Lorikeet: An Efficient Multicast Protocol for the Distribution of Multimedia Streams

Justin Viiret

Thesis submitted for the degree of

Doctor of Philosophy

in

Electrical and Electronic Engineering

at

The University of Adelaide

(Faculty of Engineering, Computer and Mathematical Sciences)

School of Electrical and Electronic Engineering

June 4, 2007

Contents

List of Tables

List of Figures

Signed Statement

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I consent to this copy of my thesis, when deposited in the University Library, being available for loan and photocopying.

SIGNEL .. DATE: 26/07/2007

Acknowledgements

A PhD is an enormous piece of work with a great many moving parts, and I have been very fortunate to have so many people help me with my work on Lorikeet. I am very grateful to all of them for sharing their experience, time and support; I certainly wouldn't have got there on my own.

Guiding my way from the first step to the last have been my supervisors at the University of Adelaide, Nigel Bean, Michael Rumsewicz and Reg Coutts. Their expertise and accessibility have prevented (or, at least, explained and corrected) many missteps along the way and I thank them wholeheartedly for their friendship and support over last few years. I could not have asked for better mentors.

I have been generously supported by scholarships from both the University of Adelaide and the CRC for Smart Internet Technology, without which I could not have worked full-time on my research. Both organisations also offered invaluable training, support and other facilities. The South Australian Partnership for Advanced Computing kindly gave me access to their Hydra cluster computing resources, making the simulation of larger networks possible in units of time smaller than weeks. Special thanks are also due to the staff of TRC Mathematical Modelling, who have looked after me and provided me with so much (the desk nearest the coffee machine, for starters!) over the last four years.

Matthew Sorell taught me about multimedia coding and network applications when I was a callow youth, and later was kind enough to bring me on board to mark assignments, tutor and occasionally lecture with him on those topics. I enjoyed myself immensely and appreciate his assistance and enthusiasm. Matthew Roughan and Olaf Maennel answered my questions about router design, BGP and many other things besides.

My fellow students in the TRC provided comradeship and many other things: from a captive audience for my expositions on technology, film and photography through to much-needed moral support and academic assistance. Andre Costa provided a helping hand and a voice of experience when I was new to postgraduate work and still more than a little green; Jeremy McMahon's enviable ability to deconstruct and then calmly obliterate a problem helped me out of several tight spots; and Mathew Kinghorn was always ready to discuss a protocol, talk film, eat lunch, or some combination of the above. Good luck with everything, guys, and thanks for all your help.

So many of my friends have helped me out in the last few years, remaining supportive and understanding in the face of my occasional weird obsessions and irregular hours. Milton Yates, Adrian Jones, Beth Nosworthy, Matthew Berryman, Kit Macfarlane, Jane Miller and Bryn Roberts, thank you for all the lunches, coffees and dinners. Nic Cottrell and Andrew Clarke have remained good friends of mine overseas, always been willing to talk or listen when I needed them. I would also like to thank David Watts and Tom Long for all the coffee and photographs, Carl Estella for his friendship and his patient instruction in the simplest elements of *Ba Gua Zhang*, and Marty Deveney for his willingness to share his vast experience in both photography and postgraduate life.

My family have tirelessly supported me throughout my years at University, prodding me into action when I ran out of steam and encouraging me when I needed reassurance. I am indebted to my mother and father, my sisters and my grandparents for their love and dependability.

Aditi Rao has been a wellspring of enthusiasm and affection as I have worked on my thesis, and I have always been inspired by her clarity of thought and expression. I am very grateful for her support while she was in Adelaide, and for her confidence and constant encouragement from afar.

Abstract

Internet Protocol multicast has been standardised since the late 1980's, but is yet to be extensively deployed by most Internet Service Providers. Many organisations are not willing to bear the additional router CPU load and memory requirements that multicast entails, and the IP multicast suite of protocols requires deployment on every router spanned by the multicast group to operate. Additionally, these protocols are predominantly designed for the general case of multiple-source, multiple-receiver transmission and can be complex and inefficient to use in simpler scenarios.

Single-source streaming of multimedia on the Internet is rapidly becoming a very popular application, and is predominantly being served by content providers using simultaneous unicast streams. A multicast transmission protocol designed for this application that can operate without requiring a widely deployed IP multicast infrastructure has the potential to save content-providers and network service providers significant amounts of bandwidth. This protocol should provide packet duplication and forwarding capabilities on routers in the network, rather than pushing this functionality to the receivers themselves, requiring them to become part of the multicast infrastructure.

