node right above the lossy link has the data and it will send a single multicast packet down the whole branch of the tree that did observe the loss). ALH schemes suffer from this overhead because of the inefficiency introduced by the unicast/multicast combination. The formula we use to compute the average data overhead across all lossy links is:

$$NormDataOverhead = \frac{\sum_{links(l)} \frac{Data(l)}{Subtree(l)}}{NumberOfLinks}$$

where $Data(l)$ is the total amount of data traffic that will be created when the packet loss is on link l, $Subtree(l)$ is the size of the subtree (in term of number of links) that did not receive the data, *i.e.,* the subtree below (and including) link l, and $NumberOfLinks$ is the total number of links in the topology.

- **Control traffic network overhead**. Similar to the data traffic overhead, the control overhead is defined as the ratio of the amount of used network resources by the control packets (the NACKs), and the size of the subtree that did not receive the data. We consider ratio of 1.0 as optimal, even though this is not the theoretically lowest ratio. For example, if the node right above the lossy link was a replier, the control overhead will be 1.0 if there was exactly one NACK sent over each of the links of the subtree below the lossy link. ALH schemes may suffer more than RAH schemes because with ALH there is less opportunity to do NACK fusion. Similar to the data overhead, the formula we use to compute the average control overhead is:

$$NormControlOverhead = \frac{\sum_{links(l)} \frac{Control(l)}{Subtree(l)}}{NumberOfLinks}$$

where $Control(l)$ is the total amount of control traffic that will be generated in the network when the packet loss is on link l.

4.2.4 Examples of Measuring ALH and RAH Performance

The metrics we described in the previous section can be illustrated by the following examples. Consider first the ALH example in Figure 4.1. Five of the receivers will send NACKs to their parents, and the control overhead will be $\frac{15}{8} = 1.875$ (the size of the subtree that

did not receive the data is 8). The data overhead then will be $\frac{3+7}{8} = 1.25$. The data latency for receiver $Rx2$ will be 6 (the RTT to $Rx1$), but the latency for $Rx3$, $Rx4$, $Rx5$, $Rx6$ will be $-1 + 6 + 3 = 8$.[2] If we assume that the sender is two hops away from $R1$, then the average data latency will be $\frac{6/8+4*8/10}{5} = 0.79$. The exposure in this particular example is 0. If the ALH data recovery hierarchy was not created efficiently, such as the hierarchy in Figure 4.2, then the latency, data and control overhead will be respectively 0.94, 1.375, and 2.375 (note that the latency for $Rx2$ is 12, because it is one hop closer to the sender than its parent). The exposure in this example is also 0.

If we look in the example for the RAH scheme in Figure 4.6 (enhanced LMS) which has the same setup of receivers and link loss as in the above example, the latency, data and control overhead, and exposure are respectively 0.79, 1.25, 1.625 and 0. On the contrary, if we used vanilla LMS (see Figure 4.3), the latency, data, and control overhead would be respectively 0.63, 1.125 and 1.625. However, the average receiver exposure in Figure 4.4 for each of the receivers that received extra packet ($Rx3$, $Rx4$, $Rx5$, $Rx6$) will be $\frac{4*(2-1)}{4} = 1.0$.

4.3 K-ary Tree Analyses of RAH and ALH

To get initial understanding of the scalability property of the ALH and RAH schemes, we apply some simple analyses on k-ary trees. In our analyses we assume that the root of the tree is the sender, and that a fraction of the leaf nodes are receivers. Thus, a k-ary tree of depth L has between k and k^L receivers. The receiver set size is specified by a parameter q ($1 \leq q \leq L$), such that the fraction of leaf nodes that are receivers is $\frac{1}{k^{q-1}}$. For example,

[2] Receiver $Rx2$ will discover the data loss and will initiate the recovery one "link-hop" time unit earlier than $Rx3$, $Rx4$, $Rx5$ and $Rx6$, hence the -1 in the latency computation.

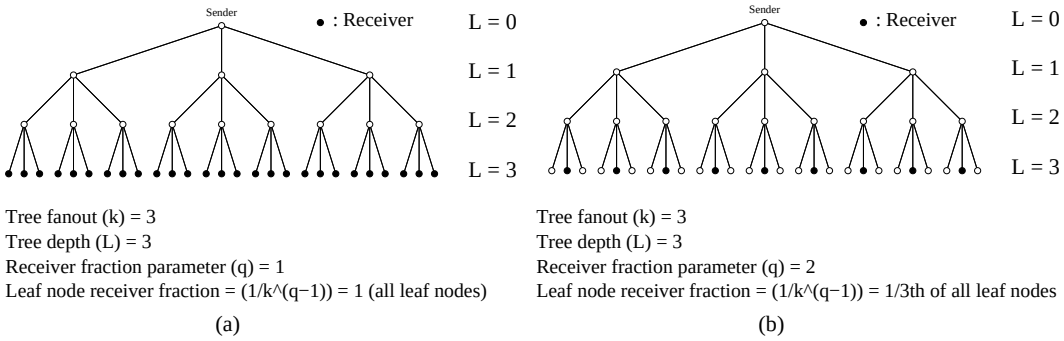


Figure 4.7: Example of k-ary tree parameters.

Figure 4.7(a) has an example of a 3-ary tree with tree depth $L = 3$ and $q = 1$ (*i.e.*, all leaf nodes are receivers). Figure 4.7(b) shows the same tree but with $q = 2$ (*i.e.*, 1/3th of the leaf nodes are receivers). In case of ALH we assume that the recovery hierarchy is created such that each parent has $k - 1$ children. Given these assumptions, the same parent nodes for ALH are the repliers for RAH. For each of the schemes we assume a single link loss and compute the average (per link-loss across all links) for each of the metrics described in Section 4.2.3.

First we present the analytical results for the control overhead, and briefly describe the methodology used to derive them. Then we present the analytical results for data overhead, and for latency recovery. We do not analyze the receiver exposure metric because in this particular scenario by definition it is always zero for both RAH and ALH.

4.3.1 RAH and ALH control overhead analyses results

In the particular scenario of a k-ary tree with the receivers chosen among the leaf nodes, the RAH and ALH control overhead are the same by definition, therefore the results below are same for both methods.

To analyze the average control overhead, we need to compute the following: (a) if there is a packet loss on a link, what is the control overhead (in term of number of hops) to recover the packet; (b) what is the size of the subtree below (and including) the link where the loss has occurred, so we can compute the relative control overhead for that lossy link, and (c) what is the relative control overhead averaged among all links in the tree being lossy (one at a time).

First we can make the following observation. If we assume that all links at some particular level of the tree become lossy one-after-another, the sum of the control overhead for those links does not depend on the level of the tree the links belong to. For example, in case of the 3-ary tree in Figure 4.7(b) with only 1/3th of all leaf nodes as receivers, if all leaf links go down one-by-one, the control overhead sum in number of hops to reach a replier or a parent for the first three leftmost receivers would be: $2 * 2 + 2 * 2 + 3 * 2 = 14$. The control overhead sum for the three receivers in the middle is same, but for the three rightmost receivers the control overhead sum is $2 * 2 + 2 * 2 + 3 = 11$. Therefore the sum among all nine receivers is $14 + 14 + 11 = 39$. If we consider the control overhead for the links right above the $L = 2$ nodes, then the total sum will be same, except that the subtree size below each lossy link is two instead of one. Similarly, if we consider the topmost three links, the control overhead sum if they are lossy (one at a time) will be same, except that each time there is a link loss, this will affect three receivers instead of one.

The control overhead sum can be computed by considering iteratively the size of the corresponding subtrees. Hence, if all links at some level are lossy one at a time, the control overhead sum is:

$$
\begin{aligned}
&ControlOverheadSum(1 \le i \le L) = \\
&= \sum_{i=1}^{L-q+1} (k^{L-q+1-i+1} - k^{L-q+1-i}) * 2 * (i + q - 1) \\
&= 2 * (\frac{k^{L-q+1} - k}{k-1} + q * k^{L-q+1}) - L
\end{aligned} \tag{4.1}
$$

where k is the tree fanout, L is the tree depth, and q defines the fraction of leaf nodes that are receivers according to the formula: $\frac{1}{k^{q-1}}$. This sum is same, regardless of the particular level of the lossy links.

To compute the relative control overhead, we need to compute the tree size below a lossy link. If the lossy links are located in level i (starting to count from the bottom of the tree), the subtree size depends on the value of i and can be expressed by the following two equations:

$$
SubtreeSize_{(1 \le i \le q)} = i \tag{4.2}
$$

$$
\begin{aligned}
&SubtreeSize_{(q+1 \le i \le L)} = \\
&= \frac{k^{i-1-q+1} - 1}{k-1} + q * k^{i-q} \\
&= \frac{q * k^{i-q+1} - (q-1) * k^{i-q} - 1}{k-1}
\end{aligned} \tag{4.3}
$$

Therefore, the relative control overhead sum across all links is:

$$\sum_{AllLinks} RelativeControlOverhead =$$

$$= \sum_{i=1}^{q} \frac{ControlOverheadSum(i)}{i} + \sum_{i=q+1}^{L} \frac{ControlOverheadSum(i) * (k-1)}{q * k^{i-q+1} - (q-1) * k^{i-q} - 1} \quad (4.4)$$

Finally, to obtain the average single-link relative control overhead, we need to divide the above sum by the total number of the lossy links (*i.e.*, the links with downstream receivers):

$$\begin{aligned} &LossyLinksNumber = \\ &= \frac{k^{L+1} - (q-1) - k}{k-1} + (q-1) * k^{L-q+1} \\ &= \frac{k^{L-q+1} * (q * k - q + 1) - k}{k-1} \end{aligned} \quad (4.5)$$

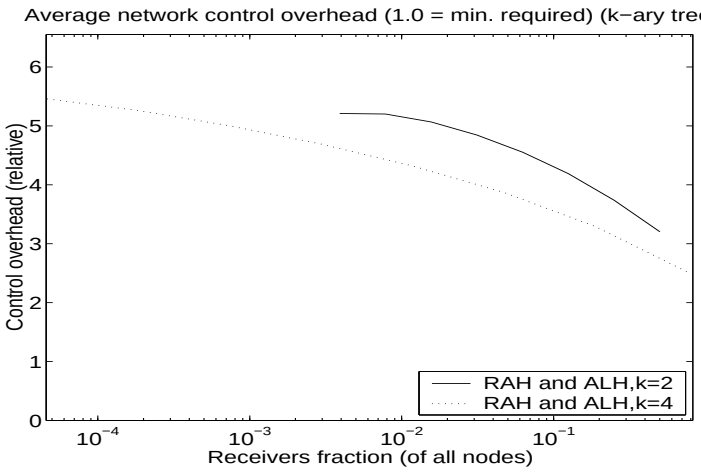


Figure 4.8: RAH and ALH: average network control overhead.

Based on the above expressions, we can plot the results for different trees. Figure 4.8 shows the network control overhead for both RAH and ALH for binary and 4-ary trees of depth

$L = 10$ when we vary the fraction of the receiver nodes. The results for larger tree depth and fanout were similar.

4.3.2 RAH and ALH data overhead analyses results

Unlike the control overhead, there is difference between the data overhead for RAH and ALH. To compute the data overhead, we use a method similar to the computation of the control overhead described in Section 4.3.1.

4.3.2.1 RAH data overhead analyses results

In case of RAH, if all links at level s ($s = 1$ for leaf links) go down one-by-one, the sum of the data overhead is:

$$\begin{aligned} & DataOverheadSum(s, 1 \leq s \leq q)_{RAH} = \\ & = \sum_{i=1}^{L-q+1} (k^{L-q+1-i+1} - k^{L-q+1-i}) * 2 * (i + q - 1) + L \\ & = 2 * \frac{k^{L-q+1} - k}{k-1} + 2 * q * k^{L-q+1} - L \end{aligned} \tag{4.6}$$

$$\begin{aligned} & DataOverheadSum(s, q+1 \leq s \leq L)_{RAH} = \\ & = \sum_{i=s-q+1}^{L-q+1} (k^{L-q+1-i+1} - k^{L-q+1-i}) * 2 * (i + q - 1) \\ & + \sum_{i=1}^{s-q} k^{L-q+1-i+1} + (q-1) * k^{L-q+1} + k^{L-s+1} * \sum_{i=1}^{s-q+1} (i + q - 1) + L \\ & = 2 * k^{L-s+1} * (s + \frac{1}{k-1}) - 2 * \frac{k}{k-1} + \frac{k^{L-q+2} - k^{L-s+2}}{k-1} + (q-1) * k^{L-q+1} \\ & + k^{L-s+1} * \frac{(s-q+1) * (s+q-2)}{L} - L \end{aligned} \tag{4.7}$$

To compute the relative data overhead, we need to divide the absolute data overhead sum by the size of the affected subtree. Therefore, the sum of the relative data overhead when all links go down one-by-one is:

$$\sum_{AllLinks} RelativeDataOverhead_{RAH} =$$
$$= \sum_{i=1}^{q} \frac{DataOverheadSum(i)}{i} + \sum_{i=q+1}^{L} \frac{DataOverheadSum(i) * (k-1)}{k^{i-q+1} * q - k^{i-q} * (q-1) - 1} \quad (4.8)$$

Finally, to compute the average relative data overhead, we need to divide the above sum by the total number of links (see Equation 4.5).

4.3.2.2 ALH data overhead analyses results

In case of ALH, if all links at level s ($s = 1$ for leaf links) go down one-by-one, the sum of the data overhead is:

$$DataOverheadSum(s, 1 \leq s \leq q)_{ALH} =$$
$$= \sum_{i=1}^{L-q+1} (k^{L-q+1-i+1} - k^{L-q+1-i}) * 2 * (i+q-1) + L$$
$$= 2 * \frac{k^{L-q+1} - k}{k-1} + 2 * q * k^{L-q+1} - L \quad (4.9)$$

$$DataOverheadSum(s, q+1 \leq s \leq L)_{ALH} =$$
$$= \sum_{i=s-q+1}^{L-q+1} (k^{L-q+1-i+1} - k^{L-q+1-i}) * 2 * (i+q-1)$$
$$+ \sum_{i=1}^{s-q} k^{L-q+1-i} * ((k-1) * (i+q-1) + 1 + q - 1) + L =$$

$$= \quad k^{L-s+1} * (q + s + \frac{2}{k-1}) - 2 * \frac{k}{k-1}$$
$$+ \quad (k^{L-q+1} - k^{L-s+1}) * (q + \frac{q+1}{k-1}) - L \tag{4.10}$$

To compute the relative data overhead, we need to divide the absolute data overhead by the size of the affected subtree. Therefore, the sum of the relative data overhead when all links go down one-by-one is:

$$\sum_{AllLinks} RelativeDataOverhead_{ALH} =$$
$$= \sum_{i=1}^{q} \frac{DataOverheadSum(i)}{i} + \sum_{i=q+1}^{L} \frac{DataOverheadSum(i) * (k-1)}{k^{i-q+1} * q - k^{i-q} * (q-1) - 1} \tag{4.11}$$

Finally, to compute the average relative data overhead, we need to divide the above sum by the total number of links (see Equation 4.5).

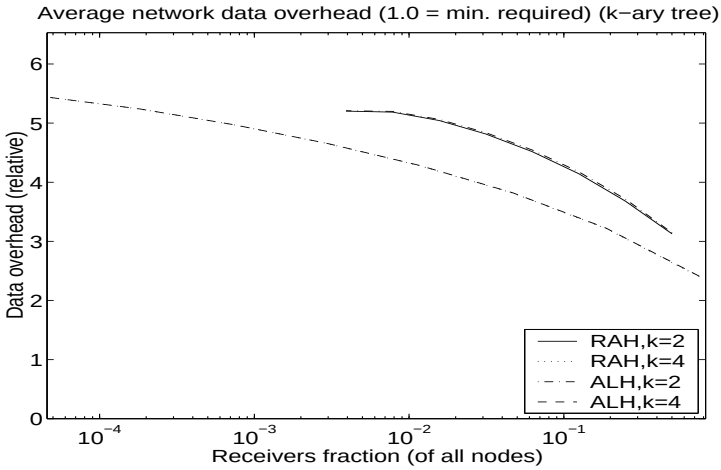


Figure 4.9: RAH and ALH: average network data overhead.

Similar to the results for the control overhead, we use the above equations to plot the results for binary and 4-ary trees of depth $L = 10$ (see Figure 4.9). First, we can notice that there is almost no difference between the RAH and ALH data overhead. Further, the results are very similar to the network control overhead (see Figure 4.8). The reason that the difference between RAH and ALH is very small is as follows. In this particular setup, the advantage of using RAH comes from the data retransmission method: in some cases a single RAH replier would use multicast to retransmit the data to all receivers within a subtree, while in case of ALH the retransmission would be a sequence of multicast retransmissions, one at each level of the hierarchy. However, only when the lossy link is close to the root of the tree, the number of sequential retransmissions can be on the order of L, and even then the extra latency would be relatively small. Therefore, on average the RAH advantage compared to ALH is very small. The reason that there is small difference between the network data overhead and network control overhead is because in most cases the data retransmission will be by unicast, and by definition the data overhead when we use unicast to retransmit the data is same as the control overhead.

4.3.3 RAH and ALH data recovery latency analyses results

The data recovery latency computation method is slightly different from the computation of the data and control overhead. First we compute the sum of the latencies when all links at some level s go down one-by-one. After that we sum the latencies for all $1 \leq s \leq L$. Finally, we normalize the result by the round-trip time to the sender, and then we average across all links.

4.3.3.1 RAH data recovery latency analyses results

In case of RAH, if all links at level s ($s = 1$ for leaf links) go down one-by-one, the sum of the data recovery latency is:

$$\begin{aligned} & RecoveryLatencySum(s, 1 \leq s \leq q)_{RAH} = \\ & = \sum_{i=1}^{L-q+1} (k^{L-q+1-i+1} - k^{L-q+1-i}) * 4 * (i - q + 1) + 2 * L \\ & = 4 * k^{L-q+1} * (q + \frac{1}{k-1}) - 4 * \frac{k}{k-1} - 2 * L \end{aligned} \tag{4.12}$$

$$\begin{aligned} & RecoveryLatencySum(s, q+1 \leq s \leq L)_{RAH} = \\ & = \sum_{i=s-q+1}^{L-q+1} (k^{L-q+1-i+1-k^{L-q+1-1}}) * 4 * (i + q - 1) \\ & + \sum_{i=1}^{s-1-q+1} (k^{L-q+1-i+1} - k^{L-q+1-i}) * 2 * (s-1) \\ & + (k^{s-1-q+1} - 1) * \sum_{i=s-q+1}^{L-q+1} (k^{L-q+1-i+1} - k^{L-q+1-i}) * 4 * (i + q - 1) \\ & + \sum_{i=1}^{s-1-q+1} (k^i - k^{i-1}) * 2 * L + 2 * L \\ & = 4 * k^{L-q+1} * (s + \frac{1}{k-1}) - 4 * k^{s-q+1} * \frac{1}{k-1} \\ & + 2 * (k^{L-q+1} - k^{L-s+1}) * (s - \frac{k}{k-1}) \\ & + 2 * (s-q) * k^{L-s+1} - 2 * L * k^{s-q} \end{aligned} \tag{4.13}$$

Therefore, the average relative data recovery latency is:

$$AveRelativeRecoveryLatency_{RAH} = \frac{\sum_{s=1}^{L} RecoveryLatencySum(s)}{k^{L-q+1} * L * 2 * L} \tag{4.14}$$

where k^{L-q+1} is the total number of receivers, L is the number of links that could affect a receiver, and $2 * L$ is the RTT to the sender (same for all receivers).

4.3.3.2 ALH data recovery latency analyses results

In case of ALH, if all links at level s ($s = 1$ for leaf links) go down one-by-one, the sum of the data recovery latency is:

$$
\begin{aligned}
& RecoverLatencySum(s, 1 \leq s \leq q)_{ALH} = \\
&= \sum_{i=1}^{L-q+1} (k^{L-q+1-i+1} - k^{L-q+1-i}) * 4 * (i - q + 1) + 2 * L \\
&= 4 * k^{L-q+1} * (q + \frac{1}{k-1}) - 4 * \frac{k}{k-1} - 2 * L
\end{aligned} \tag{4.15}
$$

$$
\begin{aligned}
& RecoverLatencySum(s, q+1 \leq s \leq L)_{ALH} = \\
&= \sum_{i=s-q+1}^{L-q+1} (k^{L-q+1-i+1-k^{L-q+1-1}}) * 4 * (i + q - 1) \\
&+ \sum_{i=1}^{s-q} k^{L-q+1-1} * (k-1) * 2 * (i + q - 1) \\
&+ (k^{s-1-q+1} - 1) * \sum_{i=s-q+1}^{L-q+1} (k^{L-q+1-i+1} - k^{L-q+1-i}) * 4 * (i + q - 1) \\
&+ \sum_{i=1}^{s-q} (k^i - k^{i-1}) * 2 * L + 2 * L \\
&= 4 * k^{L-q+1} * (s + \frac{1}{k-1}) - 4 * k^{s-q+1} * \frac{1}{k-1} \\
&+ (k^{L-q} * (k-1) * (s-q) * (s+q-1) - 2 * L * k^{s-q}
\end{aligned} \tag{4.16}
$$

Therefore, the average relative data recovery latency is:

$$AveRelativyRecoveryLatency_{ALH} = \frac{\sum_{s=1}^{L} RecoveryLatencySum(s)}{k^{L-q+1} * L * 2 * L} \tag{4.17}$$

where k^{L-q+1} is the total number of receivers, L is the number of links that could affect a receiver, and $2 * L$ is the RTT to the sender (same for all receivers).

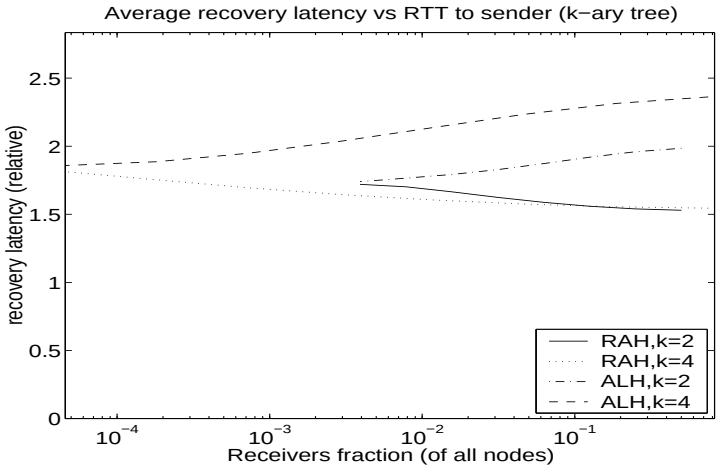


Figure 4.10: RAH and ALH: average data recovery latency.

The results for the data recovery latency latency for binary and 4-ary trees of depth $L = 10$ are in Figure 4.10. From those results we can see that, unlike data and control overhead, there is obvious difference between the RAH and ALH data recovery latency, and this difference increases logarithmically when we increase the number of receivers. On the other hand, the difference does not appear to be very large, and in the worst case the ALH latency is 50% larger (for $k = 4$). Even when we increased the tree depth to $L = 20$, the ALH was within the order of two of the RAH latency. The reason that, unlike the data overhead, the difference between the RAH and ALH data recovery latency is notable

is that when we normalize the latency by the RTT, the result is much more sensitive to a single extra link-hop the retransmitted data may travel. On the other hand, the impact of that single extra link-hop when we compute the data overhead is much smaller when we normalize by the affected subtree size.

4.4 Simulation Results

The analytical results we presented in Section 4.3 apply only given the assumptions we have about the topology and the receivers setup, and may not be true in the general case. Some of the questions we want to answer through numerical simulations are:

- How RAH and ALH perform with real-world router-level topology and how do they compare to each other.
- How other topologies may impact the results.
- What is the impact of the hierarchy creation parameter for the ALH scheme.
- What would be the performance penalty for ALH if we did not use any heuristic to create the data recovery topology (*i.e.,* if the hierarchy was randomly created).

First we will describe our simulation setup, and then will present and discuss the results.

4.4.1 Simulation Setup

In most of the simulations we used a router-level Internet-core topology of 54533 nodes [38]. To investigate the topology sensitivity, we used several other topologies: AS-level map [122, 12], Mbone [123], random graph, mesh, and tree. Some of the characteristics of all topologies are summarized in Table 4.1.

Topology	Nodes	Links	Diam.	Ave. dist.	Ave. fanout
Internet core	54533	146419	23	7.6	5.4
Mbone	4179	8549	26	10.1	4.1
AS	4830	9077	11	3.7	3.8
Random	19596	40094	16	7.2	4.1
Mesh	54756	109044	466	156.0	4.0
K-ary tree (3-ary)	29524	29523	18	16.0	2.0

Table 4.1: Metrics of used topologies.

We assumed a single-source multicast distribution tree with the source at the root of the tree. The receivers were placed according to the client placement methods described in Section: 3.2.3: random, extreme affinity, extreme disaffinity, and extreme clustering. The number of the receives varied as a fraction of topology size between 0.0001 and 0.2 (*i.e.,* 0.01% and 20% of all nodes).[3] The default hierarchy creation parameter for the ALH scheme was 0.1, *i.e.,* on average each parent had 9 children (1/0.1 - 1). Further, to prevent extremely uneven distribution of the children among the parents, the maximum number of children a parent may have at each level was set to $(4 * (1/frac_{pc} - 1))$, where $frac_{pc}$ is the hierarchy creation parameter. Some of the results we present for ALH are both with the inter-receiver distance heuristic we described in Section 4.2.1, and without any heuristic (named ALH-heuristic and ALH-random respectively). In the second case, the set of parents selected at each level of the hierarchy is completely random, and then each child chooses randomly the parent to connect to. The results for ALH-random would eventually give us the worst-case ALH performance, *i.e.,* when we do not have a good mechanism to create the recovery hierarchy. For each set of parameters we performed 50 simulations with

[3]For the smaller topologies the smallest fraction was 0.0002 or 0.001, depending on the topology size.

a different set of receivers.[4] The results we present are averaged across all simulations, but we also present the 95% confidence interval (even though in most cases this interval is very small to be noticed).

For each scheme we measured the data recovery latency, exposure, data overhead, and control overhead across all links going down (one-by-one). For simplicity we assumed that all links have the same propagation latency, and that sending a single packet over any of the links creates the same overhead to the network. The measured results were averaged across all links.[5] The metrics were computed using the expressions in Section 4.2.3.

As we mentioned in Section 4.1, we are not interested in investigating the particular protocols in details, but only in the underlaying schemes instead. For this reason we did not include in the basic schemes various protocol enhancements such as multiple LMS router state for routers with large fanout [88] that can help to reduce the control overhead.

In Section 4.4.2 we present the results for the Internet core topology with random receiver placement. In Section 4.4.3 we present the sensitivity results: ALH hierarchy organization sensitivity (Section 4.4.3.1), topology sensitivity (Section 4.4.3.2), and receiver placement sensitivity (Section 4.4.3.3).

4.4.2 RAH and ALH Simulation Results

Figure 4.11 shows the data recovery latency for RAH and ALH for the Internet core topology and random receiver placement (with and without the hierarchy creation heuristic). First, we can see that the results for RAH did match our analytical results. The fact that

[4] We did some experiments with a larger number of receiver sets but in all simulations there was relatively small variation in the results.

[5] We also experimented with weighting the results by the link loss probability which we assumed is proportional to the number of end-to-end shortest paths that use each link, but the results were similar.

the RAH latency decreases when the number of receivers increase can be explained by the simple observation that a larger number of receivers increases the probability that there is a closer replier that has received the data, and therefore the recovery latency will be shorter. Surprisingly, the ALH-heuristic results were very similar to the RAH results but did not match our analytical results. This can be explained by the fact that in the k-ary trees there is strict enforcement on the recovery hierarchy construction (*i.e.*, a parent can only be a leaf node), while in real-world topologies our heuristic will quite likely choose for each child/receiver its parent node to be reasonably close on the shortest path to the root. It is quite likely that such node will be chosen as a replier in RAH, and therefore the results for both schemes are similar. On the other hand, it is less likely that in ALH-random the parent will be on the shortest path. Hence, when the number of receivers increase, the number of levels in the data recovery hierarchy which do not follow the shortest path between the sender and each receiver will increase as well, and therefore the receiver latency will be longer.

Figure 4.12 presents the results for the receiver exposure. The RAH exposure is always zero by the mechanism definition (true for a single link loss, but may not always be true if there were multiple link losses). The results for both ALH-heuristic and ALH-random are reasonably low. Surprisingly, ALH-heuristic performed worse than the ALH-random. The reason is that in ALH we can have exposure only if the parent uses multicast to send the data to its children. In our simulations the parent would use multicast only if at least 50% of the children did not receive the data. With ALH-heuristic there is larger probability for children locality, and therefore if any of them did not receive the data, there is larger

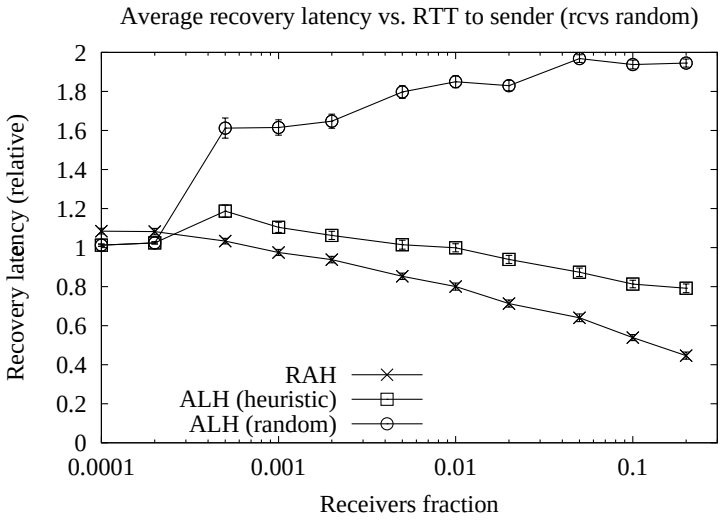


Figure 4.11: RAH and ALH (Internet core): average data recovery latency.

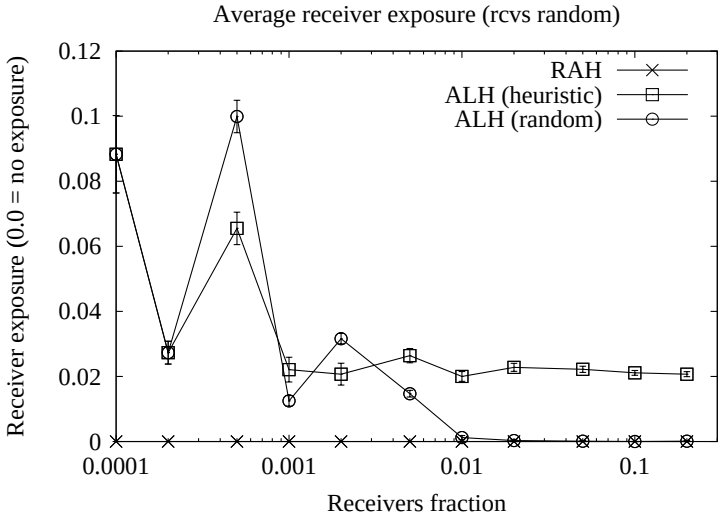


Figure 4.12: RAH and ALH (Internet core): average receiver exposure.

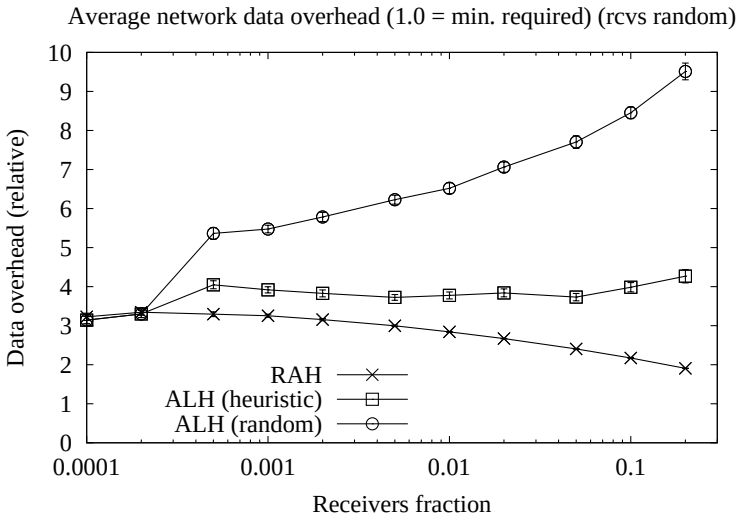


Figure 4.13: RAH and ALH (Internet core): average network data overhead.

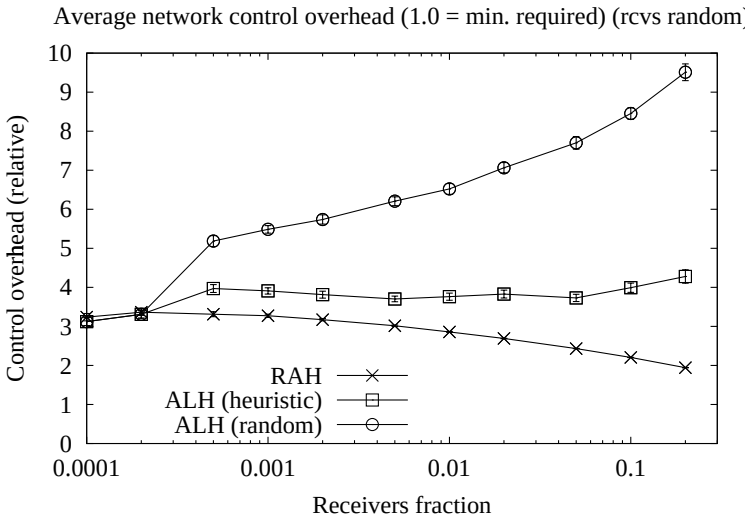


Figure 4.14: RAH and ALH (Internet core): average network control overhead.

probability that at least 50% of its siblings did not receive it either (*i.e.*, larger probability that the parent will use multicast).

Figure 4.13 and Figure 4.14 show the results for data and control overhead respectively. Here again the results for RAH and ALH-heuristic are very similar. However, while the RAH results match the analytical results, it is difficult to say the same thing for the ALH. Similar to the latency, the ALH-random results show that the overhead increases for a larger number of receivers, an artifact from the increased average depth of the data recovery tree.

We should note that for all simulation the data and the control overhead seemed to be almost identical. On closer examination, the RAH control overhead was approximately 5-10% higher than the data overhead. We can explain the reason for this small difference by the fact that there is extra control traffic only over the path between a router-turning point and its replier, a path that by definition is as short as possible for that router, therefore the control overhead is minimized. Indeed, this overhead can be up to $O(RouterFanout)$, but in most cases it did not have a significant impact. For ALH the control overhead was even closer to the data overhead. The reason for this can be explained by the observation that the data overhead can be smaller only if the parent used multicast, but then the gain in some parts in the network may be reduced by the exposure in other parts.

4.4.3 Simulation Results Sensitivity

4.4.3.1 ALH Hierarchy Organization Sensitivity

Figure 4.15 shows the latency results for three different values of the hierarchy creation parameter $frac_{pc}$: 0.02, 0.1, and 0.4.[6] Interestingly, this parameter had almost no impact on the latency (only for a very large number of receivers the results for larger parameter value were slightly better). We believe that the reason for this is as follows. The recovery tree depth would eventually be larger when there is a large number of receivers. However, when the number of receivers increase, there is a higher probability that a parent will be on the shortest path between a child and the root (or at least almost on the shortest path). Then, if all of the parents are on the shortest path, there is no extra latency regardless of the number of intermediate hops to the root.

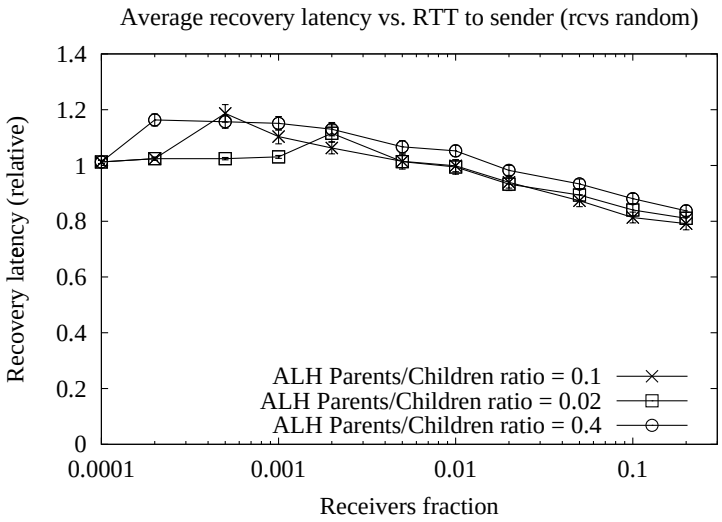


Figure 4.15: ALH: latency sensitivity to hierarchy organization.

[6]Note that for a very small number of receivers and a small parameter value the results are identical simply because the result is always a two-level hierarchy: the sender is the root and all receivers are its children.

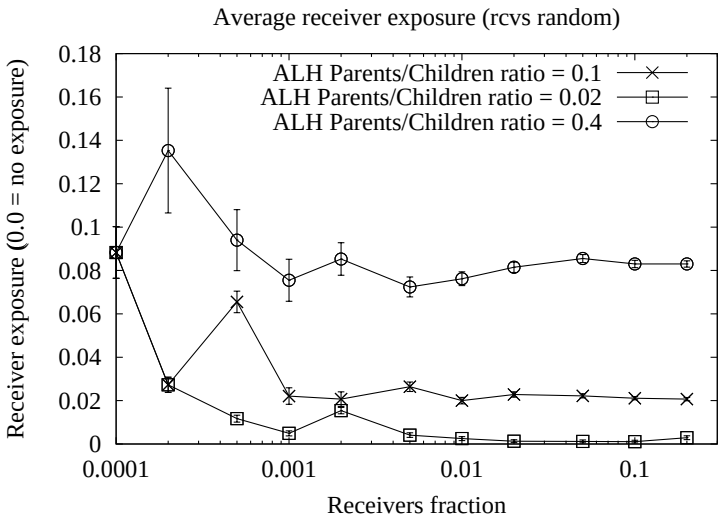


Figure 4.16: ALH: exposure sensitivity to hierarchy organization.

The data and control overhead results (Figure 4.17 and Figure 4.18) do show however, that the overhead is more sensitive to the number of parents a child has to choose from. The higher sensitivity of the data and control overhead compared to the latency sensitivity can be explained by the fact that there is a large number of leaf links (*i.e.*, when the size of the subtree that lost the data is 1), and in all those cases the overhead is much more sensitive to the distance to the parent that eventually has the data. On the contrary, the number of receivers that have very small round-trip time (the basic for comparing the latency), and therefore the distance to their parents may have a larger impact on the result, is much smaller.

From Figure 4.16 we can see that exposure increases when the number of potential parents is larger. The reason for the increase is because of the increased locality among all siblings, and therefore there is a larger probability the parent needs to use multicast to recover the data.

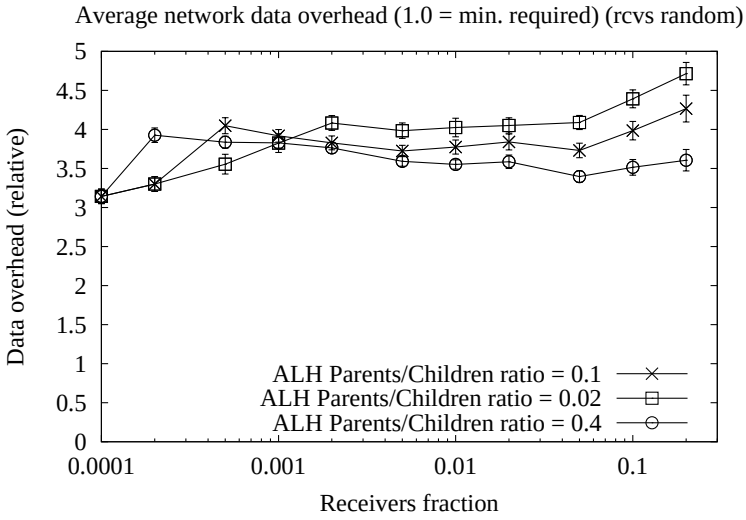


Figure 4.17: ALH: network data overhead sensitivity to hierarchy organization.

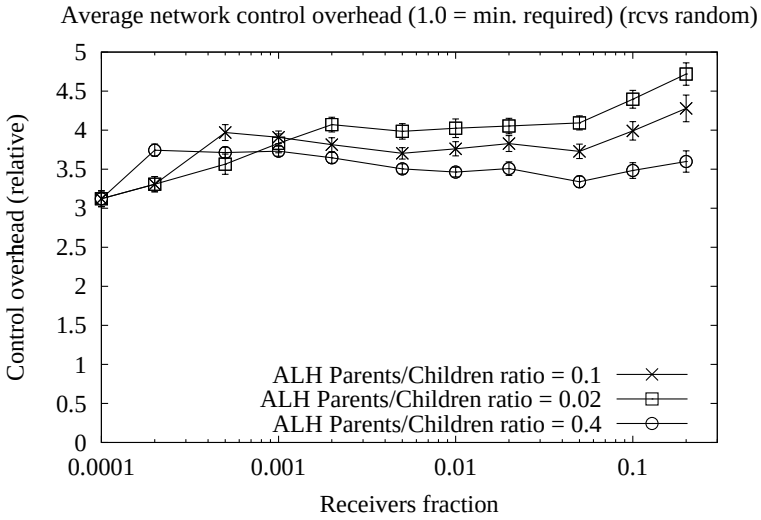


Figure 4.18: ALH: network control overhead sensitivity to hierarchy organization.

4.4.3.2 Network Topology Sensitivity

Figure 4.19, Figure 4.20, Figure 4.21, Figure 4.22, and Figure 4.23 show the average latency for AS, Mbone, random graph, mesh, and tree respectively. All results are with random receiver placement. If we compare those results with the Internet-core results (Figure 4.11), we can see that the results are similar. The only notable exception is the mesh where the difference between RAH and ALH-heuristic is much larger for a large number of receivers. We believe the reason for this is because the multicast distribution tree that is created is composed of long, skinny branches, an artifact of the particular routing in the mesh. Therefore, even small inaccuracy in the parent selection heuristic may have a large penalty in inefficiency.

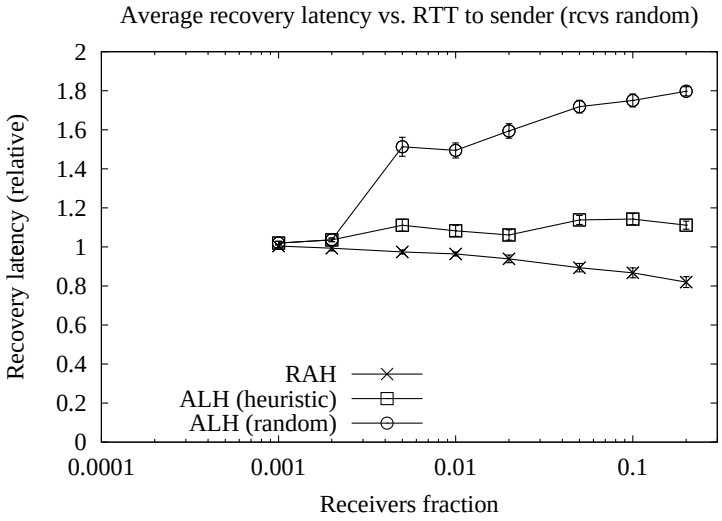


Figure 4.19: RAH and ALH topology sensitivity (AS): average latency.

The results for the data and control overhead were qualitatively similar to the Internet core results, with the notable exception of mesh for which again ALH-heuristic performed

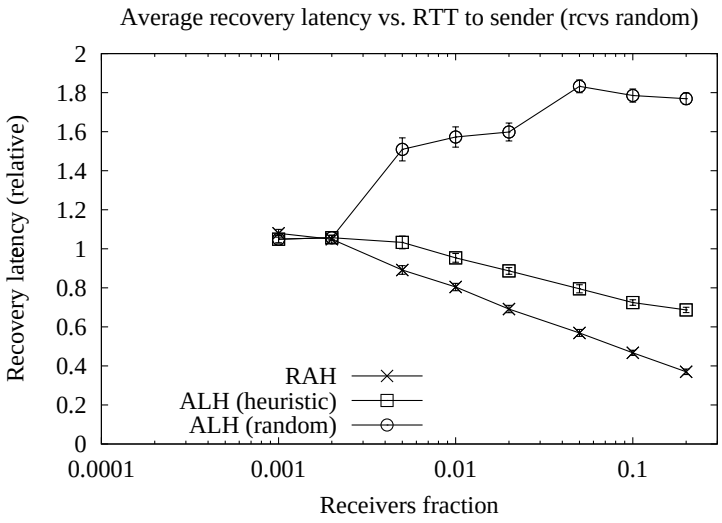


Figure 4.20: RAH and ALH topology sensitivity (Mbone): average latency.

notably worse compared to RAH for large number of receivers. The results for the receiver exposure for all topologies were qualitatively similar to the Internet core results.

4.4.3.3 Receiver Placement Sensitivity

Figure 4.24, Figure 4.25, and Figure 4.26 contain the results for network data overhead for the Internet core topology with extreme affinity, extreme disaffinity, and extreme clustering receiver placement respectively. If we compare those results with the random receiver placement (Figure 4.17), we can see that the extreme affinity and extreme disaffinity results are qualitatively similar to the random receiver placement results. Only in case of extreme clustering placement, the difference between RAH and ALH-heuristic can be on the order of four times and more. The results for the network control overhead were similar to the network data overhead results. The results for the data recovery latency and receiver exposure were similar across all receiver placement models.

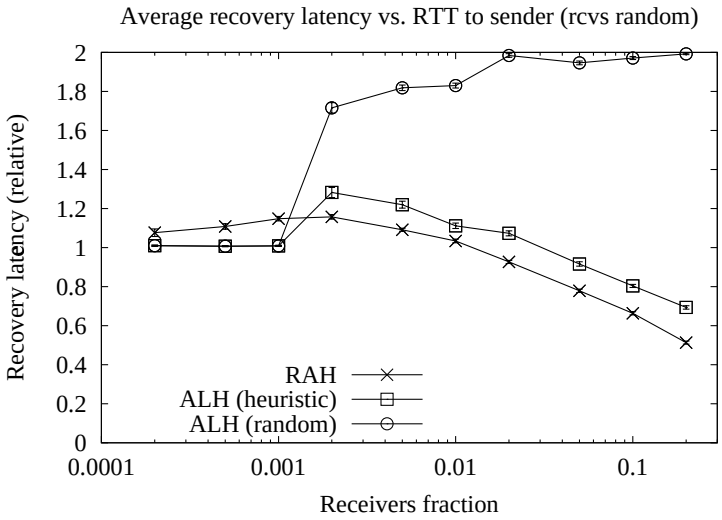


Figure 4.21: RAH and ALH topology sensitivity (Random graph): average latency.

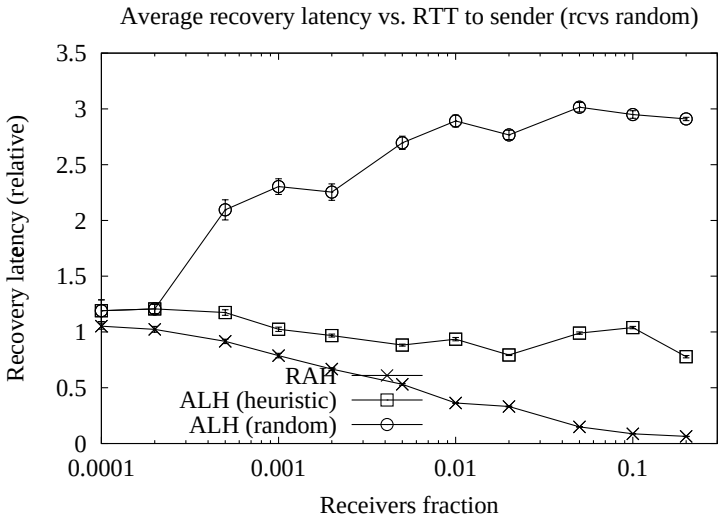


Figure 4.22: RAH and ALH topology sensitivity (Mesh): average latency.

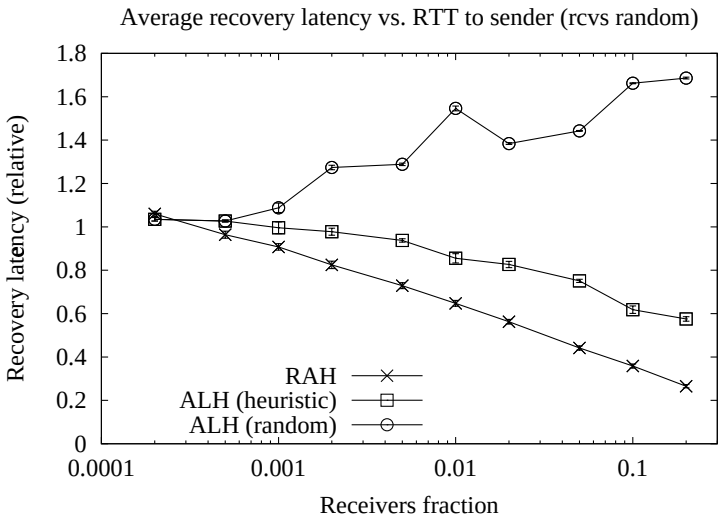


Figure 4.23: RAH and ALH topology sensitivity (Tree): average latency.

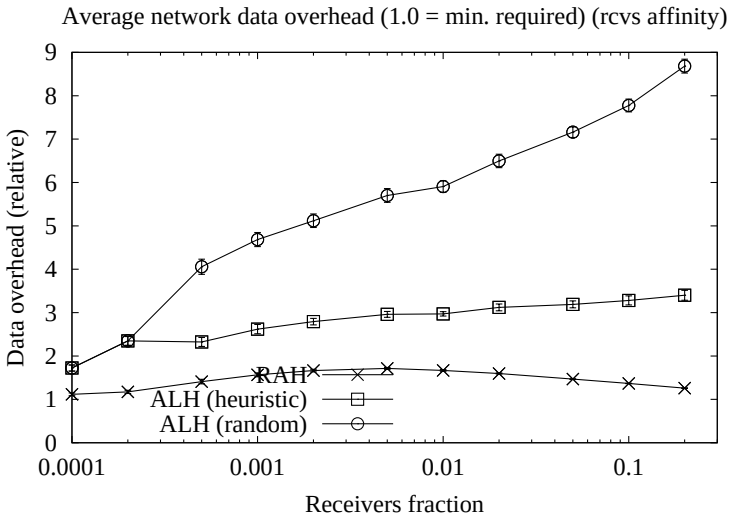


Figure 4.24: RAH and ALH (Internet core, receiver affinity): average network data overhead.

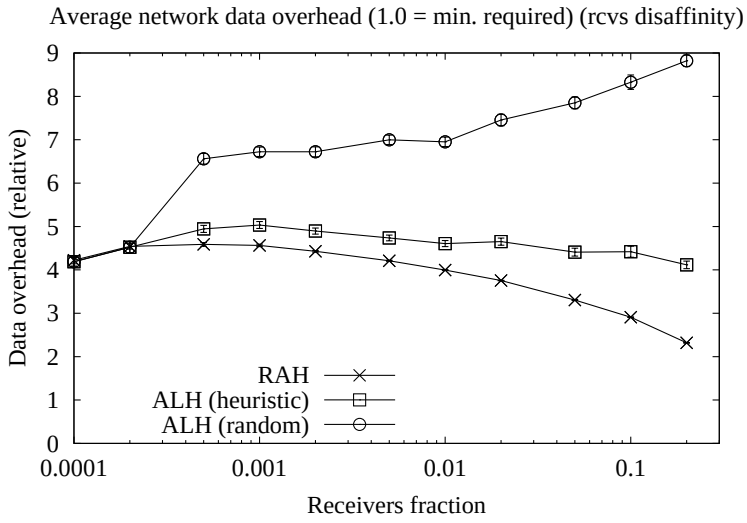


Figure 4.25: RAH and ALH (Internet core, receiver disaffinity): average network data overhead.

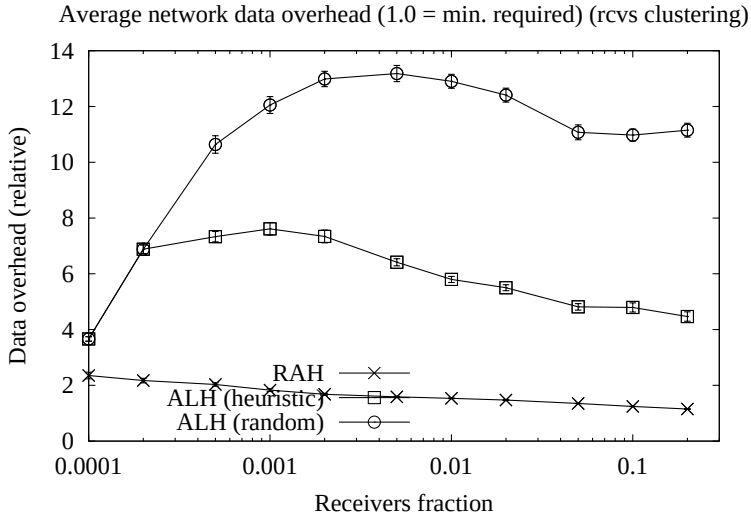


Figure 4.26: RAH and ALH (Internet core, receiver clustering): average network data overhead.

The results for the other topologies were similar to the results with random receiver placement with the notable exception again of the extreme clustering receiver placement, even though for other topologies the difference between extreme clustering and random placement was smaller.

The results from our simulations did show that ALH schemes with a good hierarchy organization can perform within a constant factor of RAH schemes. Further, the ALH performance was not affected by the levels in the hierarchy, but primarily by the parent selection at each level of the hierarchy. The results were similar for all topologies (with the exception of the mesh topology in some cases).

4.5 Conclusions

In this chapter we have studied another problem that can be beneficial from using partial knowledge about the underlying network to improve performance. We analyze and compare the performance of two types for hierarchical reliable multicast data recovery: router-assisted and application-level. We have demonstrated that if the application-level hierarchy is created appropriately, the performance can be comparable to solutions that require the help of the intermediate routers. The difference can be within a factor of 2 or even lower. Further, we show that a heuristic based on the all-receivers end-to-end distances is sufficient to create such application-level hierarchy. The results are valid for a variety of topologies and receiver placement.

Chapter 5

Improving Protocol Performance with End-to-end Mechanisms: Design-case for Application-Level Multicast

In this chapter we consider the problem of using end-to-end mechanisms to adapt the application-level virtual topology and communication pattern to the underlying network topology. We study tree construction and management for application-level multicast. First, we investigate the expected performance of application-level multicast in general. The results from that investigation show that typically the performance of application-level multicast is within a factor of two of the performance of network-layer multicast. Encouraged by those findings, we design and implement a set of adaptive algorithms for tree creation and management, within the framework of Yoid application-level multicast system [37]. By considering both data-loss and delivery latency, we can create application-level trees with reasonable performance. We evaluate the tree performance by both simulations and real-world experiments over the Internet. The results are robust for various topologies and participant placement, which demonstrates that it is possible to achieve reasonable results even if we use only end-to-end mechanisms to improve performance.