We describe Lorikeet, a new protocol for the multicast distribution of multimedia streams from a single source. This protocol builds its multicast tree from the source, discovering routers that support the protocol in the network and using them to provide branching in the tree. The tree itself is managed in a decentralised fashion, with joining receivers finding parent routers through a limited, recursive search of the tree. On a participating node, information about the tree's structure is limited

to the addresses of that node's children and its path through the tree back to the source. Unlike most other multicast protocols, a new receiver is connected to the tree using its forward path from the source and packets are delivered through the tree via hop-by-hop delivery over unicast connections between nodes. Lorikeet also actively maintains the tree structure using a localised rearrangement algorithm triggered by a topological change in the tree structure. This rearrangement allows the tree to remain efficient in the face of changes to the receiver population, which can change the shape of the tree over time.

Lorikeet is designed to operate with no further protocol support than that provided by existing Internet unicast protocols. It requires none of the standard IP multicast infrastructure, such as Class D group addressing. Its use of unicast connections between nodes allows it to be deployed incrementally on the network, and its behaviour will degrade to simultaneous unicast when no routers that support the protocol are present at all. However, significant performance gains can be achieved even when there are only a few supporting routers present in the network: Lorikeet produces trees with half the cost of a unicast tree when just 10% of routers are Lorikeet-capable. Lorikeet's tree construction and rearrangement algorithms generate multicast trees of comparable total cost to those created by algorithms of considerably higher message complexity, such as those that employ exhaustive searches of the tree during joins.

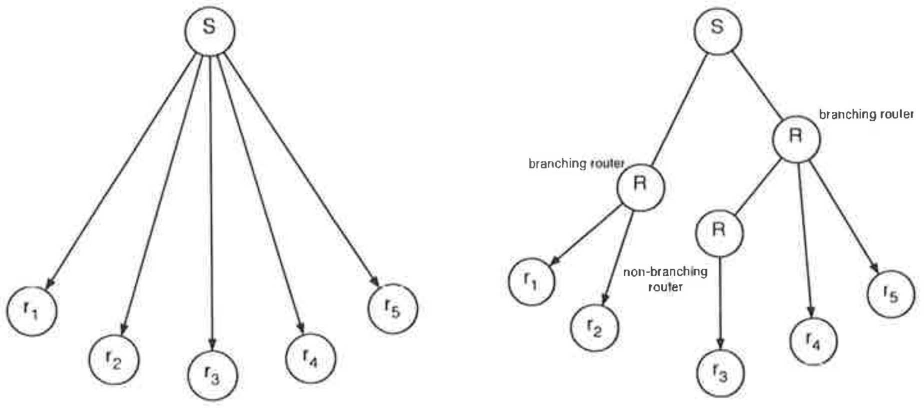
We develop the Lorikeet protocol from a set of requirements based on its target application and the properties of the current Internet. After describing the protocol's behaviour, we analyse its message complexity and its performance in terms of tree cost. We also analyse several other multicast protocols from the research literature, comparing their performance to that of Lorikeet in both complete deployment and incremental deployment scenarios.

Chapter 1

Introduction

"Multicasting" is the term used to describe group communication in which a sender or group of senders transmits information to multiple receivers. Multicasting allows senders to transmit a small number of copies of the information over a connected network which are then reproduced at intermediate hops in the network path and disseminated to all receivers in the group. This approach is significantly more efficient than the transmission of a separate, identical copy of the information for every receiver. Consequently, more receivers can be supported with the same resources. A pair of diagrams showing the topological difference between these two approaches for a single-sender scenario are given in Figure 1.

Figure 1 is a diagram of the *distribution tree* for a single source multicast transmission – this is a representation of the logical connections maintained between the source and the other nodes (receivers and routers). In the simultaneous unicast case of Figure 1(a), the source is sending five copies of the data stream through the network to five receivers, even if these streams may be travelling on the same network path for much of the transmission, a situation which is obviously duplication of effort. In comparison, the multicast tree in Figure 1(b) has the source sending only two copies of the data stream into the network, which are then replicated at routers elsewhere in the network and delivered to the same five receivers. Although the quantity of data received by the end receivers is identical, the amount of network



(a) Simultaneous Unicast: the source transmits a separate identical data stream to every receiver.