5.1 Topology Impact on Endsystem Multicast

Recently, some research efforts have proposed endsystem based multicast schemes [37, 51]. These approaches do not require network support for group communication. Rather, the endsystems (hosts) to which participants are attached conspire to set up a distribution tree comprised of point-to-point links between them. In both schemes, the endsystems continually refine the tree in order to improve the overall delivery latency and reduce network overhead. Endsystem multicast schemes are attractive because of their inherent deployability.

In this section, we study the impact of topology on the efficiency of endsystem multicast schemes. To our knowledge, ongoing research has not considered this issue. Our evaluation is meant to inform, but not resolve, the larger architectural debate about the relative merits of such schemes vis-a-vis network-layer multicast. To analyze the efficiency of endsystem multicast, we define two performance metrics, following [51]:

Tree Stretch is the ratio of links in the endsystem tree to that in a native multicast *shared* tree.

Tree Stress is the maximum number of endsystem tree links that traverse any physical link in the topology.

The stretch and stress metrics depend largely on the particulars of the tree construction methodology. Rather than attempt to faithfully model the methodologies proposed in [37, 51], we consider two simple heuristics. The first heuristic approximately models the initial tree construction procedure in [37, 51], and the second approximates the result of their continual tree refinement procedures.

Closest Receiver This simple heuristic adds new receivers in their join order. Each receiver is connected to the closest node already on the tree whose degree is smaller than some limit. In our evaluation, we choose a limit of 5.

Minimum Spanning Tree This assumes that all receivers are known in advance, as are the corresponding inter-receiver communication costs. Receivers are added to the distribution tree using a Minimum Spanning Tree (MST) algorithm.

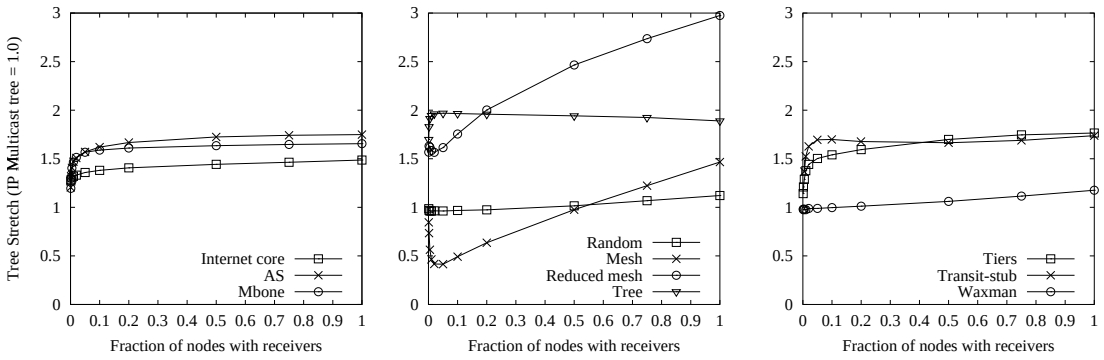
First we describe the results for the closest receiver heuristic, and then for the MST heuristic.

The topologies we use for the evaluation are same as those described in Section 2.3. To compute stretch and stress, we first uniformly select a fraction of receivers in a given topology. We then order the receivers randomly and compute the endsystem tree according to the closest receiver heuristic. To compute the corresponding native multicast shared tree, we randomly select one of the receivers to be the source. We average the stretch and stress thus obtained across a variety of receiver orderings and choices of source. We repeat this steps for different receiver occupancies.

5.1.1 Endsystem Multicast with Closest Receiver Heuristic

5.1.1.1 Discussion of Tree Stretch Results

Figure 5.1 plots the tree stretch as a function of occupancy. The canonical topologies exhibit two kinds of behavior (Figure 5.1(b)). First, in some topologies, the stretch is actually less than 1 at low (about 5%) occupancies. In this category fall the mesh and the random graph, because they, unlike the tree and the reduced mesh, have alternate paths



(a) Real topologies (b) Canonical topologies (c) Generated topologies

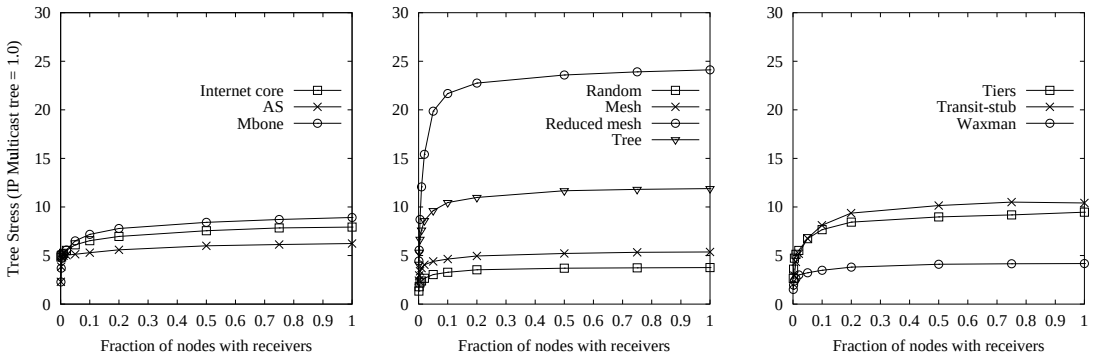
Figure 5.1: Endsystem tree stretch for closest receiver heuristic.

between the receivers. Second, the mesh and the reduced mesh exhibit a dependency of the stretch on occupancy, while the other canonical topologies do not.

For all real topologies, the tree stretch is larger than 1 even at low occupancy (Figure 5.1(a)). Further, their stretch does not depend on the occupancy. In this sense, the stretch of all real topologies lies between a tree and a random graph. Finally, the stretch for all real topologies is less than 1.6, even given that we have a small, fixed limit on the degree of each node in the endsystem tree. In particular, it is interesting to note that the endsystem tree over the Internet core shows low stretch (approximately 1.4), regardless of occupancy. This is encouraging; endsystem multicast schemes are at least not completely unreasonable from an efficiency perspective.

Among the generated topologies, Tiers and Transit-Stub qualitatively match the behavior of the real topologies (Figure 5.1(c)). The same cannot be said of the Waxman network, which behaves more like a random graph.

Finally, although we do not present the results here, we considered a variant of the closest receiver heuristic without any limit on the degree of a endsystem tree node. The tree stretch using our heuristic is about 15% higher than that of this variant.



(a) Real topologies (b) Canonical topologies (c) Generated topologies

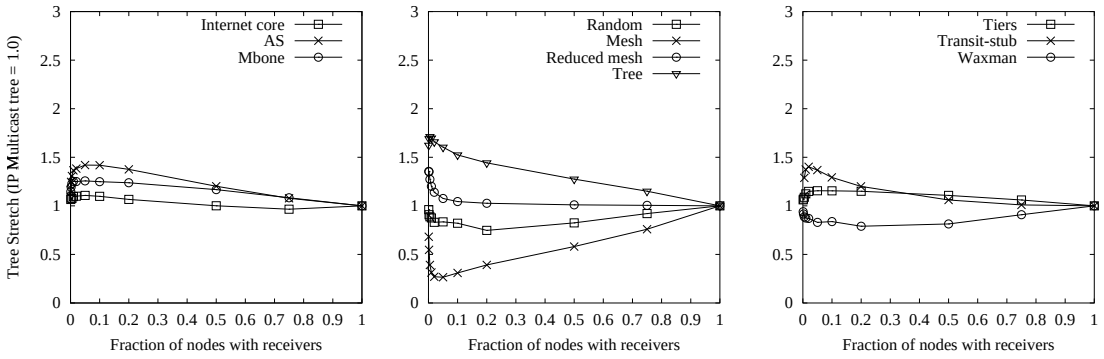
Figure 5.2: Endsystem tree stress for closest receiver heuristic.

5.1.1.2 Discussion of Tree Stress Results

Figure 5.2 plots endsystem tree stress as a function of occupancy. Beyond about 10% occupancy, all canonical topologies are insensitive to the occupancy (Figure 5.2(b)). However, while for the random graph and the mesh the stress is less than or equal to the node degree limit of 5, for the reduced mesh and the tree the stress is noticeably higher.

All real topologies have similar stress, which is close to the degree limit. Compared to the canonical topologies, this places them between a tree and a mesh (Figure 5.2(a)). Among the generated topologies, Tiers and Transit-stub, like the real topologies, are between the tree and the mesh, while Waxman is more closer to the random graph (Figure 5.2(c)).

5.1.2 Endsystem Multicast with Minimum Spanning Tree Heuristic



(a) Real topologies (b) Canonical topologies (c) Generated topologies

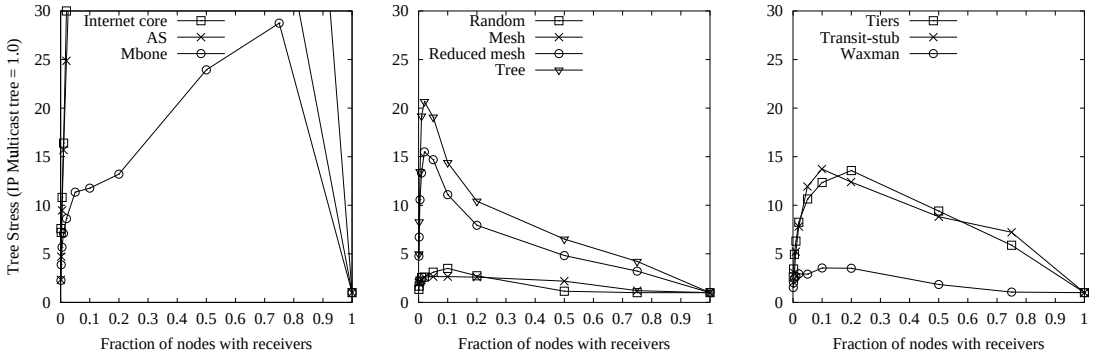
Figure 5.3: Endsystem tree stretch for minimum spanning tree heuristic.

In Section 5.1.1 we presented the tree stretch and stress results for the closest receiver heuristic with limiting node degree heuristic. Here we describe the results for the Minimum Spanning Tree (MST) algorithm.

Figure 5.3 shows the tree stretch as a function of occupancy. At an occupancy of 1, the number of links in the MST is $N - 1$, as is that in the IP multicast tree. In the limit, then, stretch is 1. The mesh and random graph have stretch less than 1 at low occupancy such as mesh and random. On the other hand, the tree and the reduced mesh have stretch higher than one, as do the real topologies.

If we compare the MST stretch with the closest receiver tree stretch in Figure 5.1, we can see that for low occupancy (20% or less) the results are qualitatively the same (*e.g.,* the topologies with low closest receiver tree stretch have also low MST stretch, and vice versa). Quantitatively, the closest receiver tree stretch is about 30% higher compared to the MST. For higher occupancy this difference increases. It is interesting to note that for all real

topologies the difference in stretch between the closest receiver and the MST heuristic for low occupancy is remarkably low (on the order of 20–30%).



(a) Real topologies (b) Canonical topologies (c) Generated topologies

Figure 5.4: Endsystem tree stress for minimum spanning tree heuristic.

Figure 5.4 plots the MST stress as a function of occupancy. All real topologies (Figure 5.4(a)) have very high stress, and this depends significantly on the occupancy. At first glance, it may seems that the real topologies are in the same category as the tree and the reduced mesh. However, after some more careful investigation, we found that in all cases the high stress of the real topologies was because of some node with very large node degree.[1]

In summary, the MST stretch improvement over the closest receiver heuristic is relatively small for low occupancy, and is on the order of 20–30%. However, because for some real topologies the MST heuristic can induce high stress, this makes it impractical.

[1]The highly overloaded link was between such node and some of its physical neighbor-receiver, because such neighbor is potentially closer to many other nodes, and therefore it will have much higher degree on the endsystem tree.

5.2 Introduction to Yoid Application-Level Multicast System

The recent interest in application-level multicast [51, 37, 90], or *endsystem* multicast [98] as it is sometimes known, is motivated by the disappointing delays encountered in deploying network-layer multicast [30]. Application-level multicast is an attractive and deployable alternative, because it requires no support from the underlying network. Rather, application software automatically creates an overlay distribution tree spanning all the participants of a multicast group and forwards application data over this tree. This software also adapts the distribution tree to participant dynamics (joins and leaves) and network failures.

We see these application-level systems as being medium-term alternatives, while IP multicast deployment issues are ironed out, for small group collaboration, and for medium-sized lecture style applications. More generally, as [37] argues, this kind of distribution applies not just to traditional multicast applications (like shared whiteboards, and audio and video conferencing), but also to most forms of content distribution. For example, one can imagine a Content Distribution Network's replication network to be built upon such an overlay [14]. It might also be feasible to implement peer-to-peer file sharing systems on top of such overlays.

In this work, we discuss the design of networking mechanisms for one kind of application-layer multicast system called *Yoid* (Your Own Internet Distribution). We begin by briefly describing the Yoid architecture in the next few paragraphs.

Similar to IP multicast ([4, 33]), a rendezvous mechanism is an important part of the Yoid system. Participants in a Yoid group rendezvous through a shared logical label for the group. In IP multicast, the logical label is a globally unique IP address. In Yoid,

however, the logical label for a group syntactically resembles a URL and encodes the name of a *rendezvous host*, a port number that the rendezvous host is listening on, and the name of the group which is unique to the rendezvous host. Thus, a Yoid label of the form `yoid://foo.bar.org/wb:5555` denotes a Yoid group named *wb*, whose rendezvous host name is `foo.bar.org`, and that host is listening on port 5555. Because group names need not be globally unique, Yoid does not require global coordination in group label assignment [66].

When a participating host (henceforth, a member) wishes to join a Yoid group, it contacts the rendezvous host (or simply, the rendezvous). From the rendezvous, it obtains its own node ID that is unique within that group, as well as a list of some (not necessarily all) current members. These are the host's *candidate-parents*. The host then uses this list to graft itself, in a manner described later, to the tree topology that is used to distribute application content. The rendezvous is responsible for keeping its list of members current, by explicitly checking for their liveness. The rendezvous does not participate in application content forwarding. It, however, plays a key role in other functions, such as partition healing and group security; these are discussed more fully in [37]. This design of a centralized rendezvous host is an obvious impediment to scaling and robustness. The scaling drawbacks can be alleviated by carefully designing the rendezvous, as Yoid does, so that members only contact the rendezvous initially or from time to time, and the rendezvous only maintains state about a subset of the members. Dealing with robustness is slightly more complicated and described in [37]; we note, however, that failure of the rendezvous only affects the joining of new members, and does not impact the existing tree.

As alluded to earlier, Yoid essentially constructs a *shared-tree* overlay. Each node on this shared tree is an endsystem on which a group member resides, and each link is a tunnel between two members. Such shared-tree overlay construction is not the only way to achieve application-layer multicast. Another approach, which may be called *mesh-first*, has new members establish *mesh links* to several current members. Over this mesh, all members run a link-state or distance-vector routing protocol. Using this, members can conspire to construct distribution trees rooted at each source of data (*i.e.,* source-specific trees). This is the approach followed in [51].

Despite the superiority of source-specific trees over shared trees for IP multicast in term of source-destination data propagation latency and data traffic concentration, we contend that Yoid's design choice of a shared-tree *overlay* is actually reasonable, even for latency and loss-sensitive applications like audio and video conferencing. First, source-specific application-level trees are very sensitive to actual member placement, and may need fairly sophisticated metric tweaking in order to work [17]. Given this, it isn't immediately obvious that source-specific application-level trees are necessarily better than shared trees. Second, most commercial server-based H.323 conferencing systems [42] in use today are essentially shared trees. This is at least an existence proof that shared application-level trees are not a completely unreasonable design choice.

On the shared tree, application content is *flooded*. That is, each member receiving a frame of application data from a neighbor forwards a copy of this frame to each of its other on-tree neighbors. Yoid defines a fairly complete application-level protocol stack that allows the transport of application content over the tree. These protocols run over UDP or TCP and perform several functions. The Yoid Identification Protocol (YIDP) identifies

the group that a Yoid frame belongs to, and the sending members. Sitting on top of YIDP is the Yoid Distribution Protocol (YDP), which provides framing, push-back flow control between tree neighbors, and sequencing. These distribution protocols are not the focus of this work, but they are described in more details in [37].

This work explores mechanisms for constructing and maintaining the Yoid shared tree. How does the Yoid shared tree actually get built? The first member to join the group is designated by the rendezvous to be the *root* of the tree. Each subsequent joining member contacts the rendezvous and obtains a list of current members. The joining member then selects one of the current members to be its *parent*. The choice of parent can be dictated by performance considerations. For example, a member might choose the topologically closest current member as its parent, if this can be determined (*e.g.,* based on heuristics such as IP address prefix). Clearly, the choice of parent crucially determines overall perceived performance; we will return to this subject later. A member is not responsible for finding *children*, although it may reject another member that requests to be its child. When a member loses connectivity to its parent, it attempts to contact other members in order to select a new parent. When the member switches parents, its relationship with its offspring is unaffected.

While this tree construction protocol is simple and requires little inter-member coordination, it presents two challenges.

First, this distributed tree construction is susceptible to loop formation (see Section 5.3.1 for an example). To deal with the possibility of loops caused by our simple, localized tree grafting algorithm, we do not use loop *avoidance* techniques. Intuitively, loop avoidance may require *a priori* knowledge of the overlay topology, or dissemination of

"routing" information. Yoid, however, starts off with a very simple method to graft nodes onto the overlay tree that doesn't require running a routing protocol. In keeping with this design, we use a novel loop *detection* mechanism, together with a technique for fast loop termination (Section 5.3.1).

Second, the tree construction algorithm described above largely ignores issues of performance. In particular, because Yoid builds overlays using hosts, it can be susceptible to performance degradation that arise from poor choices of topology.

To take a concrete example, consider a host that is behind a limited capacity broadband connection (such as a DSL line). If this host's fanout in a Yoid tree is N (one parent and $N-1$ children), it requires a bandwidth of $N * R$ where R is the application content data rate (*e.g.,* an audio stream). If this bandwidth exceeds the capacity of the host's connection to the network, it adversely affects the perceived performance *at all hosts whose on-tree path to the source traverses this host.* One obvious way to avoid this problem is to have a static fanout limit at each Yoid host. This approach is undesirable because it may require manually configuring the fanout based on a host's network connection speed, and also because the available bandwidth can vary dynamically. Accordingly, Yoid dynamically *refines* the tree based on observed data losses. In Yoid, each host compares (see Section 5.3.2.2 for details) its loss fingerprints (the specific Yoid data frames lost within a fixed window), called *lossprints* later in the text, with those of its neighbors and, if the lossprints differ significantly, the host decides whether it should terminate the connections to some of its on-tree neighbors (for example, to reduce its own fanout, or to avoid lossy links). This kind of topology adaptation is unique to application-level infrastructures. It is also complementary to *congestion control* mechanisms that adapt the sending rate of the

audio streams to the capacity of the tree. We do not explore such mechanisms in this work, but they could, in principle be designed by combining ideas from RAP [101] together with some sort of push-back flow control [37].

To take another concrete example, consider a host behind a high latency network connection, such as a phone-line modem. If the tree construction algorithm results in this host being near the root of the tree, all hosts downstream of this host with respect to a given source will observe high latency data delivery. For real-time audio and video conferencing applications, clearly, this is a problem. Yoid deals with this kind of performance degradation by dynamic refinement as well. Specifically, Yoid hosts occasionally *test* new parents to see if they can consistently deliver Yoid data frames at *significantly* lower latency, then switch to these parents (Section 5.3.2.1).

Although we have described the latency and loss-rate refinement algorithms separately, they are intended to work together to balance reasonable latency performance with low loss-rate. We illustrate how these algorithms work together in our evaluations in Section 5.4.

In Section 5.4.1 we use simulation to validate the loop-detection algorithm. We also use simulation to verify, using a scenario-based approach, the design of the tree refinement algorithms. In addition, we have also implemented a Yoid library to which we have ported several applications including *wb* [57], *vic* [78], and *rat* [99]. We also present results from experiments with our implementation (Section 5.4.2).

The importance of our work lies in the context of emerging interest in self-configuring peer-to-peer distribution systems. In these systems, loop-free rapid topology repair and

adaptation to the physical topology will be recurring problems. This work is, to our knowledge, one of the first to explicitly consider these issues, and present simple, implementable distributed algorithms for these problems.

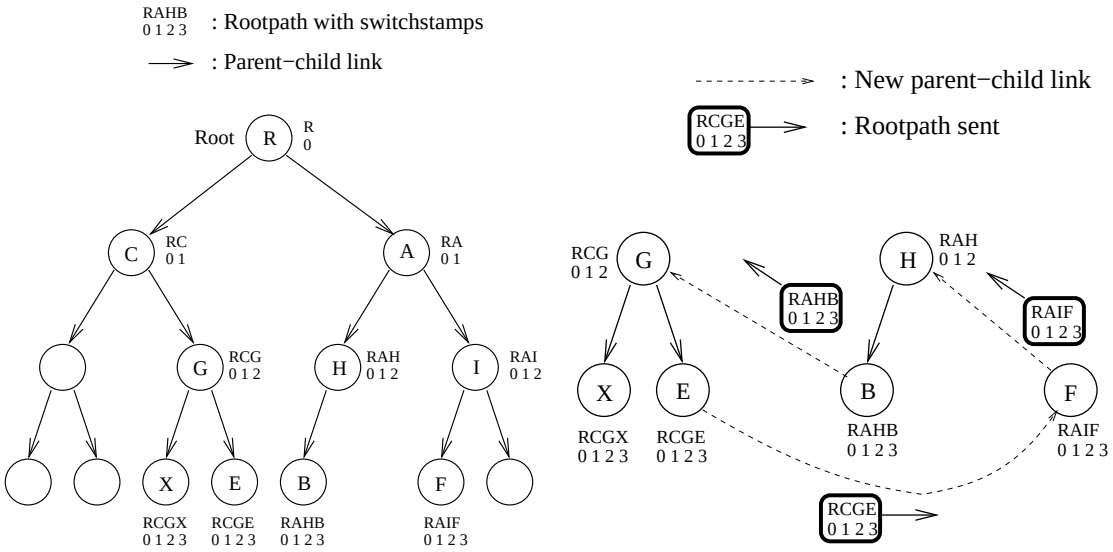
5.3 Tree Management Algorithms

In this section, we describe our designs for the two mechanisms necessitated by our tree-first approach to application-level overlay construction. We first describe our loop detection and fast termination techniques (Section 5.3.1). Then, in Section 5.3.2 we discuss our latency and loss-rate *tree refinement* techniques.

5.3.1 Loop-Detection Algorithm

Recall that Yoid's tree construction algorithm is conceptually very simple. In Yoid, each node that wants to graft itself onto the tree issues a *join* request to an on-tree node. The latter node becomes its parent. A node that detects a failed parent uses the same mechanism to find a new parent. Finally, for reasons we describe later, Yoid nodes also use this mechanism to *switch* parents. In all cases, nodes select prospective parents based on information obtained from the *rendezvous*.

If each on-tree node maintains the list of Yoid nodes between the root of the tree and itself (a list which we call *rootpath*), there exists a simple loop avoidance technique: a node accepts a join request only if the originator of that request is not in the rootpath of that node. If the join is accepted, the parent node's rootpath is sent to the new child. The child adds itself to the end of the rootpath it receives from its parent and forwards it to all of its children.



(a) Original Yoid tree.

(b) Loop formation: F selects E as a new parent, H selects F, G selects B.

Figure 5.5: Example of loop formation.

This simple join acceptance rule does not guarantee that there are no loops at all, but is sufficient to prevent most loops. However, if two or more nodes that are the roots of different subtrees select new parents at approximately the same time, this rule alone is insufficient to avoid a loop.

For example, consider the Yoid tree in Figure 5.5(a). If, for some reason, node F joins E as its new parent, and at the same time H joins F and G joins B, then a loop involving nodes $EFHBG$ is formed (see Figure 5.5(b)). At the instant the nodes join new parents, the parents' rootpath may not indicate a loop, so the loop avoidance rule described above doesn't prevent this loop. However, a possible solution follows from the following observation. A switch to a new parent triggers the propagation of the new rootpath over the subtree rooted at that node, therefore if there is a loop the rootpath is propagated back to the same node that originated it. If a node receives a rootpath that already includes

that node inside, this clearly indicates the existence of a loop and forms the basis of Yoid's loop *detection* rule.

Loop *termination* (*i.e.,* breaking the loop after it is detected) is non-trivial, however. A simple mechanism to deal with loops would be for a node to switch immediately to a new parent after it detects a loop. For example, after F, G and H discover the loop, each of them them would disconnect from its parent (E, B and F respectively), and then would try to join toward a new parent. However, such actions could result in a significant transient tree reconfiguration activity. Such activity can result in degraded application performance, since Yoid frames might be dropped while tree links are in flux.[2] Furthermore, there is no guarantee that the new configuration does not contain a loop, so this solution may result in increased tree convergence time.

We can improve the convergence if, instead of all nodes breaking the loop, only one of them is selected to break it. The difficult question, then, is how to have all nodes on the loop agree on which node terminates that loop, without introducing extra control messages or extra latency.

The solution we propose is to augment the rootpath with a small amount of additional per-hop information. Specifically, each node has associated with it a so-called *switchstamp*. The switchstamp is an integer that is initialized to zero for each node on startup, and is associated with that node's ID in the rootpath messages. When a node receives the first rootpath message from a new parent, the switchstamp for that node is set to be greater than any of the switchstamps of the nodes in the new rootpath. In other words, the switchstamp

[2]Yoid, being an application-level system, can reduce such data loss by buffering application content. However, depending on how these buffers are provisioned, such losses may still happen.

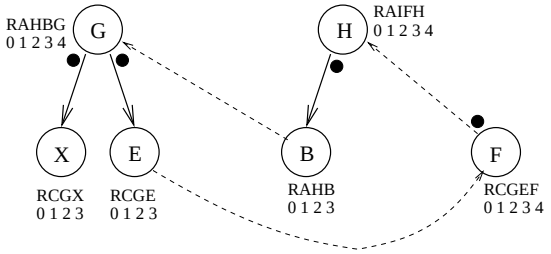
is never decremented,[3] and is never modified by later rootpath messages coming from the same parent.

The switchstamp is used to help identify the node that should terminate the loop by selecting a new parent. Recall that from any rootpath message, each node can locally determine the IDs and the switchstamps of all nodes on the loop. If a node receives a rootpath containing a loop, the node compares the switchstamps of all other nodes with its own switchstamp, and if it has the largest one (with tie-breaking based on the largest host ID), then it selects itself as the loop terminator. Otherwise, the node forwards the rootpath down to its children, but does not accept new join requests until it receives a loop-free rootpath message from its parent, an indication that the loop has been resolved.

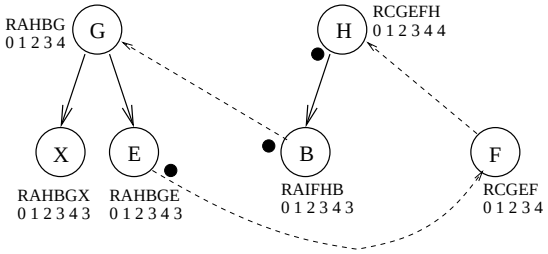
For example, Figure 5.5(a) shows the initial rootpaths with the switchstamps for some of the nodes on the tree. Figure 5.5(b) shows the loop that has been formed and the rootpaths that are sent right after F, G and H have attempted to switch to new parents simultaneously. The figures also show the internally stored rootpath information for those nodes. The hop-by-hoop rootpaths propagation is illustrated in Figure 5.6(a), Figure 5.6(b), Figure 5.6(c), and Figure 5.6(d). Figure 5.6(e) shows what all rootpaths look like after the three messages originated by F, G and H have traversed the loop to reach G, H and F respectively. The switchstamps of F, G and H are the highest among all nodes on the loop therefore one of them has to break the loop. If we assume that based on the host IDs H is the winner, then H should terminate the loop by disconnecting from its current parent F, and should attempt to find another parent (not on the loop) from among

[3]We assume that the switchstamp space is large enough that it never wraps around within the duration of a single Yoid session.

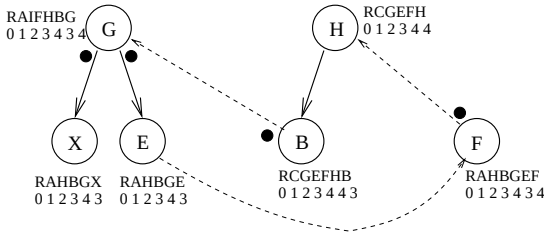
● : Rootpath update will be sent



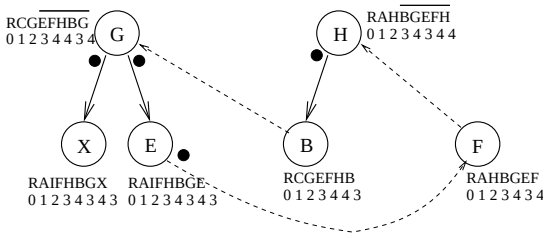
(a) The new rootpaths are propagated one hop away.



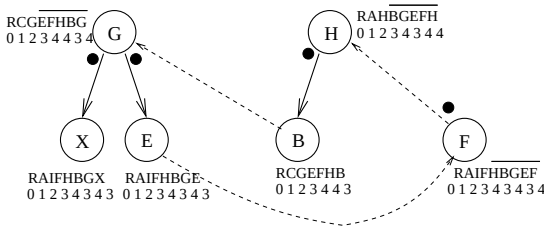
(b) The new rootpaths are propagated two hops away.



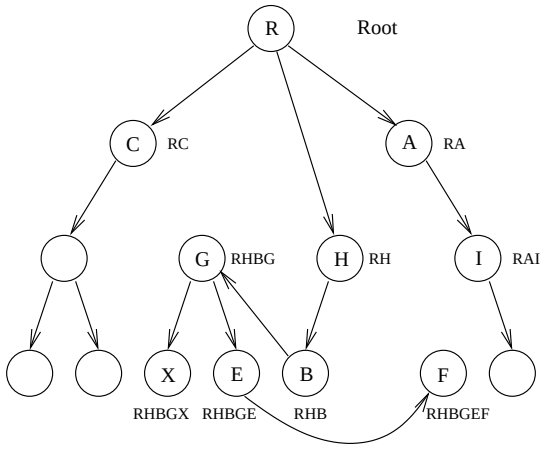
(c) The new rootpaths are propagated three hops away.



(d) The new rootpaths are propagated four hops away.



(e) All nodes that switched to a new parent have discovered the loop.



(f) Final tree after loop termination (H breaks the loop by joining toward R).

Figure 5.6: Example of loop discovery and termination.

its set of candidate-parents. In the mean time G and F will forward their rootpaths down the loop so even if the rootpath message originated by G did not reach H, then the one originated by H itself would reach H.[4] Finally, Figure 5.6(f) shows what the tree looks like after H has broken the loop and has joined the rest of the tree at node R.

The pseudo-code of the algorithm is relatively simple and is included below.

Loop detection algorithm pseudo-code

```
// Algorithm for loop-detection
rootpath_recv(rootpath)
{
  // Check for loops
  loop_found = FALSE;
  FOREACH entry IN rootpath {
    IF my_id == entry.id {
      loop_found = TRUE;
      loop_start = entry.next;
      BREAK;
    }
  }
  IF loop_found {
    i_break_loop = TRUE;
    IF new_parent
```

[4] In an earlier version of the algorithm, a rootpath was not forwarded if it contains a loop. However, when we tried to prove the algorithm correctness, we found that in some complicated scenarios this may create a deadlock, hence we had to fix the problem by modifying the rootpath forwarding rule.

```
    GOTO loop_break;
  // Compare switchstamps
  FOREACH entry IN loop_start {
    IF my_sstamp < entry.sstamp
      || my_sstamp == entry.sstamp
         && my_addr < entry.addr {
            i_break_loop = FALSE;
            BREAK;
    }
  }
  loop_break:
  IF i_break_loop {
    // Join toward a new parent
    ...
    RETURN;
  }
  dont_accept_joins = TRUE;
  GOTO send_rootpath;
}
// A loop-free rootpath
dont_accept_joins = FALSE;
// Set my switchstamp
IF new_parent {
  FOREACH entry IN rootpath {
    IF my_sstamp <= entry.sstamp
      my_sstamp = entry.sstamp + 1;
```

```
    }
  }
  send_rootpath:
  append(rootpath, me);
  FOREACH child IN my_children
    send(child, rootpath);
}
```

The algorithm properties can be summarizes by the following theorem.

Loop Detection and Termination Theorem

Theorem 1. *If two or more nodes using the loop-detection algorithm described in Section 5.3.1 select simultaneously new parents such that a loop is formed, then one and only one of those nodes is implicitly elected to break the loop. The amount of time for that node to discover that it has to break the loop is no longer than the on-loop hop-by-hop Rootpath message propagation time.*

Proof. A Rootpath message is forwarded by a node to its children as long as it does not contain a loop, or if this node is not the winner that should break the loop. Therefore, in the worst case the Rootpath message that was originated by the winner itself will come back to that node so the winner will initiate the loop break, and the latency is equal to the on-loop hop-by-hop Rootpath message propagation time. If there was a long-lasting network partitioning or other nodes on the loop switched to a new parent, the winner may not receive that Rootpath message; however, such events will effectively break the loop (*e.g.,* by neighbor timeout in case of network partitioning).

If no on-loop node changed its parent after the loop was formed, then the winner on that loop remains the same node during the lifetime of the loop. Hence, the Rootpath messages propagation effectively synchronizes the rootpath information for all on-loop nodes, and only one of those nodes is recognized as a winner. If, due to membership dynamics, a node switches several times its parent, it may effectively increase its own switchstamp, therefore the absolute winner may change while the Rootpath messages are in transit. This change however will not result in several nodes declaring themselves as winners and initiating loop break. The reason for this is that even if a future winner node W forwarded first a Rootpath message by another on-loop node before originating its own Rootpath message, the first Rootpath message will effectively break the loop when received by some other node. Therefore, the Rootpath message from W cannot complete the loop traversing and therefore "declare" W a winner as well.[5]

Finally, the reason that only a node that had switched a parent is elected to break the loop is because when a node switches to a new parent, the new switchstamp of that node is always larger than the switchstamps of the nodes above it (toward the root of the tree) that did not switch to a new parent. Therefore, a node that did not switch a parent will not have large enough switchstamp to become a winner to break the loop.

□

Obviously, loop termination could have been achieved in a simpler manner, *e.g.*, by selecting the node with the highest ID. However, such algorithm does not have the following desirable property. Within a loop, nodes that did not initiate a switch to a new parent

[5]The assumption here is that the Rootpath messages are not reordered while in transit from a parent to a child which can easily be achieved when we use reliable transport protocol such as TCP to transmit the control messages between any two nodes.

right before the loop was formed (*i.e.*, nodes that did not "cause" the loop), should not be the ones to terminate the loop, if possible. This property is desirable because it does not impact long-lasting virtual links that have already adapted (using the refinement techniques discussed below) for good performance. The switchstamp-based node election attempts to more carefully select the node that terminates the loop.

5.3.2 Tree Refinement

Yoid's tree construction algorithm allows nodes to graft themselves onto the application-level overlay simply by joining an existing tree node (the parent). Clearly, the choice of parent is oblivious to performance. As discussed in Section 5.2, this choice can lead to *degraded* trees—those that are susceptible to sustained *data loss* because a node's tree fanout exceeds the available bandwidth, or those that experience *high latency* because nodes behind high-latency links are placed high up in the tree. Either of these scenarios can adversely impact application performance, particularly that of audio and video applications.

In keeping with its overall design, Yoid attempts to iteratively and adaptively *refine* its tree in an attempt to avoid such performance degradation. Specifically, Yoid's approach relies on *local* observations of data loss and latency. If a node observes high loss or high latency, it unilaterally decides to correct the situation by *switching parents*. In this section we describe these tree refinement algorithms. Before we do so, it is worth emphasizing that our goal in tree refinement is not to optimize performance, but to avoid unacceptable scenarios that affect notably the performance. Our local refinement techniques are well

suited to this approach; optimizing tree performance might require running a routing algorithm over a richer overlay, as [51] does. Our sense, based on our experience with the Yoid software, is that avoiding unacceptable scenarios can get us a long way toward having reasonable levels of performance. It is also worth pointing out that there is some tension between loss-rate refinement, and latency refinement. For example, one can construct a tree with low loss-rate by ensuring a chain topology, but this has poor latency characteristics. Below we describe how we address these issues.

5.3.2.1 Latency Refinement Algorithm

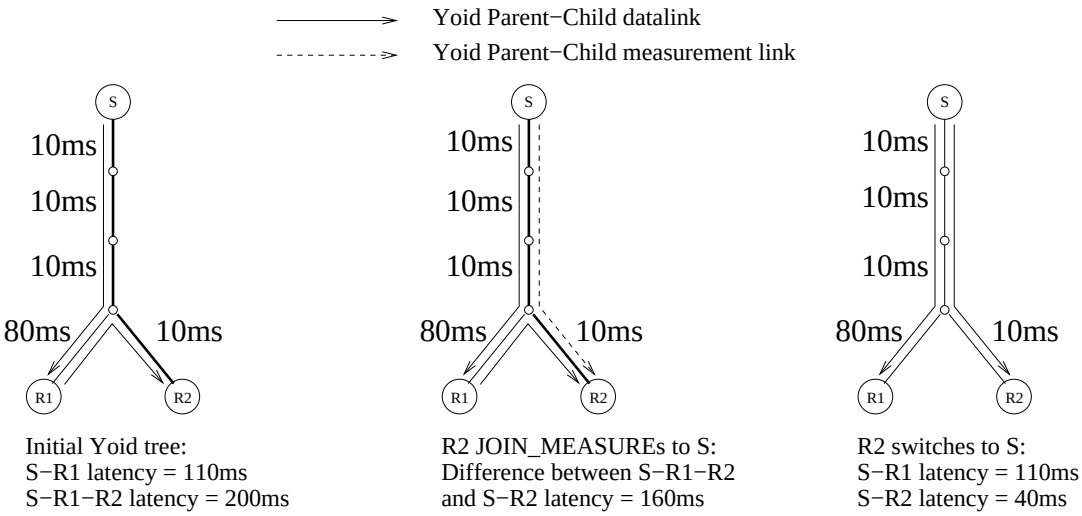


Figure 5.7: Latency refinement algorithm: R2 measures the data frames latency difference between S and R1 and then switches to S.

The algorithm for avoiding latency degradation is relatively simple. Recall from Section 5.2 that each node obtains from the rendezvous a list of nodes which it can choose

as candidate parents. The list can be periodically updated by explicitly querying the rendezvous to obtain information about new members. Among the nodes on the candidate-parents list, each node selects a subset as *active candidate-parents* by opening a *tentative* Yoid link[6] to each of them. Over a tentative link, an active candidate-parent forwards a sample (one in N, where N is a tunable algorithm parameter) of data frames to the node. Then, the node timestamps the exact time when the frame arrives, but does not forward data frames received over tentative links. It also timestamps the same data frame when it arrives on the data distribution tree.[7] If, averaged over some number of data frames, [8] the latency difference between data frames received on the tree and data frames received from an active candidate-parent is above a certain threshold, the node switches to the new active candidate-parent. In addition, the node also disconnects from its worst active candidate-parent and tries to open a connection to another candidate-parent; this ensures that over time a node can explore many candidate-parents and will eventually find the most appropriate. Our approach is similar in spirit to the multicast join experiments for detecting congestion levels [79], and the sensor net reinforcement algorithms for finding the best performing path [54]. Because our focus is on avoiding degradation, and not optimizing performance, it is reasonable to expect that the node will converge quickly on a "good enough" parent. Indeed, our simulations bear this out.

[6]These tentative links also allow Yoid to quickly recover from parent failures.

[7]Each data frame has an unique (per sender) ID, therefore the IDs of the data frames that need to be timestamp can be computed from the pre-defined N.

[8]If there are several senders, the latency difference is averaged among them. One alternative is to try to optimize for the notably worst sender, but that solution may result in oscillations.

The algorithm can be understand better with an example. Figure 5.7 is an example of how the data delivery latency from S to $R2$ can be improved from 200ms (the Yoid tree on the left) to 40ms (the Yoid tree on the right) by avoiding the 80ms long-latency edge link.

The algorithm has several parameters, the settings of which determine its efficacy. The first is the latency difference threshold which determines whether an active candidate-parent is desirable. Because our goals is to avoid worst-case placements, such as tree links that cross transcontinental links twice, or tree links that span dialin lines, we choose a relatively large value for this threshold (50ms). This choice is in keeping with transcontinental propagation delays of tens of milliseconds, and dialin line equalization latencies and scrambling/descrambling latencies of the same order. Two other parameters that affect how quickly nodes converge to acceptable latencies are the sampling rate of data frames over tentative links, the number of active candidate-parents, and the number of nodes for which a node can be an active candidate-parent. We choose 1-in-100, 5 and 5, respectively for these, based on our simulations. This choice limits the overhead of the algorithm to about 5% of overall data traffic.[9]

Even though the algorithm is designed to eliminate the impact of single long-latency links, in practice it also reduces the depth of the tree by short-cutting a long chain of upstream nodes, therefore effectively reducing the average end-to-end latency as well.

5.3.2.2 Loss-rate Refinement Algorithm

The basic idea behind the loss-rate refinement algorithm is that each node monitors its data losses (based on the recently received data frame IDs), and if the losses are above

[9]Strictly speaking, an active candidate-parent does not need to send the whole data frame, but only the sender ID and the data frame ID, therefore the overhead can be even lower.

a threshold, it joins toward a new parent (or forces a child node to find a new parent if the data is received via that child). However, this mechanism alone is insufficient, since it does not detect which link is responsible for the losses. Thus, a lossy link close to the sender may result in all downstream nodes trying to switch to new parents. Furthermore, if the problem was because of a congested edge link of a parent node with a large number of children, then all its children may try to switch to new parents, when it may suffice for only one or few of them to switch parents.

The solution we propose is for each node to coordinate with its neighbors in order to determine the location of the loss, and decide who should switch parents. To achieve this, each node periodically exchanges *lossprint* information with its neighbor. A lossprint specifies the number of data frames received within some window of data frame IDs. If high data losses are observed,[10] the lossprint information is used to locate the lossy link.

If an upstream and a downstream nodes with respect to a particular sender[11] both share similar losses, then the lossy link is very likely located closer to the sender. A notable exception is if there is a node with a large fanout: that node and its downstream nodes may share similar losses, but the problem is "between" them. Therefore, in Yoid, an upstream node collects and compares the lossprints from all of its downstream nodes, and if all of them have high losses, then one or several of the downstream nodes are "kicked-out" to reduce the upstream node fanout (unless there is information that the losses are indeed somewhere upstream). However, if there is only one downstream node with high data losses, *i.e.,* if the high losses are not shared among several downstream nodes, then very

[10] Currently, the threshold for high data losses is 5%.

[11] In case of bi-directional shared trees such as the one used by Yoid, the data from a sender can come from any neighbor.

likely the losses are because that particular downstream node has large fanout. Therefore, the disconnection of that node it temporary postponed, giving it some time to reduce its own fanout and to eliminate the losses. If the losses continue to be persistent, then the node is "kicked-out," because quite likely the particular link to that node is lossy.

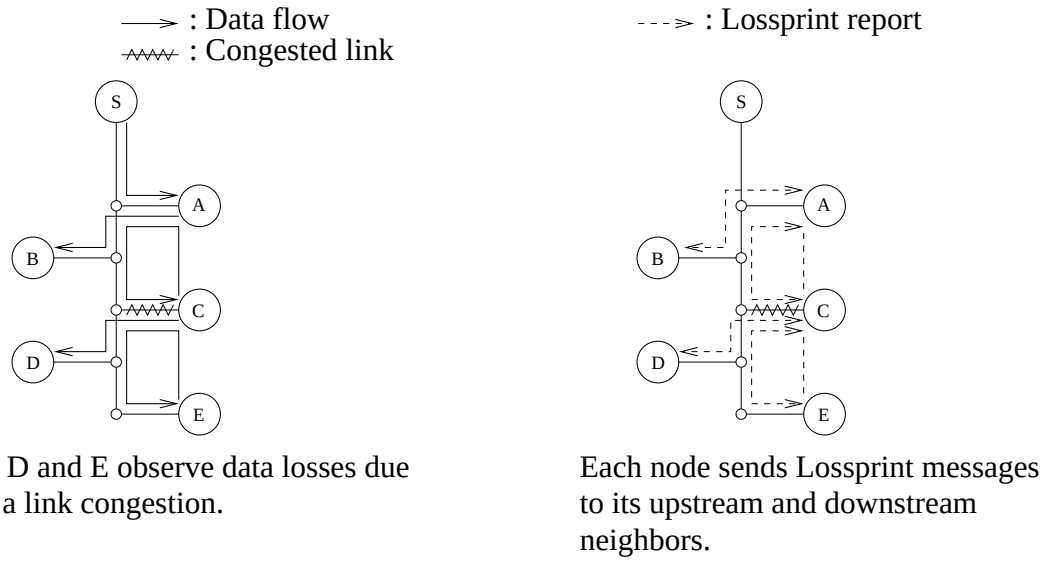


Figure 5.8: Loss-rate refinement algorithm: initial setup and lossprint messages origination.

The algorithm can be illustrated better with an example. Consider the setup in Figure 5.8 where S is the sender and the upstream node for of A, A is the upstream of B and C, and C is the upstream of D and E. If we assume that the edge link of C is a low-bandwidth link, then it may not be able to carry three times the data bandwidth (once from A to C, then from C to D and E respectively).

Due to the link congestion, C, D and E will all observe data losses. Therefore, when each of the nodes sends its lossprint to its upstream node and downstream nodes, A will learn that C observes high data losses, and C will learn that it shares similar losses with D and E. However, A will also learn that its second downstream node B does not share

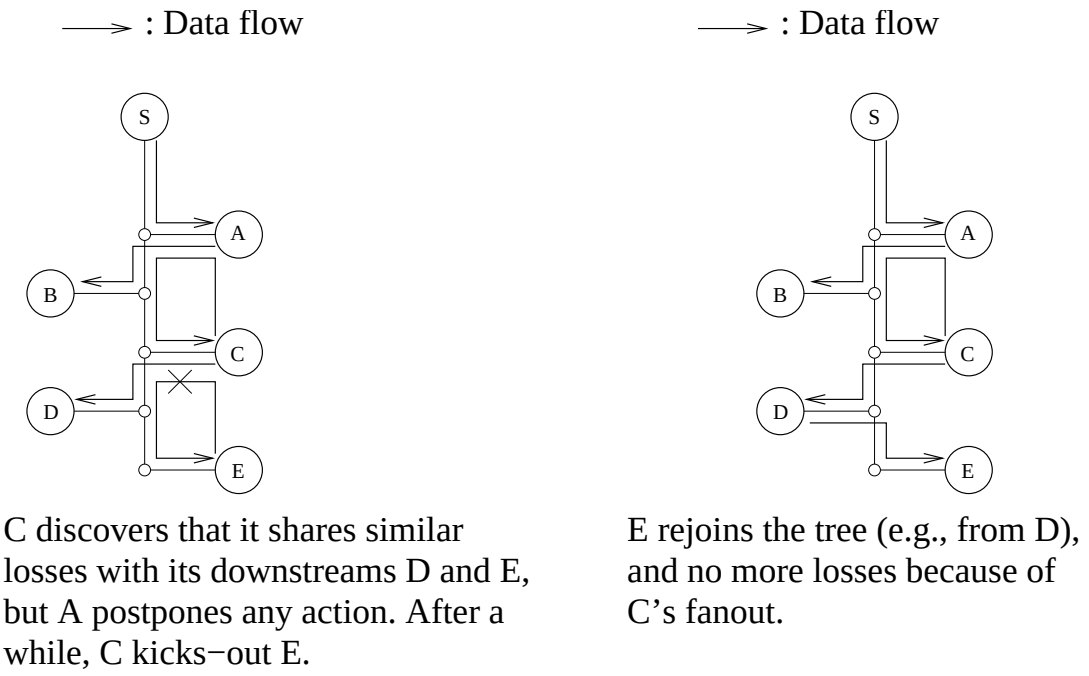


Figure 5.9: Loss-rate refinement algorithm: C, D and E reorganize to reduce the load on the congested bottleneck link.

similar losses with C, therefore A will postpone taking the action of kicking-out C. In the mean time C will kick-out the downstream node with the worst losses (node E in this example), After E is disconnected from the tree, if it was a child of C, it must connect quickly back by selecting a new parent, otherwise C is the one that will have to find a new parent on the Yoid tree. After E chooses D as its new parent, the tree has converged and there are no more losses because of C's large fanout (see Figure 5.9).

5.3.2.3 Discussion

Both tree refinement algorithms are data-triggered; in other words, if there is no active sender, the tree will not change. This has its advantages and disadvantages. On one hand, if the group is idle for long time, then there is no need to reorganize the tree, especially if

we want to minimize the amount of extra control traffic. On the other hand, if the tree has undesired characteristics and if we wait for the senders to start sending data to improve it, there is a transient period of poor performance until the tree adapts.

The tree refinement algorithms (latency and loss-rate) are independent of each other, but because both of them are running at the same time we can consider them as two forces that define the tree shape. The first force, the latency refinement algorithm tries to improve the end-to-end latency by effectively reducing the tree depth. At the same time the second force, the loss-rate refinement algorithm restricts the maximum fanout of the nodes behind low bandwidth links. Thus, Yoid implicitly considers both metrics (latency and throughput) in constructing the application-level overlay.

Finally, we should note that both the latency and the loss-rate refinement algorithms are local, involving coordination only between a node and its neighbors. Therefore, because our goal is avoidance of worst-case performance, there is a reasonable expectation that these local algorithms will converge quickly. On the other hand, strictly speaking, the loop termination algorithm is not local. For example, in the worst case, if all participants create one single loop, then each of the participants would use information about all other participants to decide whether it would be the node to break the loop. However, it is only the information that is "global," an artifact from the particular mechanism that is used. Apart of that, the mechanism does not rely on any explicit global methods, therefore it is reasonable to expect that it would not result in long convergence time. This is confirmed by the performance results described in the next section.

5.4 Performance Results

In this section we describe the results of several experiments we conducted to validate the design of the tree management algorithms, and to understand the impact of various parameter settings. Our first set describes results from a Yoid simulator. Then, we report the results of Yoid experiments over the Internet.

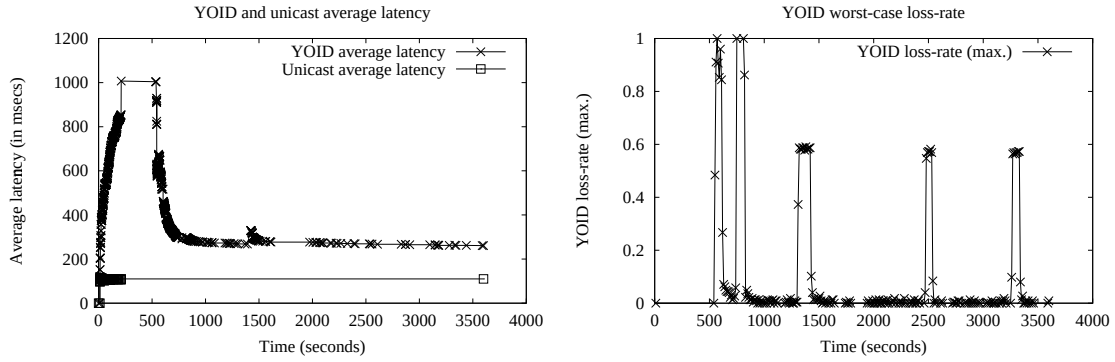
5.4.1 Simulation Results

For our simulations we developed a packet-level simulator. The inputs to the simulator include the underlying network topology and the placement of the Yoid nodes. Each of the selected Yoid nodes is running the Yoid stack. All packets are handled by the topology routing engine. The engine forwards the packets hop-by-hop, and at each hop it considers the latency and the bandwidth of the topology link to propagate or queue the packets. There is also a simple queueing mechanism (FIFO) which tail-drops the packets if the queue is full. In all simulations the queue size is 1000 packets.

For most simulations we use a real-world router-level topology. The topology is same as the Internet core topology described in Section 3.3.2. However, after we select location of Yoid nodes in the topology, we add an extra leaf node to each location, and those extra nodes are the Yoid members. This step helps us simulate bottleneck edge links.

In all simulations we use constant-rate 9kbps traffic of packet size of 50 octets which approximately models the audio traffic generated by RAT [99] with low-end GSM encoding. In most cases we use 200 receivers placed at random.

In our simulations we consider two metrics. The *average end-to-end latency* is computed as the sum of the pre-configured latency on each link from all sender nodes to all receivers,



(a) Average latency

(b) Worst-case data losses

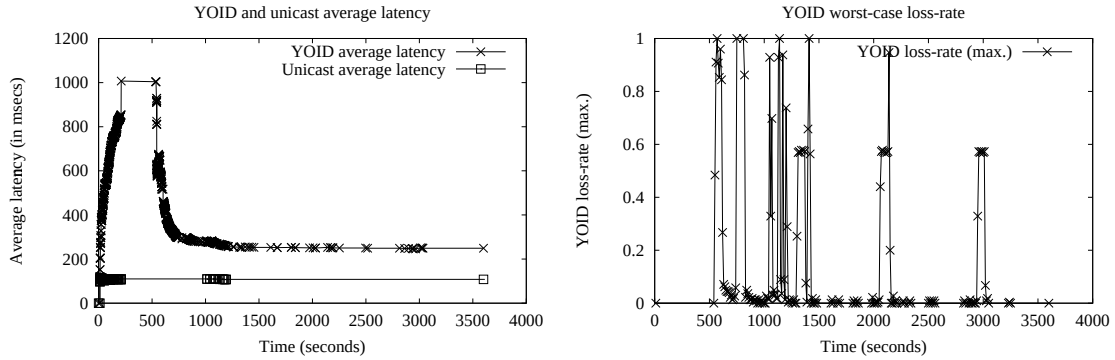
Figure 5.10: Simulation results: sender and 25% of receivers behind 40kbps/80ms links.

and then averaged across all senders and receivers. The *worst-case data loss-rate* represents the largest loss seen at a receiver in a window of 10 seconds.

In most simulations we did not observe any loops. In some, however, we saw 1 or 2 loops formed per simulation run, and these were quickly resolved within the order of few hundred milliseconds. We do not further discuss loop convergence.

In the following paragraphs, we discuss the performance of Yoid's tree refinement algorithms across a wide variety of scenarios.