(b) Multicast: the source transmits a small number of data streams which are replicated and sent to receivers by a tree of intermediate routers in the network.

Figure 1: Topology: Simultaneous Unicast vs. Multicast

capacity required to serve those receivers using multicast is considerably reduced, particularly if there is a large amount of replication taking place.

In the context of passing data over the Internet, Internet Protocol (IP) Multicast has been defined by a selection of Internet Engineering Task Force (IETF) standards for more than fifteen years [20]. It is still not widely used for dissemination of multimedia content to home users [21, 25], despite the advantages of using multicast techniques for transmission of popular real-time content.

Figure 2 shows the potential benefits of using multicast over simultaneous unicast in terms of the cost of the complete distribution tree. The graph shows the total cost of the tree plotted against the event number, where an event represents either a receiver joining or leaving the multicast session. The tree cost is the sum of the costs of the individual links being utilised by the tree. The two approaches shown (Shortest-Path and Greedy algorithms) are conceptually simple multicast tree construction algorithms, and are described in Chapter 5. As the graph shows, both multicast approaches maintain *much* cheaper distribution costs than using simultaneous unicast.

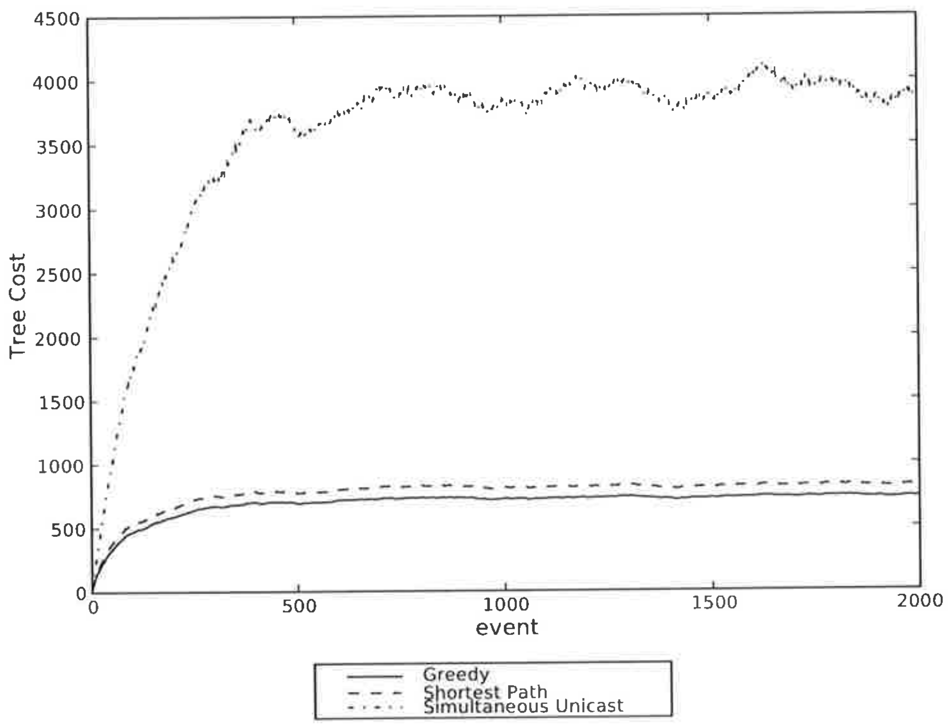


Figure 2: Comparison of Simultaneous Unicast and Multicast

These results are generated by the "Waxman" simulation described in Chapter 5.

The aim of this thesis is to extend the state of the art in multicast communications over the Internet by proposing a new protocol that overcomes some of the barriers to deployment of existing protocols, uses the existing network efficiently, and provides richer functionality to both receivers and senders.

1.1 Background

This section is a short introduction to many of the concepts underpinning multicast transmission in general, ideas that are common to most multicast systems. After this brief discussion, we present an overview of Lorikeet, our proposed multicast protocol.

1.1.1 Unicast Delivery

The majority of data travelling across the Internet today is transmitted using what are called *unicast* communications protocols. Unicast protocols send information as a stream of packets (small portions of data that can be reassembled after reception) between a single source and a single receiver. Since the Internet is a network consisting of many computers (referred to here as *hosts*) connected together, these unicast packets must traverse a path through the network, moving hop-by-hop from one host to the next, from the source to the destination.