In the first simulation we test the Yoid performance with typical setup when a number of members are behind bottleneck links. In this scenario, the edge links of 25% of all Yoid nodes have bandwidth capacity of only 40kbps and link latency of 80ms. The rest of the links in the topology have bandwidth capacity of 1Mbps and latency of 10ms. This scenario is intended to model a Yoid group with many receivers behind low bandwidth and high latency bottleneck links such as dial-up phone modems. Initially, all receivers join the tree one-by-one with an interval of 1 second between them. Then, 300 seconds after



(a) Average latency

(b) Worst-case data losses

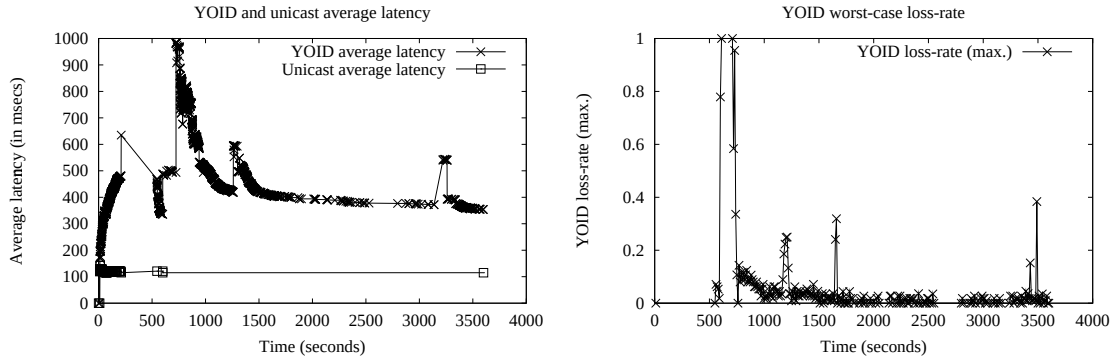
Figure 5.11: Simulation results: 25% of receivers are members for only 1000 seconds.

the last member joins the tree (500 seconds from the beginning of the simulation), that last member starts sending data. To stress the mechanisms further, the simulation is setup such that the sender is behind a bottleneck link as well.

The results for the average latency and worst-case data losses are in Figure 5.10(a) and Figure 5.10(b) respectively. The average latency results show that before the sender starts sending data at time 500, the average sender-to-receivers Yoid latency can be on the order of 10 times larger than the average unicast latency between sender and receivers. This difference is not a surprise, because basically the tree is created at random and latency refinement hasn't had a chance to work. After the sender starts sending data, eventually both the latency and loss-rate refinement algorithms are triggered. Indeed, we can see a drastic drop in the Yoid average latency from 1000ms down to 260ms (compared to the 110ms unicast latency). After that the tree has almost converged, and there are few changes. During the rapid tree improvement the worst-case loss-rate for some nodes is close to 100% (Figure 5.10(b)), primarily due to transient packet losses when a large number of

members almost simultaneously select and switch to new parents. After that, most of the losses are negligible, and occur when a member selects a new parent to improve its data delivery latency. However, over time there were few occasional high losses as well (see the spikes at time 1400, 2500 and 3300 in Figure 5.10(b)). The reason is that typically the sender behind the bottleneck link would have the maximum number of on-tree neighbors that its bottleneck link can support. However, from time to time other members would try to select the sender as a parent, as determined by the latency refinement algorithm. When an extra child joins the sender, the bottleneck edge link of the sender becomes congested, and therefore its children observe high data losses. In that case, the loss-rate refinement algorithm is triggered, and the new child, being the one that has been for very short amount of time connected to the parent, is disconnected and forced to select a new parent. Here we should emphasize that usually the worst-case losses are observed by a fraction of the Yoid nodes; indeed, when we looked at the average losses, they were significantly lower; for example, the average loss-rate during the spikes of the worst-case losses are on the order of 5-10%. For this typical scenario, we conclude that the Yoid tree refinement algorithms work well to reduce the tree latency without affecting loss performance.

The second simulation setup is same as the first one except that after 1000 seconds 25% of the receivers start leaving the group, one-by-one with interval of 4 seconds between them. The results for the average latency and worst-case data losses are in Figure 5.11(a) and Figure 5.11(b) respectively. Soon after the members start leaving at time 1000, there are nodes that observe high losses due to the temporary tree partitioning. However, in most cases there are just few on-tree nodes that are affected by a node that has just left the group. Soon after all 50 nodes have left the tree (approximately 1200 seconds



(a) Average latency

(b) Worst-case data losses

Figure 5.12: Simulation results: two senders, 25% of receivers behind 40kbps/80ms links.

from the beginning of the simulation), the worst-case losses become sporadic. Apart of the higher losses while members are leaving the group, the rest of the results are very similar to the previous simulation without member dynamics. Thus, Yoid's refinement algorithm maintains tree performance (modulo transient losses which we can alleviate with application-level buffering, Section 5.2) across membership changes.

Finally, the third simulation setup is same as the first simulation (25% of receivers are behind 40kbps 80ms links), except that now there are two senders (the first and the last members) instead of one, and each of them is sending with bandwidth rate of 4500bps so the total bandwidth is same as in the first simulation with a single sender. The results for the average latency and worst-case data losses are in Figure 5.12(a) and Figure 5.12(b) respectively. When we have more than one sender, we can see that the average latency of the Yoid tree increases from 260ms to approximately 350ms, which is not unexpected, because now the data distribution tree has to be "balanced" among the senders. Another interesting observation is that with two senders, the spikes in the worst-case losses have been

reduced notably (compare Figure 5.10(b) with Figure 5.12(b)). One possible explanation for this is that fewer members would select one of the senders as their parent, because it is less likely that would improve their overall data propagation latency if we consider the second sender as well.

We actually performed several more simulations, whose results we summarize in the next few paragraphs.

To stress the loss-rate refinement algorithm, we use a setup similar to the first simulation, except that the selected bottleneck links have capacity of only 15kbps, but their latency is reduced to 10ms. Thus, a member behind a bottleneck link cannot have more than one on-tree neighbor without introducing significant losses. In this case, we observed that the average tree latency increased from 260ms to approximately 300ms, primarily due to the fewer choices there are to interconnect the members. However, the time duration length of the spikes with the data losses increased approximately twice and reached 100% losses (even for the average losses). The reason is that now the sender bottleneck link is much more fragile, and after it is congested, not only it will take longer to return to its non-congested state, but disturbing the single outgoing data flow from the sender will affect all members equally.

To stress the latency refinement algorithm, we increase the latency of the same edge links as in the previous simulation to 80ms, but at the same time we increase the link bandwidth to the default of 1Mbps. In that case the average latency was lower, on the order of 210ms, but the worst-case losses were much smaller (*i.e.,* on the order of 10% right after the sender was activated, and no spikes were observed after that). The reason for

that is because we have much more flexibility about the number of children a parent node can have.

To test the sensitivity of the latency refinement algorithm to the latency improvement threshold, we use setup same as in the first simulation, except that the latency improvement threshold is 5ms instead of 50ms. Because 5ms is smaller than the latency of any single link, the expectation is that the average latency will improve and will become closer to the unicast average latency. However, it turned out that the results are very similar to the previous results with 50ms threshold. This suggests that the bandwidth limitation of the bottleneck links is the dominant factor in defining the shape of the data propagation tree.

On a smaller group of 28 members, the average Yoid tree latency was on the order of 180ms (versus 100ms for unicast), and there were almost no losses. Even during the rapid tree reconfiguration right after the sender was activated, the worst-case losses were below 5%. This result suggests that on smaller groups such as desktop conference meetings Yoid would perform much better than the results we have demonstrated with 200 members.

We tried other member placement as well–extreme affinity, extreme disaffinity, extreme clustering (see Section 3.2.3), and in all cases the results were similar to the results with random member placement. We tried also some other topologies–Mbone, AS, random graph, generated power-law graph,[12] with various member placement. For all topologies, the results were qualitatively similar to the Internet-core results. This result suggests that Yoid tree management algorithms are robust for a variety of topology and member placement.

[12]See Section 3.3.2 for some of their metrics.

5.4.2 Experimental Results

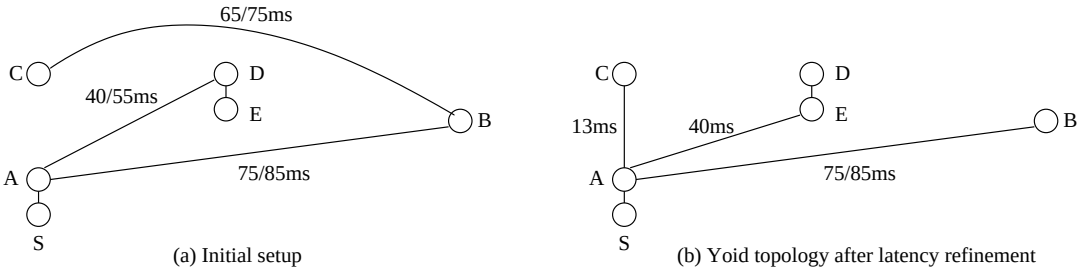


Figure 5.13: Latency refinement experiment topology.

To verify how the tree refinement algorithms work in practice, we created two real-world experiments over the Internet with our Yoid implementation. Our experiments validate the overall design, and show how our algorithms handle real world artifacts, such as abnormally configured unicast routing and asymmetric access links.

For the first experiment we explicitly setup a number of hosts in the Yoid tree topology shown in Figure 5.13(a), where S is the sender and the root of the tree, and all other hosts are receivers. Node S and A are on the same LAN on the South-West coast of the US, C is on the North-West coast, D and E are on the same LAN on the East coast of the US, and B is in the UK. Some of the ping-measured end-to-end latencies (computed as half of the round-trip time) are shown in the figure, including the latencies in both directions if they differ notably. Approximately one minute and a half after S began sending data at approximately 4kbps constant-rate (5 packets/s, 64 octets data payload), all of the nodes collected several samples from their active candidate-parents. We expected that only C would select A or S as its new parent, and no other tree changes would occur. Surprisingly, first E selected A as its new parent despite the fact that the average

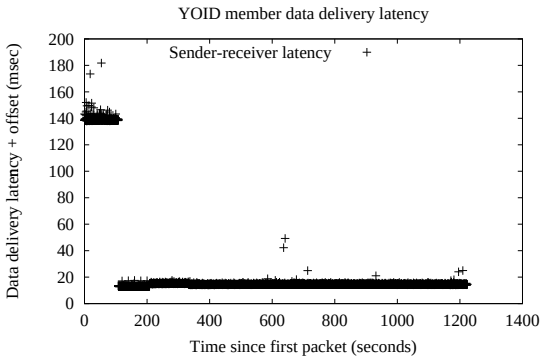
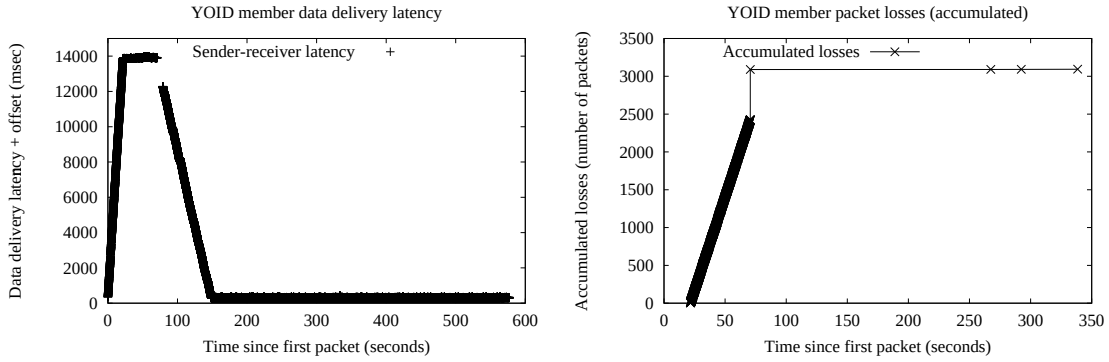


Figure 5.14: Latency refinement experiment result.

improvement seemed to be below the 50ms threshold. A closer investigation suggested that the reason is probably the unusual underlying unicast routing (hosts B, D and E are part of the CAIRN experimental network [7]). First, the unicast paths from A to D and E are 9 hops, while the paths in the opposite directions are 15 and 16 hop respectively! Second, even though D and E are on the same LAN, their paths to A are different, and as a result there was notable difference in the latency: from D to A the round-trip time was 110ms, while from E to A it was 80ms. Therefore, asymmetric routing, as well as the different paths that are used may have resulted in larger difference in the one-way propagation latency from A to D and E than the difference we have estimated based on the round-trip time. Soon after E selected A as its parent, C selected A as its parent as well. Interestingly, soon after that D selected E as its new parent. After that no changes were observed, and the tree converged to the one shown in Figure 5.13 (b). Figure 5.14 shows how the data propagation latency[13] from S to C changes over time (note the 130ms

[13]The propagation latency is computed based on the absolute time difference measured by the sending and receiving hosts, but despite that the hosts are running NTP [83], only the relative time difference change should be considered accurate.



(a) Data delivery latency

(b) Accumulated packet losses

Figure 5.15: Loss-rate refinement experiment results.

latency improvement approximately 80 seconds after the sender was activated). During the experiment, only one packet was lost by a member (D), probably because of switching parents, but if the sending rate were higher we would expect greater packet loss.

For the loss-rate adaptation experiment we set up a Yoid tree similar to the one in the example in Figure 5.8, where C is a host behind an ADSL link with bandwidth of 384+/128kbps downstream and upstream respectively. Initially, all nodes joined the group, except E. After the sender began sending data at the rate of approximately 75kbps (80 packets/s), no notable losses were observed, and the propagation latency was on the order of tens of milliseconds. However, after E joined the tree (it was explicitly configured to select C as its parent), the ADSL link became congested, because the outgoing traffic was higher than the upstream bandwidth capacity of the link. Figure 5.15(a) shows how the data propagation latency observed by E quickly increased to approximately 14 seconds.[14] Soon after that the link began to drop packets, and both D and E observed high losses.

[14]We wanted to test the loss-rate refinement algorithm only, therefore we explicitly disabled the latency refinement algorithm; otherwise, E may have switched to a new parent with the help of the latency refinement algorithm instead of the loss-rate refinement algorithm.

Figure 5.15(b) shows the accumulated packet losses for E. Node C itself did not observe any losses because the downstream and upstream bandwidth capacity are independent, therefore soon after C received the lossprint control messages from D and E that indicated high losses, C "kick-out" E, because both D and E had similarly high losses, but E had been a child of C for much shorter time. After E was forced to find a new parent, it selected node D,[15] and no significant losses were observed after that. The total amount of time between when E joined the group and when it switched to a new parent to eliminate the losses was about 90 seconds. In some earlier experiments with a different ADSL line that had same bandwidth capacity, we observed that the line became congested much faster, and packets were dropped very soon. In this experiment, it seems that the router on the other side of the DSL link was queueing much more packets, therefore for more than 20 seconds D and E observed long propagation latency due to the large queue, but no losses. If the router was dropping the excess packets instead of queueing them for very long time, the amount of time for E to switch to a new parent would have been shorter.

5.5 Conclusions

In this chapter we studied the problem of tree construction and management for application-level multicast. First, we studied the inefficiency of application-level multicast in general for a variety of topologies, and we found that in most cases the inefficiency can be within a factor of 2.5–3.0 compared to network-layer multicast.

[15] In fact, it was explicitly configured to select node D, so we could stress further the mechanism; otherwise, it may have selected some of the nodes closer to the sender.

Encouraged by those results, we attempted to design practical mechanisms that can be used to achieve similar performance in reality. We designed, within the framework of Yoid application-level multicast system, a set of algorithms that can be used for rapid management of the application-level multicast data delivery tree, and for refining the tree characteristics in term of end-to-end propagation latency and data losses. The simulation results demonstrate that the end-to-end propagation latency indeed can within a factor of 2 of the unicast latency, and at the same time the losses due to bottleneck links can be minimized. The mechanisms we use are adaptive, and in our simulations we demonstrate that the performance is not impacted by the underlying topology or the placement of the participants. We also used real-world experiments to verify how those algorithms can work in practice. The tree management mechanisms are implemented and are used in Yoid application-level multicast system.

By this study, we demonstrate that if we use the appropriate end-to-end mechanisms, we can achieve reasonable performance, and if the mechanisms are adaptive, they can reduce the impact of the underlying topology or participant placement on the protocol performance.

Chapter 6

Related Work

6.1 Multicast Forwarding State Aggregation Related Work

Multicast forwarding state scalability is analyzed in greater details in [131] by taking into account receiver placement and membership interest. Various topologies (Tiers, Transit-Stub, Mbone and AS) are considered as well. One of their findings is that there is difference in state among the topologies, and one of the reasons for this is the difference in node-fanout distribution among topologies. Further, increasing network connectivity increases the amount of state in few core routers or domains, but reduces it everywhere else. This is the only work we are aware of that has studied the topology impact on multicast forwarding state aggregatability. Our study is complementary to theirs.

To our knowledge, leaky multicast forwarding entry aggregation has not been studied in the literature. However, several other forwarding table compaction approaches have either been implemented or considered in the literature. We discuss these now.

One alternative to aggregation is to only cache forwarding entries of currently *active* multicast groups. By active, we mean those groups for which a router received data in

the recent past. The cache entries are populated from a *routing table* held in slower, less expensive memory. However, this approach only works when there is sufficient traffic locality to maintain high cache hit ratio. As the number of concurrently active groups increases, the degree of traffic interleaving will grow, requiring a large cache to achieve good hit ratios. Conversely, because each cache miss results in slow path router forwarding, router performance can degrade with an increase in the number of groups.

A number of papers ([28] [126] [85] [67] [109]) describe software based solutions that use carefully chosen data structures to reduce the size of the unicast forwarding table such that it fits in on-chip caches. Accesses to such cache are fast, allowing very high-speed unicast packet forwarding even without expensive lookup hardware. These solutions are attractive because of their flexibility, but unfortunately they are limited by the capacity of the CPU cache. The number of unicast forwarding entries that can be stored in the CPU cache is on the order of 100,000. This is an order of magnitude or more smaller than the possible number of multicast forwarding entries.

A hybrid hardware/software approach is described in [40]. Using some simple additional hardware and a reasonable amount of cheap memory (33 MB of DRAM for IPv4), the lookup can be completed by using only 1-2 memory accesses (50-100 ns, i.e. 10-20 Gbps lookup bandwidth) to slower main memory. This type of solution can be used to store a much larger number of entries (several million entries), but may not be flexible enough to accommodate the expected size of multicast forwarding tables.

Study [118] is the first work we are aware of that addresses directly the problem of reducing the number of multicast forwarding entries. The basic idea is to dynamically establish tunnels that bypass the routers with fanout of only one outgoing interface for

a given group. To accomplish this, however, the multicast routing protocol needs to be modified, and there is the additional encapsulation/decapsulation overhead per data packet. Also, this solution is not beneficial if the fanout of the forwarding state is more than one. Nevertheless, this solution can be applied together with the leaky aggregation scheme we have studied.

The basic idea in REUNITE [111] is similar to [118]: bypass the routers with fanout on the multicast tree of only one outgoing interface by using unicast to the next downstream router that has more than one outgoing interfaces. The mechanism to unicast the packets is different and does not use dynamic tunneling.

Distributed Core Multicast (DCM) [6] proposes a mechanism to setup a number of core routers in each area at the edge of the backbone. The DCM routers act as backbone access point for sender inside their area, and outside receivers. A special protocol for receiver membership and data distribution is run among the DCM routers. This protocol eliminates the need for any non-DCM backbone router to perform multicast routing.

Study [116] proposes and analyzes a mechanism for per-interface state aggregation. The assumption is that the router multicast lookup is designed as "filters" on each interface. This design has the advantage that it can be fast and easy to implement on high-end routers that can have a separate CPU and lookup table on each interface, but its drawback is that it imposes a specific architectural choice. The state aggregatability is analyzed with different multicast address allocation schemes and receiver placement, but the impact of topology is not considered. Their results demonstrate that it is possible to achieve some aggregation, even though not dramatic, and qualitatively they match our entropy-based results.

6.2 Replica Placement Related Work

A number of papers have studied the impact of Web server replica or cache placement on performance. Note that some of the recently published papers are independent studies, but have notable similarity in problem formulation and final results. The replica or cache placement problem can be modeled after the *center placement problem*, a well-known problem in graph theory, and in particular two of its variations: *the facility location problem*, or *the minimum K-median problem* [22]. A number of approximate solutions have been proposed in the past [125], but they are either very computationally expensive, or are difficult to apply in practice.

Krishnan et al. [65] study the problem of placing transparent en-route network caches (TERCs), and in particular how various placement methods can be used to reduce the network traffic or the average access latency. Unlike our work where we assume the replicas can be placed anywhere in the network, their work restricts the caches to be only on the path between a client and the server. The topology-based evaluation is performed by using traceroute-collected Internet trees; other topologies are not considered.

Qiu et al. [95] consider the problem of placement strategies for Web server replicas within the context of CDNs that offer Web server hosting services. They propose several placement algorithms, including a simple greedy placement which we use in our work, and which is very similar to the greedy algorithm in [65]. They find that this greedy algorithm performs very well in practice (typically within a factor of 1.1–1.5 of the optimal solution). Further, its performance is relatively insensitive to imperfect input data such as client

locations and network topology information. However, this study does not consider node-fanout based placement. Similar to our study, the evaluation metric is *relative performance*. In their study they use random trees, random graph and AS topologies, and the results for all topologies are qualitatively similar, which supports our thesis that for relative metrics the topology impact is relatively small.

The study by Jamin et al. [60] is similar to [95]. Their work examines the impact of the number of replicas on the performance of various replica placement methods. Their main finding is that, regardless of the placement method, increasing the number of replicas is effective in reducing client download time only for a very small number of replicas. In their study they use power-law generated topologies, as well as Internet traces. They also discuss an AS-level fanout-based placement, in which replicas are placed within ASs in decreasing order of node degree on the AS topology. The results suggest that the AS-level fanout-based placement can perform almost as well as the greedy placement. Our study on replica placement is centered around this finding, and we try to verify it through more detailed simulations by using router-level Internet topology, instead of only AS-level topology, and by exploring in more details the impact of various replica and client placement methods.

6.3 Reliable Multicast Related Work

Previous work on reliable multicast that compares router assisted with non-assisted schemes is [88]. However, the comparison there is limited in scope compared to our work. For example, network overhead is not considered, and the used topologies are much smaller (approximately 200 nodes) generated topologies, while in our study we use a variety of topologies–canonical, generated and real-world, including a large (over 50K nodes) real

world router-level network topology. Moreover, in our work we have added analysis to complement our results.

We now briefly describe some of the latest proposals for reliable multicast and point out relations to our ALH and RAH schemes. While our list is not exhaustive, it covers a broad range of current proposals. We begin with the non-assisted schemes first.

SRM [36] employs two global mechanisms to limit the number of messages generated, namely duplicate suppression and back-off timers. In SRM, recovery messages (requests and replies) are multicast to the entire group; receivers listen for recovery messages from other receivers before sending their own, and suppress duplicates. Thus, SRM creates a virtual hierarchy on the fly every time there is loss in the group. However, lack of scoping means that requests and retransmissions generated by SRM will reach the entire group. Local recovery methods have been proposed for SRM [73], which bring SRM closer to our ALH scheme.

RMTP [72] is a typical example of a static hierarchical scheme which closely resembles our generic ALH scheme. The group is manually configured into Designated Receivers (DRs) and their children. DRs and their children form local groups. The source multicasts data to all receivers on the global group, but only the DRs return acknowledgments. Children unicast acknowledgments to their DRs, which schedule retransmissions using either unicast or local multicast depending on how many requests a DR has received. The Log-Based Receiver-reliable Multicast (LBRRM) [49] is another example of a static hierarchical scheme. LBRRM uses a primary logging server and a static hierarchy of secondary logging servers which log all transmitted data. Data is multicast from the source to all logging

servers and all receivers, but only the primary logging server sends acknowledgments. Receivers request lost data from the secondary loggers, and in turn the secondary loggers request lost data from the primary logger.

The Tree-based Multicast Transport Protocol (TMTP) [132] is another example of an ALH scheme, but it uses a dynamic hierarchy. In TMTP, new members discover parents using an expanding ring search. Each endpoint maintains the hop distance to its parent, and each parent maintains the hop distance to its farthest child. These values are used to set the TTL field on requests and replies to limit their scope. LGMP [48] is another hierarchical, subgroup-based protocol, where receivers dynamically organize themselves into subgroups by selecting a Group Controller to coordinate local retransmissions and process feedback messages. TRAM [16] is another dynamic tree-based protocol designed to support bulk data transfer. The tree formation and maintenance algorithms borrow from other schemes like TMTP, but TRAM has a richer tree management framework. TRAM supports member repair and monitoring, pruning of unsuitable members, and aggregation and propagation of protocol related information.

Moving to router-assisted schemes, Addressable Internet Multicast (AIM) [68] is a scheme that uses forwarding services that require routers to assign per-multicast group labels to all routers participating in that group. AIM uses these labels to send a request towards the source which get redirected to the nearest upstream member. If data is available, the NACK receiver responds with a retransmission which is also forwarded according to the router labels. AIM is very similar to our RAH scheme. Active Error Recovery (AER) [63] is another scheme that is very similar to our RAH scheme. In AER, each

router that has a repair server attached periodically announces its existence to the downstream routers and receivers, and serves as a retransmitter of the lost data on the subtree below it, or collects and send NACKs upstream. OTERS [71], uses a modified version of the mtrace [11] utility to build the hierarchy by incrementally identifying sub-roots using back-tracing. For each subroot, OTERS selects a parent. Unlike our RAH scheme, OTERS assumes the responsibility of discovering the topology and keeping track of changes in the structure of the underlying multicast group. Similar to OTERS, Tracer [69] also uses mtrace to allow each receiver to discover its path to the source. Once the path is discovered, receivers advertise their paths to near-by receivers using expanding ring search. Once receivers discover nearby receivers, they use the data from the traces and their loss rate to select parents.

Finally, PGM [108], unlike the schemes described earlier, peeks into transport headers to filter messages. NACKs create state at the routers which is used to suppress duplicate NACKs and guide retransmissions to receivers that requested them. PGM creates a hierarchy rooted at the source, but provision is made for suitable receivers to act as Designated Local Retransmiters (DLRs) if desired.

6.4 Application Level Multicast Related Work

Recently, there has been significant research activity in application-level multicast architectures. Among these, Narada [51, 17] is architecturally the closest to Yoid, but its design is very different. Yoid creates a single shared tree per session, while Narada creates multiple, source-specific trees. The tree construction is different as well. In Yoid the "tree-first"

approach creates first the tree, and then the tree is gradually refined. In Narada the "mesh-first" approach creates first a virtual mesh among all participants, and then a distance-vector protocol with a dual latency-bandwidth metric creates the tree. The mechanisms for measuring the latency and bandwidth are also very different. Narada uses active probing to determine available network bandwidth, while Yoid passively measures data losses to locate bottleneck links; on the other hand, Narada measures the absolute latency of each link, while Yoid considers the latency difference between alternative solutions. The performance evaluation of Narada is done by using three different topology types: Waxman, AS, and backbone connectivity. The results for all topologies are similar, which supports our thesis that end-to-end mechanism can be adaptive to topology characteristics, among other factors. Other architectures differ significantly from Yoid. Overcast [61] is another end-system multicast architecture, but designed specifically for single-source reliable multicast services. The performance of the tree is carefully monitored, and the tree is reorganized when necessary. ALMI [90] (Application Level Multicast Infrastructure) is an example of an architecture that uses centralized mechanisms to manage the distribution tree. In that work the authors evaluate their scheme over Random graph and Transit-Stub generated topologies [8]. For both topologies the results are similar. This result, similar to our work, demonstrates that with the appropriate end-to-end mechanisms it is possible to achieve results that are well adaptive to various topologies. Banana Tree Protocol [47] is an end-host multicast protocol for distributed file sharing. Bayeux [136] uses hierarchical addressing and routing to achieve scalability and fault-tolerance. Unicast-based approach for building multicast services is used in [19] and [129] as well.

Tangentially related to Yoid are overlay networks. Scattercast [14] is an architecture that can be used to create an overlay of servers for broadcast services. The overlay trees are created using a protocol similar to Narada, but designed for better scalability. Resilient Overlay Networks [3] is a service for application-level routing that can be used to discover "short-cut" paths with better characteristics than the paths provided by the underlying unicast routing. The X-Bone [119] is designed for rapid, automated deployment and management of overlay networks. An architecture for self-organizing overlays is described in [58]. Within the context of ad-hoc networks, CEDAR [107] is a distributed algorithm to establish and maintain a self-organizing routing infrastructure.

Finally, peer-to-peer architectures for sharing files, music and other information have gained large popularity (*e.g.,* Napster and Gnutella). These, together with peer-to-peer distributed hash tables [110, 100] are architectural cousins of Yoid.

6.5 Network Topology and Protocol Performance Related Work

One of the early works that consider multicast protocol performance by comparing shared with source-specific trees is [128]. The topologies used in the evaluation are the early ARPAnet topology, as well as some generated topologies using the Waxman model [127]. The evaluation shows that the results across all topologies are similar. The impact of topologies on multicast routing protocol performance is studied in [134]. Their results show that different topology generation methods and parameters may create topologies

with different metrics. However, the difference between topologies in term of multicast protocol performance metrics is not significant, or virtually not existing.

A more recent work [92] examines the interesting suggestion made earlier [18] that $L(m) \propto m^{0.8}$ where m is the number of receivers, and $L(m)$ is the multicast tree size in number of links. Through analyses and simulations over a range of topologies (real and generated), they have found that this correlation exists for several topologies and for a variety of receiver placement. Only when the topology reachability does not have exponential property, this correlation is not so strong. This result is a good demonstration of when a particular protocol performance metric does not depend on the particular topology, but eventually is affected by a specific topology metric.

In recent years there has been notable interest in modeling network topologies. GT-ITM [8] and Tiers [31] are two popular topology generators. However, more recently they have been compared with real-world topologies and it has been suggested that the topologies generated by GT-ITM and Tiers have different properties from the real-world topologies [98].

The first work to suggest that real-world topologies have power-law characteristics is [34]. One possible model that explains this observation is described in [5]. The factors that contribute to the power-law characteristics of the topologies based on that model are examined in [81]. Topology generators that create topologies with power-law characteristics are described in [81, 1, 87, 62]. Power law topologies are examined more closely in [112], and is suggested that degree-based generators produce better models of both AS-level and router-level Internet graphs. Very recently, the power-law characteristics of real-world Internet topologies (AS and router-level) have been re-examined and it has been suggested

that the topologies have power-law-like characteristics, but in fact are not exact power-law graphs [15].

In [98] network topology properties are investigated, including how topologies properties may affect the protocol performance. Our work originated from this one. Virtual topology constriction is considered in [133], a study that also originated from [98]. A number of virtual topology construction algorithms are studied for different topologies: random graph, tree, mesh, reduced mesh, Mbone and AS. The results show that for some metrics there is a notable performance difference for different topologies, but for other metrics the difference is small.

Chapter 7

Conclusions and Future Work

In the beginning of this thesis we asked the question if there is anything else beyond raw wire speed and router throughput that has impact on network performance. To find an answer, we considered the impact of network topology on protocol performance. In particular, (a) what is the impact of the underlying topology on protocol performance; (b) can we use partial knowledge about the underlying topology to improve performance, and (c) can we use end-to-end mechanisms to design protocols that are adaptive to the underlying network topology?

To answer those questions, we performed four case-studies. Below we summarize our findings for each of the studies.

7.1 Summary of Case Studies

Summary of multicast forwarding state aggregation case-study

We use a simple information-theoretical approach, the *interface entropy* to estimate the lower bounds on the aggregatability. We look into different topologies and different receiver size set, and found that the aggregatability in fact depends significantly on the

topology: for some topologies we can aggregate the state with compression ratio on the order of 10 times or more if less than 5% of nodes are receivers. On the other hand, for other topologies the aggregation ratio is much lower and could be close to 1.0 (*i.e.,* no aggregation is possible). Unfortunately, those aggregation ratios may not be sufficient or may be difficult to implement in practice to solve the problem. Therefore, we look into alternative solutions that can be used to achieve better aggregation, but at the expense of losing some information. One such solution that trades router memory for network bandwidth is described and evaluated.

Summary of Content Distribution Network replica placement case-study

The replica or placement problem can be modeled after the *facility location problem*, a well-known NP-complete problem in graph theory. Various approximation algorithms exist, but they are not practical because they assume detailed knowledge about the underlying network topology. Surprisingly, we found that a simple heuristic such as node fanout can be used to achieve results that are within a factor of 1.1–1.2 of existing approximation algorithms that usually perform within a factor of 1.1–1.5 of the optimal solution. Further, in most cases the fanout-based placement performance does not depend on client placement. However, those results are not universal: they apply for power-law and random graphs, but do not apply for topologies such as tree, mesh, and overlay networks such as Mbone [75].

Summary of hierarchical reliable multicast schemes case-study

We look into two classes of hierarchical schemes for reliable multicast: application-level and router-assisted. Our comparison shows that, surprisingly, the application-level schemes could have performance that is qualitatively comparable to router-assisted schemes, as long

as there is a good algorithm to create the application-level hierarchy. One such algorithm is based on the topology distance among all the participants. Further, the results are consistent for a variety of topologies and client placement.

Summary of application-level multicast case-study

First, we study the impact of various topologies on application-level multicast in general and we found that the inefficiency of application-level multicast is acceptable and, in case of overall network overhead for example, in most cases it is within a factor of 2.5–3.0 compared to network-layer multicast. Encouraged by those findings, we ask whether it is possible to have an application-level multicast system that can indeed achieve such level of performance. We design and implement a set of algorithms for endsystem tree management within the framework of an existing application-level multicast system named Yoid. Those algorithms use only end-to-end mechanisms to manage the overlay data distribution tree. Through simulations and real-world experiments we demonstrate that indeed it is possible to achieve reasonable level of performance.

7.2 Conclusions

Now it is time to return back to the three questions we asked in the beginning of this thesis and try to answer them with the help of the findings from each case-study.

1. *What is the impact of the underlying topology on protocol performance? In other words, if the Internet topology were substantially different, should we expect similar performance?*

In all of our case studies we considered various topologies for the protocol performance evaluation. In some cases we found that the topology has some impact on performance (multicast state aggregation and replica placement), but in others it had practically no impact (reliable multicast and application-level multicast). By considering the particular metrics in each of those studies, it seems that if we are considering absolute metrics, then the topology has impact; if the metrics are relative, then the impact is much smaller. On the other hand, the topology impact we observed was considerably smaller than we were expected. From those findings we cannot conclude that the topology impact on other metrics will be similar; rather, in many cases the topology factor could be ignored. A notable exception of a topology that constantly produces results that are significantly different from the other topologies is mesh. This observation can be used for stress-testing new protocols and algorithms. On the other hand, in some cases the random graphs had performance that was notably better compared to real-world topologies; therefore protocol evaluation based on random graphs may produce unrealistic results that are better than they would be in reality.

2. *Can we use information about the underlying topology to improve protocol performance, and what gain can we expect?*

   In two of our studies (replica placement and reliable multicast) we considered using partial knowledge about the underlying topology. Surprisingly, small amount of carefully selected knowledge was sufficient to improve significantly the performance. In case of replica placement problem, selecting replica location based on node fanout

was sufficient to achieve very good performance, within a factor of 1.1–1.2 of much more complicated solutions. Similarly, if we carefully create an application-level multicast hierarchical scheme, its performance can be comparable to the performance of schemes that require router support. It turned-out that in our particular case, information about the end-to-end distance among all pairs of participants is sufficient to create a reasonably good hierarchy. The importance of this finding is that we can achieve reasonably good results even without the explicit support from the network, as long as we have the right mechanism. Indeed, this is what we confirm with our last case-study on application-level multicast, and also the answer of our last question.

3. *Can we use end-to-end mechanisms to design protocols that are adaptive to the underlying network topology?*

   In our case-study on reliable multicast we demonstrated that in many cases the performance of application-level hierarchy can be comparable to the performance of router-assisted hierarchy that uses support from the network to improve performance. However, we did not design any algorithms that can be used to create such efficient application-level hierarchy. In our last case study on application-level multicast we wanted to demonstrate that it is possible to design practical mechanisms that are adaptive to the underlying network topology, and can be used to achieve performance comparable to topology-informed mechanisms. We considered the problem of application-level multicast distribution tree tree creation and management, and we designed a set of algorithms to ensure tree integrity and at the same time to gradually refine the tree performance. We demonstrated through simulations and real-world

Internet experiment that it is possible to achieve results that are typically within a factor of 2.5–3.0 of the optimal solution. The algorithms are adaptive and the result are consistent for various topologies and placement of participants. The particular set of algorithms was implemented and now is used by Yoid, an application-level multicast system.

7.3 Future Work

In this work we have studied four problems. Each of those problems itself can be a candidate for future research.

To improve the performance of the leaky multicast state aggregation, we may need to find stateless mechanisms for identifying high bandwidth data flows. Also, we may need to consider the macro-impact of leaky aggregation on the network, such as whether the leaks are primarily local or global.

The node-fanout based replica placement algorithm we studied is very simple, and reasonably efficient in most cases. However, it is not difficult to imagine some scenarios when such placement is not sufficient. In that case, we may need to apply a hybrid solution: e.g, apply first a node-fanout based placement, and then refine the solution with the help of other, more sophisticated algorithms.

In our case-study on reliable multicast we ignored the cost of collecting the information about the underlying topology, and the cost for maintaining the application-level data recovery hierarchy. Future research is needed to find a practical solution to the problem.

The fourth case study, application-level multicast, is a whole new area that can be studied further. For example, in our work we did not consider fault-tolerant issues and we

did not study scalability limitations. Another problem to consider could be to add support for alternative mechanisms for on-tree data delivery such as anycast or subcast.

Our study of topology impact on protocol performance is far from being completed. In fact, it is one of the initial steps toward understanding the mechanisms that may have impact on protocol performance. Our goal was to have some initial understanding of the problem, hence we used the case-study approach. Thus, we were able to concentrate on various specific details of the issues that may be essential.

The natural next step would be to generalize the problem, and to use more systematic approach in our study. For example, first we may want to classify the factors that may have directed impact on protocol performance (*e.g.,* inter-participant distance, or any-to-any path-concentration).[1] On the other hand, early in that stage we may want to ignore the factors that may not have notable impact, such as the routing (*e.g.,* non-hierarchical vs hierarchical shortest-path). Then, we can try to relate, for example, the topology characteristics and participant placement method to the above factors. Such an approach would eventually give us more conclusive answers. However, without having the case-study results first, taking this approach instead may be much more challenging, and may be more error-prone. Further, there is no guarantee that the problem can indeed be generalized, so it is possible that we may end with analyzing a number of classes of applications. In that case, our case-study results could help to to classify the variety of problems we may have to analyze.

[1] Topology metrics and protocol performance have been considered in an earlier work [98]. We believe that the answer is probably more complicated, especially because currently it is still not completely clear what the exact characteristics of real-world topologies are.

Reference List

[1] Aiello, W., Chung, F., and Lu, L. A Random Graph Model for Massive Graphs. In *Proc. of the 32nd Annual Symposium on Theory of Computing* (2000).

[2] Akamai. http://www.akamai.com/.

[3] Andersen, D. G., Balakrishnan, H., Kaashoek, M. F., and Morris, R. The Case for Resilient Overlay Networks. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (Elmau/Oberbayern, Germany, May 2001).

[4] Ballardie, T., Francis, P., and Crowcroft, J. Core Based Trees. In *Proceedings of the ACM SIGCOMM'93* (San Francisco, USA, September 1993).

[5] Barabasi, A.-L., and Albert, R. Emergence of scaling in random networks. *Science*, 286 (1999), 509–512.

[6] Blazevic, L., and Boudec, J.-Y. L. Distributed Core Multicast (DCM): a routing protocol for IP with application to host mobility. In *Proceedings of Networked Group Communication Workshop* (Pisa, Italy, November 1999).

[7] CAIRN: Collaborative Advanced Interagency Research Network. http://www.cairn.net/.

[8] Calvert, K. L., Doar, M. B., and Zegura, E. W. Modeling Internet Topology. *IEEE Communications Magazine* (June 1997).

[9] Carzaniga, A., Rosenblum, D., and Wolf, A. Design of a Scalable Event Notification Service: Interface and Architecture. Tech. Rep. CU-CS-863-98, Department of Computer Science, University of Colorado at Boulder, September 1998.

[10] Casner, S. First IETF Internet audiocast. *Computer Communication Review 22*, 3 (July 1992), 92–97.

[11] Casner, S., and Thyagarajan, A. *mtrace(8): Tool to print multicast path from a source to a receiver.* UNIX manual page.

[12] Center, U. D. S. The NLANR Project. http://moat.nlanr.net/Routing/rawdata/.

[13] Chandy, K. M., Rifkin, A., and Schooler, E. Using Announce-Listen with Global Events to Develop Distributed Control Systems. In *Proceedings of the ACM Workshop on High Performance Java Network Computing* (February 1998).

[14] Chawathe, Y. *Scattercast: An Architecture for Internet Broadcast Distribution as an Infrastructure Service.* PhD thesis, University of California at Berkeley, December 2000.

[15] Chen, Q., Chang, H., Govindan, R., Jamin, S., Shenker, S. J., and Willinger, W. The Origin of Power Laws in Internet Topologies Revisited. http://topology.eecs.umich.edu/archive/origin.ps.

[16] Chiu, D., Hurst, S., Kadansky, M., and Wesley, J. TRAM: A Tree-based Reliable Multicast Protocol. Tech. Rep. Sun Technical Report SML TR-98-66, Sun Microsystems, July 1998.

[17] Chu, Y.-H., Rao, S. G., Seshan, S., and Zhang, H. Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture. In *Proceedings of the ACM SIGCOMM 2001* (San Diego, CA, USA, August 2001).

[18] Chuang, J., and Sirbu, M. Pricing Multicast Communications: A Cost-Based Approach. In *Proceedings of the INET'98* (1998).

[19] Cohen, R., and Kaempfer, G. A Unicast-based Approach for Streaming Multicast. In *Proceedings of the IEEE Infocom 2001* (Anchorage, Alaska, USA, April 2001).

[20] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms.* MIT Press, 1990.

[21] Cover, T. M., and Thomas, J. A. *Elements of Information Theory.* John Wiley & Sons, Inc., 1991.

[22] Crescenzi, P., and (Editors), V. K. A compendium of NP optimization problems. http://www.nada.kth.se/~viggo/problemlist/.

[23] Danzig, P., Jamin, S., Caceres, R., Mitzel, D., and Estrin, D. An Empirical Workload Model for Driving Wide-Area TCP/IP Network Simulations. *Journal of Internetworking: Research and Experience 3*, 1 (March 1992), 1–26.

[24] Davidson, S. B., Garcia-Molina, H., and Skeen, D. Consistency in Partitioned Networks. *Computing Surveys 17*, 3 (September 1985).

[25] Deering, S. *Multicast Routing in a Datagram Internetwork.* PhD thesis, Stanford University, 1991.

[26] Deering, S., Estrin, D., Farinacci, D., Jacobson, V., Helmy, A., Meyer, D., and Wei, L. Protocol Independent Multicast Version 2 Dense Mode Specification. *Internet Draft, draft-ietf-pim-v2-dm-03.txt* (June 1999). Work in progress.

[27] Deering, S., and Hinden, R. Internet Protocol, Version 6 (IPv6) Specification. *Request for Comments 2460* (December 1998).

[28] Degermark, M., Brodnik, A., Carlsson, S., and Pink, S. Small Forwarding Tables for Fast Routing Lookups. In *Proceedings of the ACM SIGCOMM'97* (Cannes, France, September 1997).

[29] Digital Island. http://www.digitalisland.com/.

[30] Diot, C., Levine, B. N., Lyles, B., Kassem, H., and Balensiefen, D. Deployment Issues for the IP Multicast Service and Architecture. *IEEE Network* (January/February 2000).

[31] Doar, M. B. A Better Model for Generating Test Networks. *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM)* (November 1996).

[32] Droms, R. Dynamic Host Configuration Protocol. *Request for Comments 2131* (March 1997).

[33] Estrin, D., Farinacci, D., Helmy, A., Thaler, D., Deering, S., Handley, M., Jacobson, V., Liu, C.-G., Sharma, P., and Wei, L. Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification. *Request for Comments 2362* (June 1998).

[34] Faloutsos, M., Faloutsos, P., and Faloutsos, C. On Power-Law Relationships of the Internet Topology. In *Proceedings of the ACM SIGCOMM'99* (Cambridge, Massachusetts, USA, August 1999).

[35] Floyd, S., and Fall, K. Promoting the Use of End-to-End Congestion Control in the Internet. under submission to IEEE/ACM Transactions on Networking.

[36] Floyd, S., Jacobson, V., Liu, C.-G., McCanne, S., and Zhang, L. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking* (November 1997).

[37] Francis, P. Yoid: Extending the Internet Multicast Architecture. White paper, available from http://www.isi.edu/div7/yoid/, April 2 2000.

[38] Govindan, R., and Tangmunarunkit, H. Heuristics for Internet Map Discovery. In *Proceedings of the IEEE Infocom 2000* (Tel-Aviv, Israel, March 2000).

[39] Govindan, R., Yu, H., and Estrin, D. Large-Scale Weakly Consistent Replication using Multicast. Tech. Rep. 98-682, Department of Computer Science, University of Southern California, July 1998.

[40] Gupta, P., Lin, S., and McKeown, N. Routing Lookups in Hardware at Memory Access Speeds. In *Proceedings of the IEEE Infocom'98* (San Francisco, USA, March 1998).

[41] Gwertzman, J. Autonomous Replication in Wide-Area Internetworks. BA thesis, Harvard College, Cambridge, Massachusetts., 1995.

[42] Packet-Based Multimedia Communications Systems. ITU-T Recommendation H.323. http://www.itu.int/.

[43] Handley, M. Session Directories and Scalable Internet Multicast Address Allocation. In *Proceedings of the ACM SIGCOMM'98* (Vancouver, Canada, August 1998).

[44] Handley, M., and Hanna, S. R. Multicast Address Allocation Protocol (AAP). *Internet Draft, draft-ietf-malloc-aap-04.txt* (June 2000). Work in progress.

[45] Handley, M., Kouvelas, I., Speakman, T., and Vicisano, L. Bi-directional Protocol Independent Multicast. *Internet Draft, draft-ietf-pim-bidir-03.txt* (June 2001). Work in progress.

[46] Hanna, S. R., Patel, B. V., and Shah, M. Multicast Address Dynamic Client Allocation Protocol (MADCAP). *Request for Commments 2730* (December 1999).

[47] Helder, D. A., and Jamin, S. Banana Tree Protocol, and End-host Multicast Protocol. CSE-TR CSE-TR-429-00, University of Michigan, Jul 2000.

[48] Hofmann, M. Home page of the Local Group Concept (LGC). http://www.telematik.informatik.uni-karlsruhe.de/~hofmann/lgc/.

[49] Holbrook, H., Singhal, S., and Cheriton, D. Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation. In *Proceedings of the ACM SIGCOMM'95* (Cambridge, MA, USA, August 1995), pp. 328–341.

[50] Holbrook, H. W., and Cheriton, D. R. IP Multicast Channels: EXPRESS Support for Large-scale Single-source Applications. In *Proceedings of the ACM SIGCOMM'99* (Cambridge, Massachusetts, USA, August 1999).

[51] hua Chu, Y., Rao, S. G., and Zhang, H. A Case For End System Multicast. In *Proceedings of ACM SIGMETRICS 2000* (Santa Clara, CA, USA, June 2000).

[52] Internet Assigned Numbers Authority. http://www.iana.org/.

[53] Inktomi. http://www.inktomi.com/.

[54] Intanagonwiwat, C., Govindan, R., and Estrin, D. Directed Diffusion: A scalable and Robust Communication Paradigm for Sensor Networks. In *Proceedings of the ACM Mobicom 2000* (Boston, MA, USA, August 2000).

[55] Jacobson, V. *traceroute(8): Tool for displaying the route packets take to network host.* UNIX manual page.

[56] Jacobson, V. Some Notes on Multicast Scaling and PIM. IDMR Working Group Presentation, 30th IETF, Toronto, Canada, July 1994.

[57] Jacobson, V., and McCanne, S. wb - LBNL Whiteboard Tool. http://www-nrg.ee.lbl.gov/wb/.

[58] Jain, S., and Mahajan, R. Self-Organizing Overlays. http://www.cs.washington.edu/homes/sushjain/overlays.html, May 2000.

[59] Jamin, S., Danzig, P. B., Shenker, S. J., and Zhang, L. A Measurement-based Admission Control Algorithm for Integrated Services Packet Networks (Extended Version). *EEE/ACM Transactions on Networking 5*, 1 (February 1997), 56–70.

[60] Jamin, S., Jin, C., Kurc, A. R., Raz, D., and Shavitt, Y. Constrained Mirror Placement on the Internet. In *Proceedings of the IEEE Infocom 2001* (Anchorage, Alaska, USA, April 2001).

[61] Jannotti, J., Gifford, D. K., Johnson, K. L., Kaashoek, M. F., and James W. O'Toole, J. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)* (San Diego, CA, USA, October 2000).

[62] Jin, C., Chen, Q., and Jamin, S. Inet: Internet Topology Generator. Tech. Rep. CSE-TR-433-00, University of Michigan at Ann Arbor, 2000.

[63] Kasera, S. K., Bhattacharyya, S., Keaton, M., Kiwior, D., Kurose, J., Towsley, D., and Zabele, S. Scalable Fair Reliable Multicast Using Active Services. *IEEE Network Magazine (Special Issue on Multicast)* (January/February 2000).

[64] Kiniry, J. R. Wavelength Division Multiplexing: Ultra High Speed Fiber Optics. *IEEE Internet Computing 2*, 2 (March/April 1998).

[65] Krishnan, P., Raz, D., and Shavitt, Y. The Cache Location Problem. *IEEE/ACM Transactions of Networking 8*, 5 (October 2000), 568–582.

[66] Kumar, S., Radoslavov, P., Thaler, D., Alaettinoglu, C., Estrin, D., and Handley, M. The MASC/BGMP Architecture for Inter-domain Multicast Routing. In *Proceedings of the ACM SIGCOMM'98* (Vancouver, Canada, August 1998).

[67] Lampson, B., Srinivasan, V., and Varghese, G. IP Lookups using Multiway and Multicolumn Search. In *Proceedings of the IEEE Infocom'98* (San Francisco, USA, March 1998).

[68] Levine, B., and Garcia-Luna-Aceves, J. J. Improving Internet Multicast with Routing Labels. In *Proceedings of the 5th IEEE International Conference on Network Protocols (ICNP'97)* (Atlanta, GA, USA, October 1997).

[69] Levine, B. N., Paul, S., and Garcia-Luna-Aceves, J. J. Organizing Multicast Receivers Deterministically According to Packet-Loss Correlation. In *Proceedings of the 6th ACM International Conference on Multimedia* (September 1998), pp. 201–210.

[70] Li, B., Golin, M. J., Italiano, G. F., Deng, X., and Sohraby, K. On the Optimal Placement of Web Proxies in the Internet. In *Proceedings of the IEEE Infocom 1999* (New York, USA, March 1999), pp. 1282–1290.

[71] Li, D., and Cheriton, D. R. OTERS (On-Tree Efficient Recovery using Subcasting): A Reliable Multicast Protocol. In *Proceedings of the 6th IEEE International Conference on Network Protocols (ICNP'98)* (October 1998), pp. 237–245.

[72] Lin, J., and Paul, S. RMTP: A Reliable Multicast Transport Protocol. In *Proceedings of the IEEE Infocom'96* (San Francisco, USA, March 1996), pp. 1414–1424.

[73] Liu, C.-G., Estrin, D., Shenker, S., and Zhang, L. Local Error Recovery in SRM: Comparison of Two Approaches. Tech. Rep. 99-648, Department of Computer Science, University of Southern California, January 1997.

[74] Livingston, M., Lo, V. M., Windisch, K. J., and Zappala, D. Cyclic Block Allocation: A New Scheme for Hierarchical Multicast Address Allocation. In *Proceedings of the First International Workshop on Networked Group Communication* (Pisa, Italy, November 1999), pp. 216–234.

[75] Macedonia, M. R., and Brutzman, D. P. MBone Provides Audio and Video Across the Internet. *IEEE Computer* (April 1994).

[76] Multicast Address Allocation (MALLOC) Working Group. http://www.aciri.org/malloc/.

[77] McCanne, S. Scalable Multimedia Communication with Internet Multicast, Lightweight Sessions, and the Mbone. Tech. Rep. CSD-98-1002, Department of Computer Science, University of California, Berkeley, March 1998.

[78] McCanne, S., and Jacobson, V. Vic: A flexible framework for packet video. In *Proceedings of the ACM Multimedia'95* (November 1995), pp. 511–522.

[79] McCanne, S., Jacobson, V., and Vetterli, M. Receiver-driven Layered Multicast. In *Proceedings of the ACM SIGCOMM'96* (Stanford, USA, August 1996).

[80] McKeown, N. Fast Switched Backplane for a Gigabit Switched Router. white paper, available from http://www.cisco.com/warp/public/733/12000/fasts_wp.pdf.

[81] Medina, A., Matta, I., and Byers, J. On the Origin of Power Laws in Internet Topologies. Tech. Rep. 2000-004, Boston University, January, 20 2000.

[82] Meyer, D., and Lothberg, P. GLOP Addressing in 233/8. Request For Comments (RFC) 3180, September 2001. http://www.ietf.org/rfc.html.

[83] Mills, D. L. Network Time Protocol (Version 3) Specification, Implementation and Analysis. *Request for Comments 1305* (March 1992).

[84] Moy, J. Multicast Extensions to OSPF. *Request for Comments 1584* (March 1994).

[85] Nilsson, S., and Karlsson, G. Fast address lookup for Internet routers. In *Proceedings of the IFIP 4th International Conference on Broadband Communications* (Stutgart, Germany, 1998), pp. 11–22.

[86] Obraczka, K., and Silva, F. Network Latency Metrics for Server Proximity. In *Proceedings of the IEEE Globecom 2000* (San Francisco, California, USA, November 2000).

[87] Palmer, C. R., and Steffan, J. G. Generating Network Topologies that Obey Power Laws. In *Proceedings of the IEEE Globecom 2000* (San Francisco, California, USA, November 2000).

[88] Papadopoulos, C., Parulkar, G., and Varghese, G. An Error Control Scheme for Large-Scale Multicast Applications. In *Proceedings of the IEEE Infocom'98* (San Francisco, USA, March 1998), pp. 1188–1196.

[89] Patridge, C., Carvey, P., Burgess, E., Castineyra, I., Clarke, T., Graham, L., Hathaway, M., Herman, P., King, A., Kohlami, S., Ma, T., Mcallen, J., Mendez, T., Milliken, W., Osterlind, R., Pettyjohn, R., Rokosz, J., Seeger, J., Sollins, M., Storch, S., Tober, B., Troxel, G., Waitzman, D., and Winterble, S. A Fifty Gigabit Per Second IP Router. *IEEE/ACM Transactions on Networking 6*, 3 (June 1998), 237–248.

[90] Pendarakis, D., Shi, S., Verma, D., and Waldvogel, M. ALMI: An Application Level Multicast Infrastructure. In *Proceedings of the 3th USENIX Symposium of Internet Technologies and Systems (USITS)* (San Francisco, CA, USA, March 2001).

[91] Perlman, R., Lee, C.-Y., Ballardie, T., Crowcroft, J., Wang, Z., Maufer, T., Diot, C., Thoo, J., and Green, M. Simple Multicast: A Design for Simple, Low-Overhead Multicast. *Internet Draft, draft-perlman-simple-multicast-03.txt* (October 1999). Work in progress.

[92] Phillips, G., Shenker, S., and Tangmunarunkit, H. Scaling of Multicast Trees: Comments on the Chuang-Sirbu scaling law. In *Proceedings of the ACM SIGCOMM'99* (Cambridge, Massachusetts, USA, August 1999).

[93] Postel, J. Internet Protocol. *Request for Comments 791* (September 1981).

[94] Pusateri, T. Distance Vector Multicast Routing Protocol. *Internet Draft, draft-ietf-idmr-dvmrp-v3-10.txt* (August 2000). Work in progress.

[95] Qiu, L., Padmanabhan, V. N., and Voelker, G. M. On the Placement of Web Server Replicas. In *Proceedings of the IEEE Infocom 2001* (Anchorage, Alaska, USA, April 2001).

[96] Radoslavov, P. MASC implementation (mascd) and MASC simulator (mascsim). http://netweb.usc.edu/masc/mascd/, October 1999.

[97] Radoslavov, P., Estrin, D., Govindan, R., Handley, M., Kumar, S., and Thaler, D. The Multicast Address-Set Claim (MASC) Protocol. Request For Comments (RFC) 2909, September 2000. http://www.ietf.org/rfc.html.

[98] Radoslavov, P., Tangmunarunkit, H., Yu, H., Govindan, R., Shenker, S., and Estrin, D. On Characterizing Network Topologies and Analyzing Their Impact on Protocol Design. Tech. Rep. 00-731, University of Southern California, Dept. of CS, February 2000.

[99] Robust Audio Tool (RAT). http://www-mice.cs.ucl.ac.uk/mice/rat/.

[100] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. A Scalable Content-Addressable Network. In *Proceedings of the ACM SIGCOMM 2001 Conference* (San Diego, California, USA, August 2001).

[101] Rejaie, R., Handely, M., and Estrin, D. RAP: An End-to-end Rate-based Congestion Control Mechanism for Realtime Streams in the Internet. In *Proceedings of IEEE Infocom'99* (New York, USA, March 1999).

[102] Rekhter, Y., and Li, T. A Border Gateway Protocol 4 (BGP-4). *Request for Comments 1771* (March 1995).

[103] Rekhter, Y., and Topolcic, C. Exchanging Routing Information Across Provider Boundaries in the CIDR Environment. *Request for Comments 1520* (September 1993).

[104] Rodriguez, P., and Biersack, E. W. Continuous Multicast Distribution of Web Documents over the Internet. *IEEE Network Magazine* (March/April 1998).

[105] Saltzer, J. H., Reed, D. P., and Clark, D. D. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems 2*, 4 (November 1984), 277–288.

[106] Sharma, P., Estrin, D., Floyd, S., and Jacobson, V. Scalable Timers for Protocol Independent Multicast (PIM). *Internet Draft, draft-ietf-pimwg-PIM-STimers-00.ps* (December 1998). Work in progress, currently available only from ftp://catarina.usc.edu/pub/puneetsh/pim/stimers_id.ps.

[107] Sivakumar, R., Sinha, P., and Bharghavan, V. CEDAR: a Core Extraction Distributed Ad hoc Routing Algorithm. *IEEE Journal on Selected Areas in Communication (JSAC) 17*, 8 (August 1999).

[108] Speakman, T., Farinacci, D., Lin, S., Tweedly, A., Bhaskar, N., Edmonstone, R., Johnson, K. M., Sumanasekera, R., Vicisano, L., Crowcroft, J., Gemmell, J., Leshchiner, D., Luby, M., Montgomery, T. L., and Rizzo, L. PGM Reliable Transport Protocol Specification. *Internet Draft, draft-speakman-pgm-spec-07.txt* (September 2001). Work in progress.

[109] Srinivasan, V., and Varghese, G. Fast Address Lookups using Controlled Prefix Exnansion. *ACM Transactions on Computer Systems 17*, 1 (February 1999), 1–40.

[110] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM 2001 Conference* (San Diego, California, USA, August 2001).

[111] Stoica, I., Ng, T. S. E., and Zhang, H. REUNITE: A Recursive Unicast Approach to Multicast. In *Proceedings of the IEEE Infocom 2000* (Tel-Aviv, Israel, March 2000).

[112] Tangmunarunkit, H., Govindan, R., Jamin, S., Shenker, S., and Willinger, W. Network Topologies, Power Laws, and Hierarchy. Tech. Rep. 01-746, University of Southern California, 2001.

[113] Tangmunarunkit, H., Govindan, R., Shenker, S., and Estrin, D. The Impact of Routing Policy on Internet Paths. In *Proceedings of the IEEE Infocom 2001* (Anchorage, Alaska, USA, April 2001).

[114] Tangmunarunkit, H., Govindan, R., Shenker, S., Jamin, S., and Willinger, W. Network Topologies, Power Laws, and Hierarchy. Work in progress.

[115] Thaler, D., Estrin, D., and Meyer, D. Border Gateway Multicast Protocol (BGMP): Protocol Specification. *Internet Draft, draft-ietf-bgmp-spec-02.txt* (November 2000). Work in progress.

[116] Thaler, D., and Handley, M. On the Aggregatability of Multicast Forwarding State. In *Proceedings of the IEEE Infocom 2000* (Tel-Aviv, Israel, March 2000).

[117] Thaler, D., Handley, M., and Estrin, D. The Internet Multicast Address Allocation Architecture. Request For Comments (RFC) 2908, September 2000. http://www.ietf.org/rfc.html.

[118] Tian, J., and Neufeld, G. Forwarding State Reduction for Sparse Mode Multicast Communication. In *Proceedings of the IEEE Infocom'98* (San Francisco, USA, March 1998).

[119] Touch, J., and Hotz, S. The X-Bone. In *Proceedings of the Third Global Internet Mini-Conference at Globecom '98* (Sydney, Australia, November 1998).

[120] Troll, R. Automatically Choosing an IP Address in an Ad-Hoc IPv4 Network. *Internet Draft, draft-ietf-dhc-ipv4-autoconfig-05.txt* (March 2000). Work in progress.

[121] Tsuchiya, P. Efficient and Flexible Hierarchical Address Assignment. *INET92* (June 1992), 441–450.

[122] University of oregon route views project. http://www.antc.uoregon.edu/route-views/.

[123] USC/ISI. The SCAN Project. http://www.isi.edu/scan/.

[124] Vahdat, A., Eastham, P., and Anderson, T. WebFS: A Global Cache Coherent Filesystem. http://www.cs.berkeley.edu/~vahdat/webfs/webfs.html, December 1996. Department of EECS, University of California, Berkeley.

[125] Vazirani, V. *Approximation Methods.* Springer-Verlag, 2001.

[126] Waldvoge, M., Varghese, G., Turner, J., and Plattner, B. Scalable High Speed IP Routing Lookups. In *Proceedings of the ACM SIGCOMM'97* (Cannes, France, September 1997).

[127] Waxman, B. M. Routing on Multipoint Connections. *IEEE Journal of Selected Areas on Communications 6*, 9 (December 1988), 1617–1622.

[128] Wei, L., and Estrin, D. A Comparison of Multicast Trees and Algorithms. In *Proceedings of the IEEE Infocom'94* (Toronto, Canada, June 1994).

[129] Wen, S., Griffioen, J., and Calvert, K. L. Building multicast services from unicast forwarding and ephemeral state. In *Proceedings of the IEEE OpenArch 2001* (Anchorage, Alaska, USA, April 2001).

[130] Wong, T., and Katz, R. An Analysis of Multicast Forwarding State Scalability. In *Proceedings of the 8th IEEE International Conference on Network Protocols (ICNP 2000)* (Osaka, Japan, November 2000).

[131] Wong, T. H.-T. *Multicast Forwarding and Application State Scalability in the Internet.* PhD thesis, University of California at Berkeley, 2000.

[132] Yavatkar, R., Griffoen, J., and Sudan, M. A Reliable Dissemination Protocol for Interactive Collaborative Applications. In *Proceedings of the Third International Conference on Multimedia '95* (San Francisco, CA, USA, November 1995).

[133] Yu, H. *Design Issues in Large-Scale Application-Level Routing.* PhD thesis, University of Southern California, August 2000.

[134] Zegura, E. W., Calvert, K. L., and Donahoo", M. J. A Quantitative Comparison of Graph-based Models for Internet Topology. *IEEE/ACM Transactions on Networking 5*, 6 (1997), 770–783.

[135] Zhang, L., Michel, S., Nguyen, K., Rosenstein, A., Floyd, S., and Jacobson, V. Adaptive Web Caching: Towards a New Caching Architecture. *3rd International WWW Caching Workshop* (June 1998).

[136] Zhuang, S. Q., Zhao, B. Y., Joseph, A. D., Katz, R., and Kubiatowicz, J. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination . In *Proceedings of Eleventh International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2001)* (Port Jefferson, New York, USA, June 2001).

Appendix A

Multicast Address Allocation

A.1 Introduction

Unlike unicast addresses which are assigned permanently by a central authority, IANA [52], the IPv4 multicast addresses (224.0.0.0–239.255.255.255 for total of 2^{28} or 270 million addresses) are not assigned using the same procedure. Some of the addresses in the multicast address space are designated as "link-wide" or "administratively-scoped," other addresses are assigned for some particular applications, but a majority of the addresses are not pre-assigned.

One possible long-term impediment to the deployment of global multicast is a multicast address allocation strategy. Statically pre-allocating multicast addresses to domains may lead to poor utilization of the overall address space. Some have proposed circumventing the need for multicast address allocation by changing the multicast service model [50, 91]; discussion of these alternatives is beyond the scope of this work. We believe that it is at least plausible to consider the allocation strategy where multicast addresses are dynamically assigned to group initiators. A multicast address is used to uniquely identify a multicast

session, and if a multicast session is not in use (*e.g.*, a one-time teleconference meeting session is over), its multicast address can be reused by another session. By carefully assigning the multicast addresses "on demand," such that each address has a fixed lifetime, the addresses that are not in use can be re-allocated.

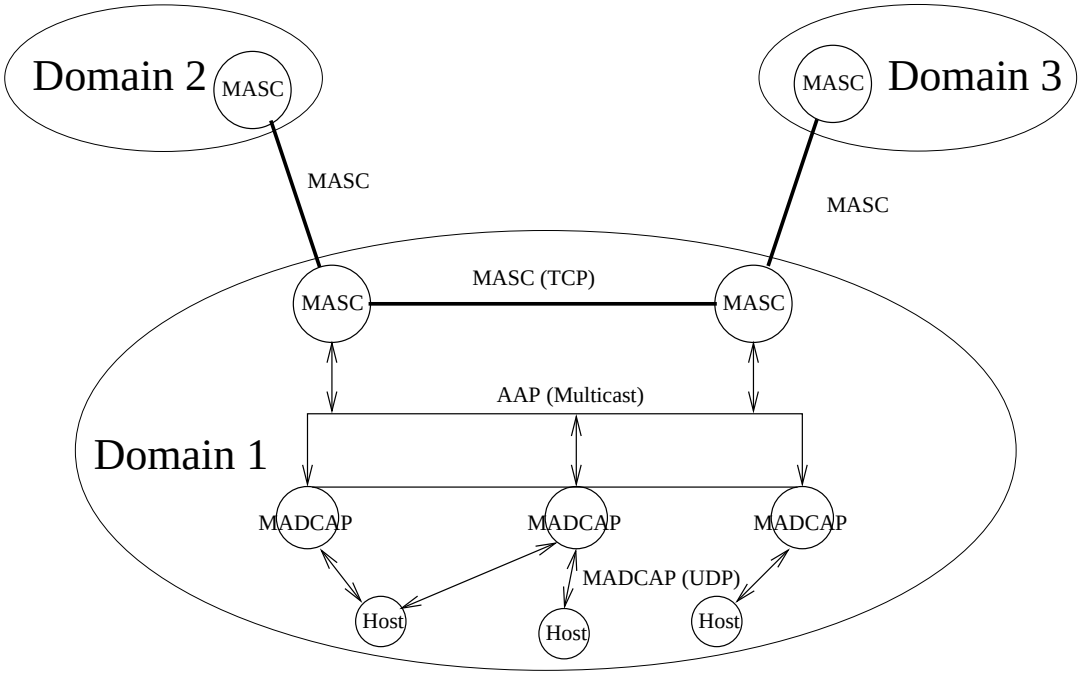


Figure A.1: The malloc architecture.

The work of IETF *malloc* [76] working group is focused on creating a multicast address allocation architecture that can manage and dynamically allocate the addresses. The *malloc* architecture [117] is modular and is separated into three layers: host-server, intra-domain server-server, and inter-domain address allocation (see Figure A.1). At the inter-domain level blocks of addresses are allocated among the domains (the Autonomous Systems), based on the needs of each domain. Within each domain, the addresses allocated to that domain by the inter-domain allocation layer are controlled and managed by a number of

servers. Whenever a host needs one or several multicast addresses to create a new multicast session, one of those servers is used to allocate the addresses to that host.

By separating the architecture into three layers we can use different mechanisms to address the particular problems we may have at different scale. For example, the host-server address allocation mechanisms can be designed to minimize the allocation latency, while the inter-domain address allocation mechanisms can be designed for better utilization of the multicast address space.

In this appendix we describe MASC, the Multicast Address-Set Claim Protocol that has been adopted by IETF *malloc* working group for inter-domain multicast address allocation. MASC is designed for robustness, scalability, address space availability, and efficient address space utilization. MADCAP, the host-server protocol in the *malloc* architecture, and AAP, the intra-domain server-server protocol are described in [46] and [44] respectively. The MASC protocol itself is specified in [97].

The rest of this appendix is organized as follows. In Section A.2 we describe MASC, including the *claim-collide* mechanism it uses, as well as some of the other algorithms. In Section A.3 we present and discuss the simulation results from evaluating MASC. Related work and conclusions are is in Section A.4 and Section A.5 respectively.

A.2 MASC Description

MASC is the protocol used by the multicast address allocation architecture [76] to dynamically allocate multicast addresses at the inter-domain level across Internet. It allocates a set of addresses to each domain. The amount of addresses allocated to a domain is based on the needs of that domain. The allocation is dynamic and has lifetime after which it

expires. The allocated addresses are used by the multicast address-allocation servers within that domain for allocation to the hosts that also belong to that domain.

In Section A.2.1 we begin with an overview of MASC and some of its requirements. In Section A.2.2 we describe the *claim-collide* mechanism used by MASC for address allocation. In Section A.2.3 we describe some of the algorithms that can be used with MASC to achieve efficient address allocation.

A.2.1 MASC Overview

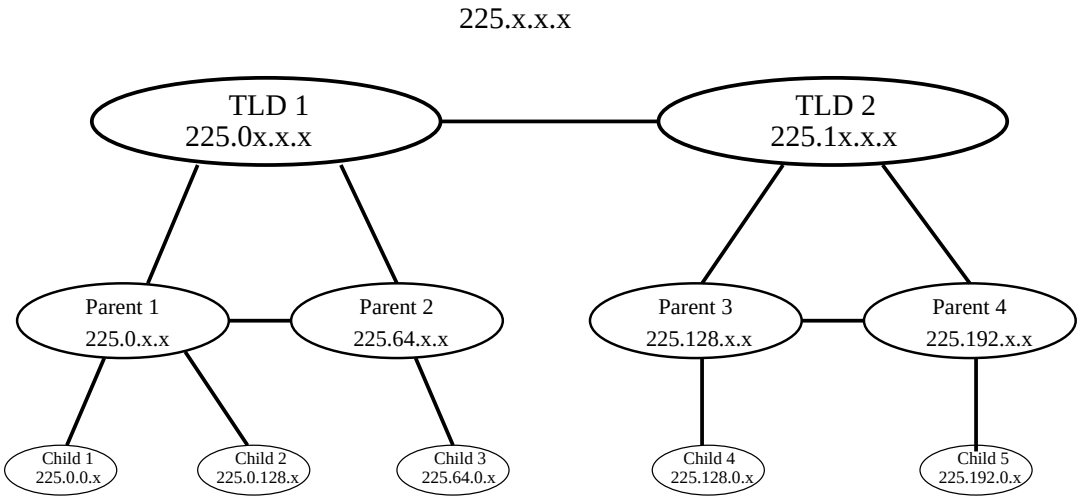


Figure A.2: MASC association of group ranges with AS's.

MASC performs the following functions: (a) allocates sets of addresses to a MASC domain (typically, a MASC domain is congruent with an Autonomous System); (b) advertises those sets to the intra-domain multicast address allocation servers [44, 46], so addresses from those sets can be allocated to the end users/hosts, and, (c) injects the "prefix-domain" associations into the inter-domain routing protocol (*e.g.,* BGP4+ [102]). These associations are used by inter-domain multicast routing protocols such as BGMP [115] to construct

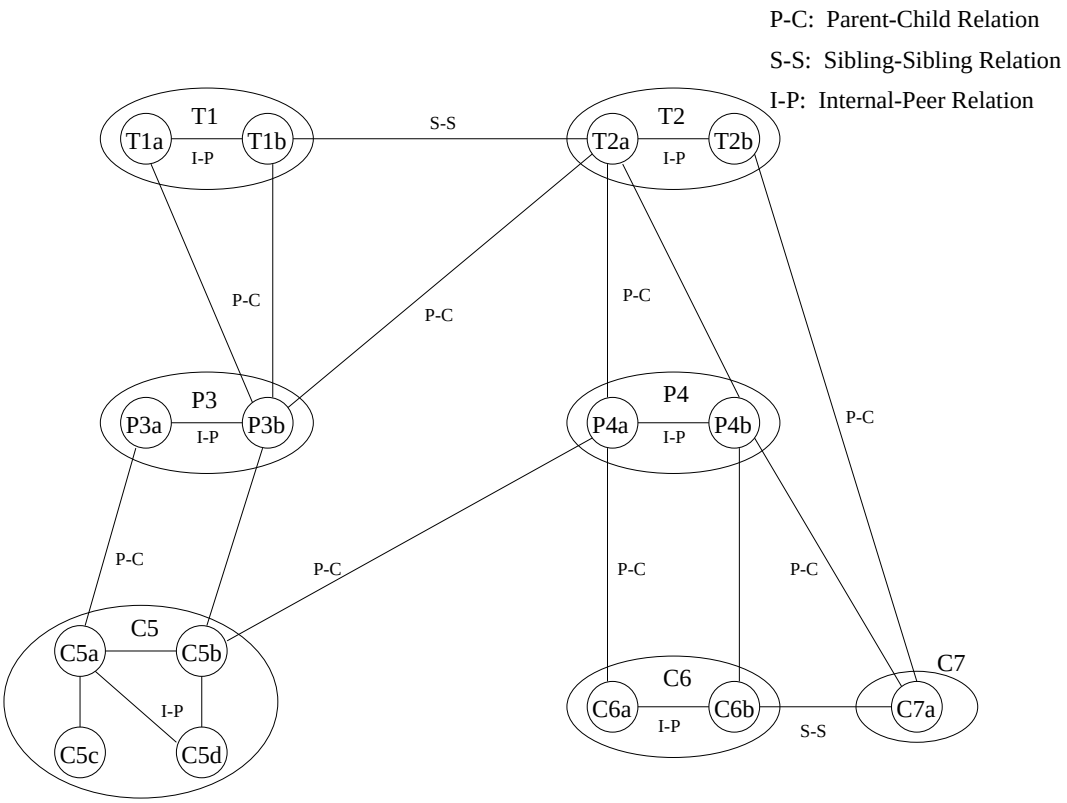


Figure A.3: MASC topology example.

multicast data distribution trees. Each allocation has a fixed lifetime, and after the lifetime expires, the addresses can be reused by other domains.

MASC domains can be configured into a hierarchy (see Figure A.2). At each level in the hierarchy, siblings coordinate to allocate multicast address space that belongs to their common parent. The top-level domains in the hierarchy allocate addresses from the global multicast address space. A MASC domain can have a number of child domains. A child domain allocates addresses from its parent's address space. To improve address availability, a MASC child domain can have more than one parent domains, and can allocate addresses from the address space of any of them (see Figure A.3). For robustness, each MASC domain can be associated with several MASC servers connected in a virtual mesh. The addresses allocated by the MASC servers within a domain are associated with that domain. If a domain has no children domains, all of its address space is used by the intra-domain allocation servers for allocation to the hosts within that domain, otherwise only a part of that space is used for internal allocation.

Some of the requirements for MASC are:

- Addresses should be allocated in an aggregatable manner, because the sets of addresses will be distributed for routing purpose, and scaling the routing information is essential.

- The allocation should be robust, and should not be affected by the network conditions. This requirement is the primary reason for MASC to use the claim-collide based mechanism described in Section A.2.2.

- The allocation latency for the MASC clients (the domain internal servers that use AAP [44] and MADCAP [46] to allocate addresses to hosts) should be nearly zero most of the time, so that clients are not subjected to the claim waiting time for every request for multicast addresses. Only if there is sudden and unusual increase of the demand for addresses, the allocation can be longer.

- Hierarchical allocation and prefix-based address allocation (see below) can result in poor resource utilization. Carefully performed bottom-up demand-driven allocation is required to achieve good utilization.

- Because clients cannot often correctly predict the duration of their multicast sessions, MASC must give clients a reasonable expectation that their allocation's lifetime can be extended, if necessary.

The above requirements conflict with each other, and MASC needs to allocate addresses very carefully to satisfy all of them. Below we describe the mechanisms used by MASC to achieve the desired allocation.

A.2.2 Claim-Collide Mechanism Description

In this section, we describe the *claim-collide* mechanism used by MASC for Internet-wide dynamic address allocation. First we describe the mechanism and how it is used in MASC, and then we present an example that justifies the waiting time used by the claim-collide mechanism.

In MASC, a server that uses the claim-collide mechanism (on behalf of a MASC domain) *proposes* to use some set of addresses by selecting a particular subset of the available

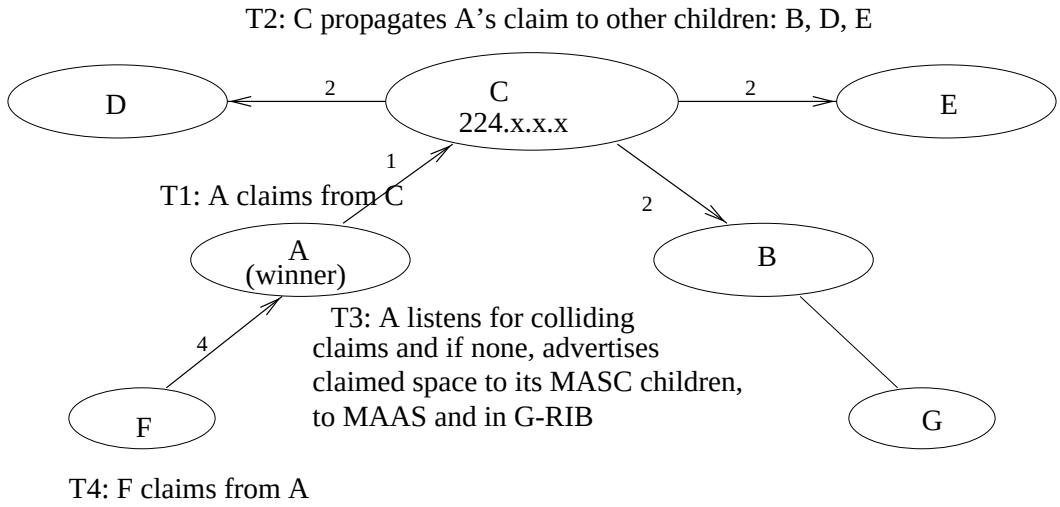


Figure A.4: MASC claim-collide mechanism.

addresses of its parent space (or the global space for the top-level MASC domains), and then sends a claim to all other servers that may claim the same set of addresses. In particular, a claim from a child is sent to the parent, and then to all siblings of the originator of the claim. If no other server opposes the claim (*i.e.,* there is no *collision* due to another server requesting the same set of addresses at the same time), the originator of the claim assumes that it has successfully allocated the claimed addresses.

Figure A.4 contains an example of the claim-collide mechanism used by MASC. When A wants to allocate more addresses, it selects an unused set of addresses from its parent space (224.x.x.x owned by C). Child A *claims* the selected set of addresses by sending a claim to its parent C. Parent C propagates the claim from A to the rest if its children, B, D, and E. After A originated the claim, it waits for sufficient amount of time (see below) and listens for conflicting claims by some of its siblings (B, D, and E). If no such claim is

received, *i.e.*, if there is no *collision*, then A assumes that it can use the set of addresses it has claimed. If a colliding claim is received, then A would eventually *back-off* by selecting a new set of addresses and initiate a new claim. After A has allocated a set of addresses, this set is announced to its own children (F in this particular example), and then those children can claim themselves.

The amount of time to wait after a claim should be sufficient enough such that there is time for a claim to propagate to all siblings that may claim the same set of addresses, and for a colliding claim, if any, to come back. Network partitions may prevent a claim from reaching all siblings, therefore the claim wait period should be sufficiently long enough to outlive potential network partitions. Because MASC is designed to be used in an environment with non-negligible probability for network partitioning, its claim period has the conservative value of 48 hours. In most cases it takes less than 24 hours for an Internet Service Provider to fix a network-related problem such as intra or inter-domain partitioning. Hence, waiting for 48 hours should be long enough to ensure the end-to-end claim propagation, and therefore the uniqueness of the allocation.

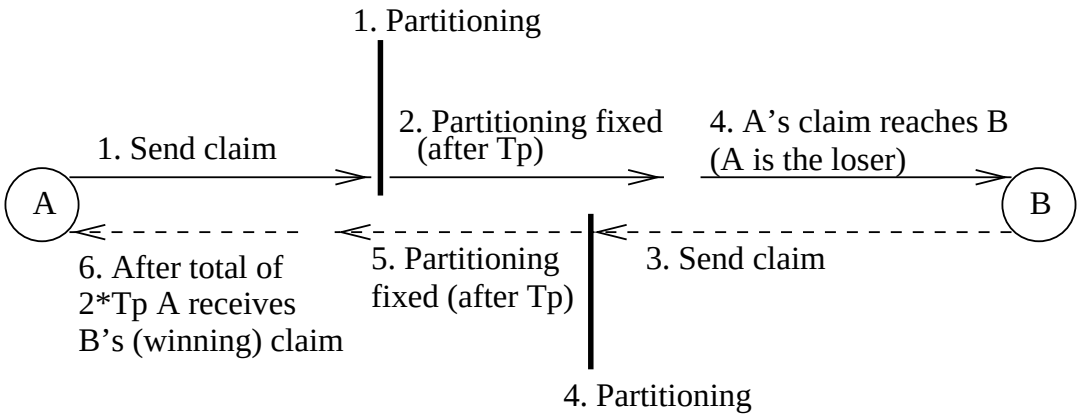


Figure A.5: An example of collision detecting latency that takes $2 * T_p$ time units.

The reason that the wait time after a claim (the *claim wait time*) is 48 hours, i.e. twice the worst amount of time it would typically take to fix a network partition, can be explained by the example in Figure A.5. If A sends a claim toward B, and if it takes up to T_p time units to fix a partition, it can take up to T_p for that claim to reach B (we assume that either A or the intermediate MASC nodes can detect a partition, and a claim is resent after the partition is healed). However, a claim originated by B can also take up to T_p to reach A, therefore if there are two partitions in the network, one after another, it may take up to $2 * T_p$ for A to receive a colliding claim from B. Obviously, there is no guarantee that a network partition will disappear within some period of time, but for practical reasons a conservative value of 24 hours would be reasonably sufficient.

An alternative solution to claim-collide would be to use *request-response* mechanism where allocation can be considered successful only after explicit acknowledges are received from all siblings. Indeed, if there is no network failure, request-response based allocation may be faster than a claim-collide based allocation. However, a sibling failure may block the allocation process, therefore robustness of allocation becomes a complicated issue. On contrary, claim-collide can be used to achieve better robustness by trading-off longer allocation latency, and by simplifying significantly the allocation mechanism.

A.2.3 MASC Algorithms

A MASC server needs to use some local algorithms to carefully select the addresses to claim. By using such algorithms and by pre-allocating addresses, a server can reduce the collision probability (and therefore the allocation latency as well), and at the same time can maintain reasonable address space utilization.

To reduce the allocation latency, a MASC server can (a) use pre-allocation, and (b) avoid collisions. The basic idea behind pre-allocation is that a server claims resources in advance, so a client will not need to wait to obtain the resource. The pre-allocation strategy must, however, be carefully designed. If the server pre-allocates too many addresses without using them the address space utilization will be very low. If the server does not claim enough addresses in advance, many clients will still have to wait for long time.

One simple pre-allocation strategy that we discuss here attempts to keep ahead of client demand by dynamically tracking client request patterns. A MASC server constantly monitors the demand for addresses from its children (or intra-domain servers), and attempts to predict what would be the address usage after 48 hours, the MASC claim wait time. Only if the available addresses will be used up within 48 hours, the server claims more addresses in advance, so it would be prepared to meet the demand later.

Designing a pre-allocation strategy is not sufficient. Collisions can result in allocation failures and can increase significantly the latency. To reduce the probability for collisions, a server needs to keep state about addresses that are currently not in use. For reasons of scale, this state needs to be aggregatable. To reduce the amount of state, the addresses allocated by MASC are aggregated into prefixes (*e.g.,* the set of all 256 addresses that start with 224.1.2 are described with address 224.1.2.0 and mask 0xffffff00, also noted as 224.1.2/24). Alternative solutions, such as using address ranges or non-contiguous masks are more complicated and may require changes to the routing protocols.

The first prefix a MASC server chooses is selected at random. If a server chooses to allocate another, smaller prefix, then, instead of doubling the size of the first one, a second "neighbor" prefix is chosen. For example, if prefix 224.0/16 was already allocated,

and the server needs 256 more addresses, the second prefix to claim will be 224.1.0/24. If it needs more addresses, the second prefix will eventually grow to 224.1/16, and then both prefixes can be automatically aggregated into 224.0/15. Only if 224.0.1/24 could not be allocated, a server will choose another prefix (eventually random among the unused prefixes). The particular prefix selection algorithm used in MASC is the so called *reverse-bit expansion algorithm.*[1] For example, if we had 3-bit address space, the addresses assigned in reverse-bit ordering are 000, 100, 010, 110, and so on. Therefore, if we had A, B, and C claiming/allocating one address at a time, the addresses allocated to each of them would look like:[2]

3-bit space:	xxx
A:	000
B:	100
A:	00x
B:	10x
C:	010
C:	01x

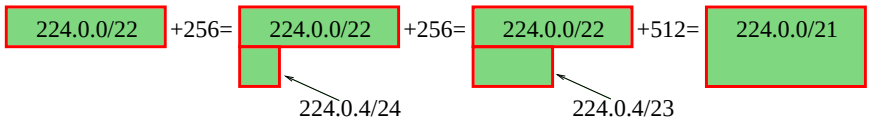
To increase the resource utilization, a prefix is implicitly returned to the parent after its lifetime expires. If the demand for addresses does not decrease, then a MASC server re-claims the prefixes it has allocated before their lifetime expires. If the demand for addresses decreases, the server implicitly returns the unused prefixes, or re-claims some smaller sub-prefixes. Also, because every prefix is a power of two, if a node tries to allocate just a

[1] Courtesy of Dave Thaler; inspired by Kampai [121].
[2] An alternative scheme called *Cyclic Block Allocation* is proposed in [74].

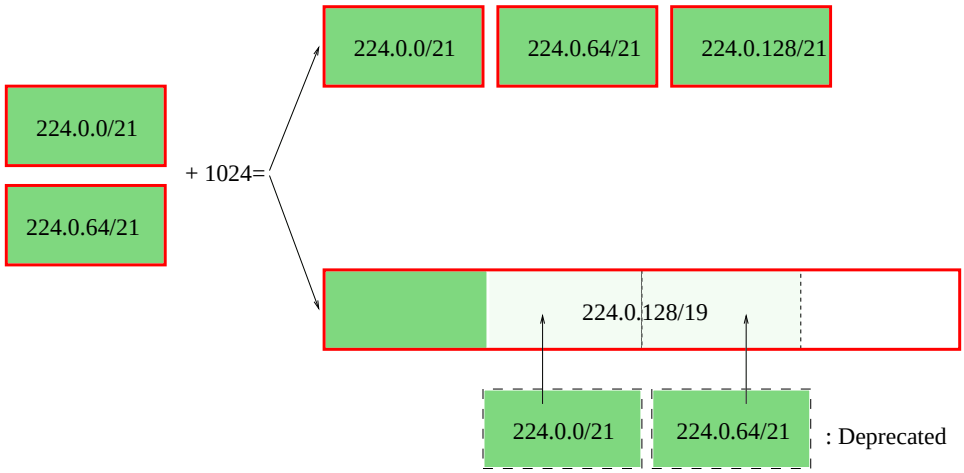
single prefix, the utilization at that node can be as low as 50%. To improve the utilization, a server can have more than one prefix allocated at a time (typically, each of them with different size).

If the number of allocated prefixes increases above some threshold, and none of them can be extended when more addresses are needed, then, to reduce the amount of state, a MASC server claims a new larger prefix and stops re-claiming the older non-expandable prefixes. This, obviously, increases the prefix flux. Similarly, if a server tries to keep the utilization at some very high level by allocating a large number of prefixes (each with different size), a smaller increase or decrease of the demand would result in allocating or "releasing" prefixes. A large number of allocated prefixes also creates additional fragmentation, hence a server should allocate just few prefixes.

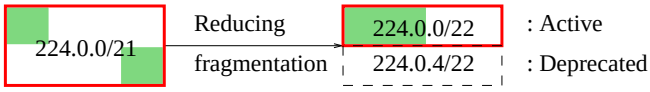
Figure A.6 shows an example of how the addresses for allocation are selected. When the demand for addresses increases, the MASC server would try to add the new prefixes incrementally, by doubling and aggregating the address prefixes it already owns (see Figure A.6(a)). If the existing prefixes cannot be doubled in size, a MASC server may choose to select a new, large enough prefix that can accommodate the current demand. Then the server would gradually "move" older allocations to the new prefix by not renewing those older prefixes (see Figure A.6(b)), therefore effectively reducing the space fragmentation. Similarly, if the currently allocated address prefix is under-utilized, the MASC server may try to aggregate the used addresses by "moving" the fragments within a single block (see Figure A.6(c)).



(a) Incremental increase



(b) Adaptive increase



(c) Adaptive decrease

Figure A.6: Increasing and decreasing the allocated addresses.

Finally, we should note that the same algorithms can be used in a hierarchy if the amount of state becomes a scalability issue, or if there are other reasons such as management or administrative control. In the hierarchy, the servers at the lower level of the hierarchy are clients of the servers at the higher level. In the next section we show through simulations how the allocation performs in a flat topology, and in a hierarchy.

A.3 Simulation Results

Our primary goal was to evaluate the design of the claim-collide mechanism and the set of algorithms that are used in MASC. In our evaluation we were interested primarily in the allocation latency and address space utilization, and how robust the architecture is to network partitions. We also wanted to evaluate MASC in term of amount of state and state flux. As part of our evaluation we considered hierarchical allocation as well.

The simulator we used is based on our MASC implementation [96]. In fact, the MASC-specific part of the implementation was used without modification in the simulator.

A.3.1 Methodology

We evaluated three different allocation topologies: a flat topology, one in which servers form a two-level hierarchy, and one in which they form a three-level hierarchy. For simplicity, each child node had only one parent. We expect that, if nodes have more than one parent, the overall allocation latency would be less than the latency observed in our simulations. This is because a child would be more likely to acquire addresses from the least address-constrained parent. Because the allocation algorithms (see Section A.2.3) are applied independently with regard to each parent, the utilization at each child will be similar as if

it had only one parent. The flat topology had 100 nodes; the two-level hierarchy had 10 parent nodes and each parent had 10 children; the three-level hierarchy had 5 nodes at the top, each with 5 children, and each of those 25 children had 4 children on its own. Hence, all topologies had the same number of leaf nodes.

The same type address demand pattern for multicast addresses, parameterized by request size, request lifetime, and inter-request time, was applied to each leaf node. In the simulation runs we dynamically varied these parameters to simulate varying demand. Each request lifetime was fixed to 32 days and each request size had a fixed value that could change over time. The inter-request time was always a random value between 0 and twice the average inter-request time, which also changed over time. The parameters were changed according to Table A.1.

Time (hours)	0	500	1000	1500	2000	5000	15000
Request size	1024	1024	1024	2048	2048	1024	2048
Ave. inter-request time (hours)	32	24	16	32	24	24	24
Lifetime (days)	32	32	32	32	32	32	32

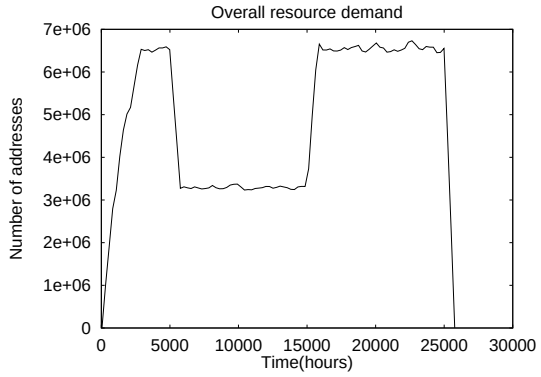
Table A.1: Demand parameters change over time.

We were interested primarily in the allocation latency among all MASC servers, and in the long term overall resource utilization (the amount of all addresses allocated to the clients vs. the amount of all addresses allocated by all servers). We also measured parameters such as the average number of prefixes allocated to a server, and the average number of prefix changes. The former defines the amount of state in the servers and the address space fragmentation; the latter defines the flux of the addresses allocated to the users, or the change in associated routing information. In our simulations we defined prefix change

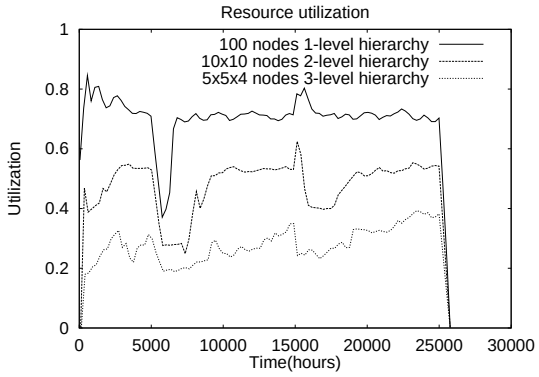
as the allocation of a new prefix, or the expansion or shrinking (including complete release) of an already allocated prefix, hence the results are closer representation of the flux in the association of set of addresses with a MASC domain.

Figure A.7(a) shows the shape of the overall demand pattern. Because we wanted to stress the allocation, initially we started increasing the demand rapidly, later at time 5000 we suddenly decreased it by half, and finally at time 15000 we increased it twice. Finally, on average, every 5 hours, a random link would go down for 23 hours. Because a child node was connected by a single link to its parent, and all top-level nodes were connected in a star, a single link failure could partition the topology. By using a relatively high rate of link failures, and by using topology that is not robust against partitioning, we wanted to stress the robustness of the allocation mechanism, and wanted to investigate how network failures would affect the allocation latency.

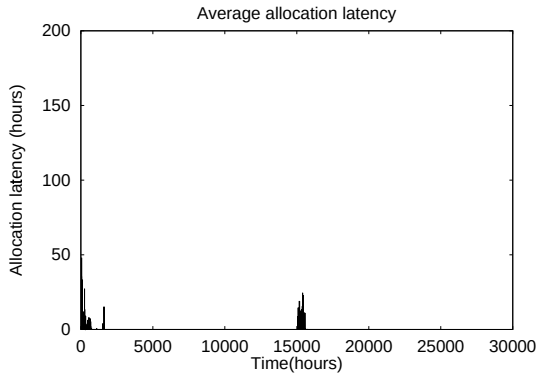
In our simulations the address utilization target was 70-90% per node; the target number of prefixes was three prefixes per node. A higher utilization target could make the pre-allocation less efficient, and would increase the allocation latency. A higher target number of prefixes would increase the state per node; a lower target number would decrease the utilization per node, because, within each prefix there is a likelihood of a 50% internal fragmentation.



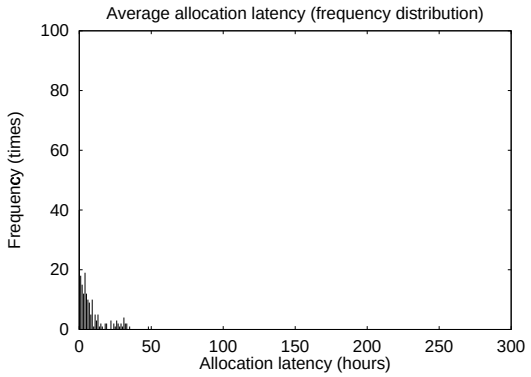
(a) Overall resource demand pattern



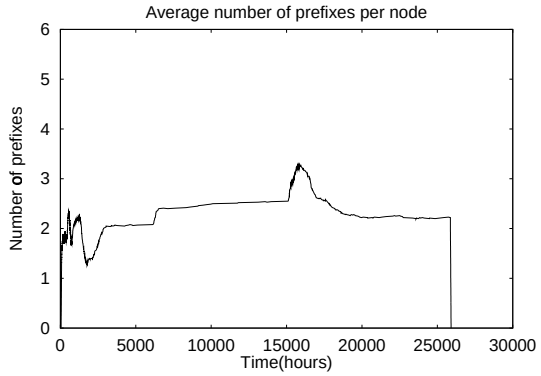
(b) Overall resource utilization



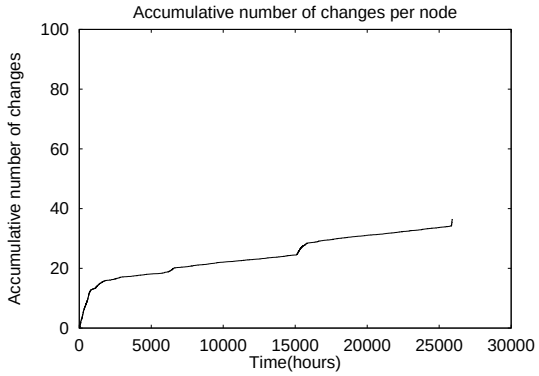
(c) Average allocation latency



(d) Average allocation latency(frequency distribution)



(e) Average number of prefixes



(f) Prefix changes (accumulatively)

Figure A.7: Flat topology of 100 nodes.

A.3.2 Simulation Results

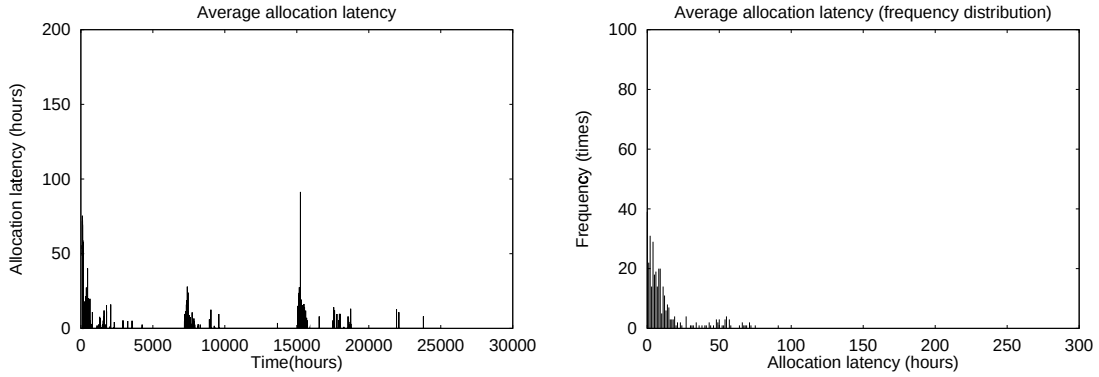
Figure A.7(b) shows the overall resource utilization for flat topology, two, and three level hierarchy. The long-term utilization in the flat topology is within the target of 70-90%. Obviously, each additional level in the hierarchy will multiplicatively decrease the utilization by 70-90%, hence the utilization for the two and three-level hierarchy is lower.

Figure A.7(c) shows the average allocation latency among all requests for the flat topology averaged with time window of 1 hour. The frequency distribution is shown in Figure A.7(d). Without pre-allocation we would expect that the allocation latency will always be 48 hours plus the time to resolve collisions. Because of pre-allocation however, most of the time the allocation latency is close to zero. Only when there was sudden change of the demand, the allocation latency was non-zero, simply because such changes are difficult to predict. One explanation is that the particular utilization targets we chose did not have enough slack to accommodate the surge in demand. We do expect that in practice the demand will gradually increase, and if there are such sudden changes of the demand, they would be triggered by predictable events (*e.g.,* release of extremely popular multicast application). In that situation, the system administrators can take some actions in advance such as reducing the target utilization so the MASC servers will pre-allocate more addresses.

We should note that only during the startup the amount of collisions were significant such that the allocation latency of some requests was longer than 48 hours. This is primarily because of two factors. First, a network partition increases the collision discovery latency, hence it took longer to back-off and claim a new prefix. Second, when the nodes that had backed off claimed new, non-adjacent prefixes, they both used an identical, deterministic algorithm for this (the reverse-bit ordering algorithm described in A.2.3). As a result, they

selected the same prefix again. Small randomness should be used to significantly reduce the collision probability, and therefore the allocation latency.

The average allocation latency results for two and three-level hierarchy are shown in Figure A.8 and Figure A.9 respectively. It is not a surprise that the allocation latency is longer than the flat topology. If we ignore the collisions, we would expect that the allocation latency would be no longer than $HierarchyDepth * 48hours$, hence for the two and three-level hierarchy it should be 96 and 144 hours respectively. Indeed, with few exceptions, this was the case. We visually observed that during the startup period, and at time 15000 in case of three-level hierarchy, there were few occasions of latency higher than expected. As in the flat topology, collisions were one of the reasons, but there were two more factors. First, if a child server needed to claim more addresses from its parent's space, but the link to the parent was down, the child did not originate the claim before the link comes up; that added up to 23 hours to the allocation latency. The second reason applies only to the three-level (or any hierarchy with more than two levels) hierarchy. If a parent's allocation is behind the demand, a child might send a claim-hint for more space to that parent. For simplicity of implementation, and to reduce the multi-level dependency, this hint had only one-time meaning to the parent. The parent immediately claimed more space based on that hint, but it might itself had to send a claim-hint to the upper level (level 1) server. However, after the level 1 server obtained and advertised more space to the level 2 servers, our parent could not immediately claim more space, unless it had received more claims from its children (including claim-hints). Hence, a leaf child server should periodically resend claim-hints to its parent, until the parent has enough space. In our simulations the leaf servers were sending the claim-hints once in 48 hours, hence this



(a) Average allocation latency

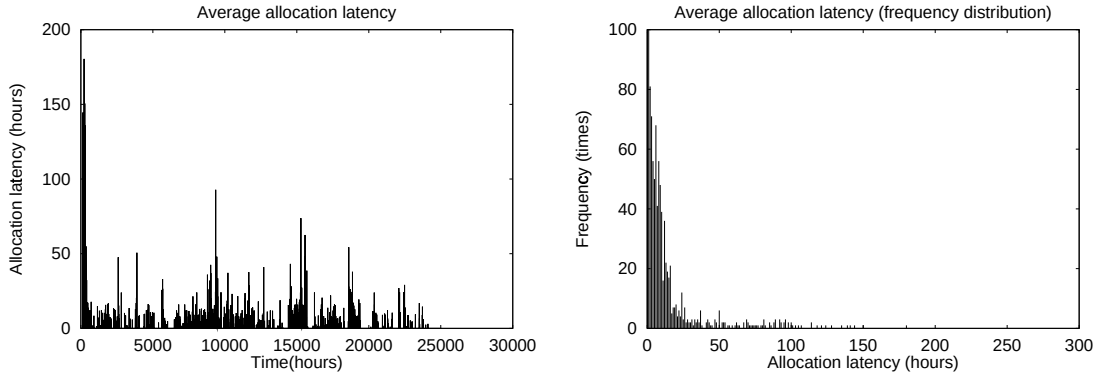
(b) Average allocation latency(frequency distribution)

Figure A.8: Two-level topology with 100 leaf nodes nodes.

additionally increased the allocation latency. Investigating the relationship between this frequency and the allocation latency is part of our future work.

Figure A.7(e) shows the average number of address prefixes per server for the flat topology. We can see that the number of prefixes is within our target of three prefixes per node. Figure A.7(f) shows the accumulative number of average prefix change per node. We do not have a hard-number target for the prefix changes, because this number depends primarily on the dynamics of the resource request demand pattern (in particular, a sudden increase of the demand). If we exclude the changes during the startup, we can compute that the average change of prefix was relatively low: on the order of one change in 50 days. The results for the two and three-level hierarchy were similar, except that the average number of changes at the leaf nodes were respectively two and three times higher.

The results of running the same set of simulations, but without links failures were similar, except that during largest contention for resource (during startup and at time 15000) was slightly shorter. This is primarily due to the smaller probability for collision.



(a) Average allocation latency

(b) Average allocation latency(frequency distribution)

Figure A.9: Three-level topology with 100 leaf nodes nodes.

We also run the same simulations using exponential and Pareto inter-request time distribution instead of uniform random distribution.[3] The results with exponential inter-request time distribution were similar to the results with uniform time distribution, except that the overall address utilization was a little bit lower, and the accumulated allocation latency was approximately 15-25% more. With Pareto distribution however, the overall address utilization was approximately up to 15% lower (*e.g.,* for the three-level hierarchy hierarchy the overall resource utilization was approximately 20%, compared to 35% with uniform inter-request time), and the accumulated allocation latency and the prefix flux were in some cases three times more. This is not a surprise, because the variation of Pareto is much larger, and it is much more difficult to predict such variations of the demand and allocate enough addresses in advance without compromising the address utilization.

In our simulations each request lifetime was fixed at 32 days. Because the allocation is completely demand driven, if the lifetime was longer, the servers would allocate the

[3]We should note that the inter-request time defines primarily the demand for addresses, and increasing or decreasing its average value is equivalent to increasing or decreasing the demand.

resources for longer lifetime. Longer allocation lifetime however means that it will take longer to expire a prefix that is not in use. Hence, if the demand suddenly decreases (as in time 5000 in our simulations), it would take longer for the servers to "release" the unused resources, and therefore the utilization will be below the target for longer period of time. If each request lifetime was much shorter, then the unused addresses will be released sooner, and the utilization will not be compromised, but then the allocation is more sensitive to short-term demand variations, and the flux of the allocated addresses will be much larger.

A.4 Related Work

In [24] the authors classify distributed resource allocation mechanisms into two classes: pessimistic and optimistic approaches. If there is a network failure, the pessimistic approach simply prevents the servers from allocating resources. In contrast, the optimistic approach allows them to allocate resource even if some of the other servers are not reachable; the assumption is that conflicting allocation will be rare, but if it happens, it can be resolved after the partition heals. The claim-collide mechanism for address allocation we describe in this paper is in some sense a pessimistic approach. It assumes that if there is network partitioning, it is quite likely that immediate allocation will result in inconsistency. On the other hand, because a server does not expect explicit positive acknowledgments from the other servers, it has some similarity with the optimistic approaches.

The Mbone multicast session directory, *sdr* [43][4] is used to create and manage multicast sessions, and one of its functions is to allocate a multicast address to each session. It is the first architecture widely used over the Internet that allocates the multicast addresses

[4] *sdr* was based on *sd* (Session Directory), written by Van Jacobson.

without centralized authority and without explicit acknowledgment messages among the participants. Sdr also used a claim-collide mechanism, but it operates at the granularity of a single address and much shorter claim wait time. If the global address space utilization is low, then sdr can allocate addresses with high probability for uniqueness. Sdr however does not scale if the address space utilization is high, or if there is a large number of participants.

Announce-listen communication [13] is a paradigm for robust coordination using multicast. In Announce-Listen, the clients use multicast to announce there requests. The providers listen to these requests and to the responses of other providers, and take the appropriate actions such as granting or demanding tokens back from the clients. Claim-collide is an instance of announce-listen communication, to the extent that servers announce their intent to acquire resources (the multicast addresses), and other servers listen passively to claims, responding only to generate collisions when necessary.

A claim-collide based mechanism is used for link-local unicast address allocation [120]. Typically, a host that does not have configured IP address will use DHCP [32] to get one. If there is no DHCP server running, or if it is down, the host can choose an unicast address at random from some well known range, then use a claim-collide-like mechanism (ARP "probes") to ensure that the address is not used by other hosts on the same LAN. The allocated address then can be used to communicate with the range of that LAN. In this example claim-collide is preferred because of its simplicity (the hosts are their own servers), and because it does not need any external servers to operate.

A number of multicast routing protocols ([4, 33, 26, 115]) need to elect a single router-forwarder on shared LANs to avoid data duplication. A router sends an explicit "claim" message with metric and preference on a LAN. Other routers might also "claim" the right

to forward packets for same *(source,group)* on that LAN. The router with best metric and preference is considered the winner; the rest of the routers "back off" by removing the interface to that LAN from the outgoing interfaces set on the particular forwarding state.

A.5 Conclusions

Due to the scarcity of the IPv4 multicast addresses, it is desirable to use dynamic address allocation with explicit lifetime. The work of the IETF malloc working group proposes one such architecture, composed of three-layer allocation scheme: host-server, intra-domain server-server and inter-domain address allocation.

In this Appendix we describe and evaluate MASC, the Multicast Address-Set Claim Protocol, that is used for inter-domain address allocation within malloc. MASC uses several techniques to achieve robust and efficient address allocation. To avoid synchronization-related issues associated with explicit request-response mechanisms, MASC servers use claim-collide mechanism to allocate a set of addresses. The advantage of the claim-collide mechanism is that it is much simpler, and is more robust to network failures. However, this is achieved by trading-off longer allocation latency. To levigate the impact of longer allocation latency, MASC servers trace address demand and use pre-allocation. To improve the address utilization, and at the same time to reduce the amount of state and address fragmentation, MASC uses a combination of techniques such as prefix-based allocation, reverse-bit ordering selection algorithm, and multi-prefix allocation per domain.

We use simulations to demonstrate that MASC can perform reasonably well even with highly unpredictable demand for addresses. In our simulations we use flat MASC server topology, as well as two and three level hierarchy. The results show that MASC can perform

reasonably well, but increasing the levels of the hierarchy reduce the address utilization, therefore for practical purposes the hierarchy should be limited to two levels.

There are a number of multicast-related deployment issues that need to be solved [30] to achieve Internet-wide multicast, and multicast address allocation is one of them. On the other hand, each additional multicast-related protocol adds complexity to the system, which can be another deployment-related issue. More recently, a simple static multicast address prefix-allocation mechanism has been proposed for inter-domain address allocation [82], where the prefix of multicast addresses allocated to an AS is algorithmically derived from that AS number. Such mechanisms are very attractive because of their simplicity, but they limit the number of addresses that can be allocated to each domain. If the demand for multicast addresses is relatively small, then we could use AS number-derived allocation schemes, and we can avoid the complexity of inter-domain dynamic address allocation. If the demand for multicast addresses is significant, then we would need some dynamic allocation mechanism such as MASC. One possible solution is to use a combination of static and dynamic address allocation. For example, we can use MASC to manage some fraction of the address space, and the rest of the address space can be divided based on the AS numbers. Therefore, only domains that have larger demand for addresses would have to use MASC to obtain more multicast addresses. The rest of the domains would not be forced to deploy MASC if they do not need more multicast addresses. Thus, the complexity of dynamic inter-domain address allocation is added to the network only where it is needed.