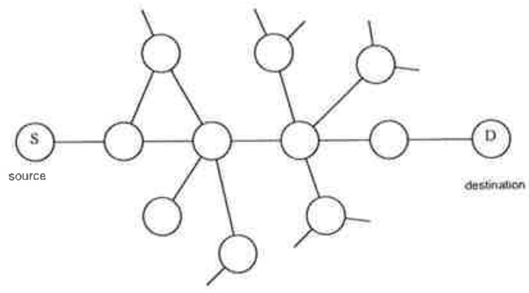


Figure 3: Unicast transmission over a packet-based network

Figure 3 illustrates this idea. Information is sent from the source S to the destination D through a series of intermediate nodes in the network that are connected together. This series of nodes is referred to as the *path* or *route* through the network for this (source, destination) pair. In an Internet network, each packet may travel via a different path, since each intermediate node selects the next hop in the path independently according to a routing protocol. This allows unicast transmissions to recover from changes to the network structure, such as node or link failures.

In a unicast connection, the two end-points (the source and destination) are often designated *end-hosts*. For the vast majority of Internet protocols, all the processing that occurs happens at these end-hosts, with the routers along the path between them simply forwarding packets.

Not all Internet nodes are connected to each other with the same type of network link: some nodes are connected via very high-capacity links, such as nodes in the core of the network through which large amounts of data are flowing, while others are connected by lower capacity links, such as home users connected via slower ADSL connections. In general, end-hosts tend to be connected to the Internet via limited capacity links, and this capacity is referred to as the *access bandwidth* of those end-hosts; that is, the maximum bandwidth that their access link is limited to. Even if they are connecting to another host through a path that is otherwise of very high capacity, the throughput between them will be limited by the access bandwidth of the slower host.

1.1.2 Multicast Delivery

Unlike unicast, *multicast* transmission describes the sending of information to a *group*: from a source or collection of sources to a group of receivers. This is done by having the sources send their data packets into the network, where they are replicated by hosts along the paths to their destinations, ensuring that all receivers in the group receive copies of them. Hence, the idea of a single path through the network is not sufficient: instead, multicast transmission can be visualised as a *tree*. If we consider the case of a single transmitting source S, sending information to a set of five receivers r_1 through r_5, the resultant multicast tree might look like Figure 4. In this diagram we can see that the source is supplying five receivers with data, but only sending two copies. When a packet reaches an intermediate host (which we call a *router*, marked R in the diagram) that is participating in the multicast, that host makes copies of the packet and sends one copy to each receiver that it is supporting. Routers with more than one direct child in the tree are referred to as *branching routers*, while routers with only single children are *non-branching*.

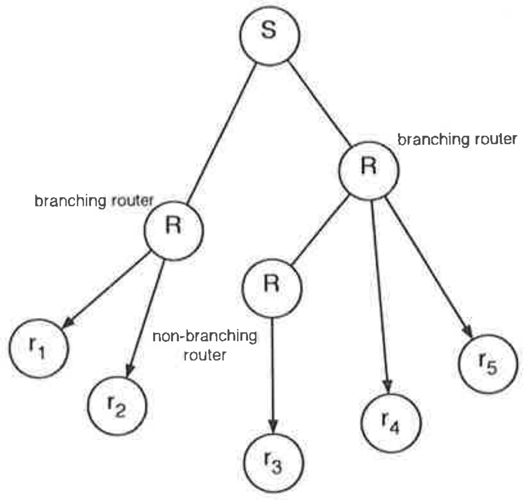


Figure 4: A Multicast Delivery Tree

These multicast trees are *logical* trees overlaid on the underlying network structure. For example, the path from R to r_1 in Figure 4 may actually consist of a path through several other routers in the network which are not participating in the multicast group and are simply forwarding packets.

1.1.3 Tree Construction

As described in the previous section, a multicast tree is a tree, rooted at the source or another "core" node, that connects the source to all of the receivers in the group. When a new receiver joins the tree, it connects to the tree along a path through the network: the way that a path is chosen to join this receiver to the tree may be characterised as either a *forward-path join* or a *reverse-path join.*

In a reverse-path system, the receiver initiates the join by contacting its nearest router, or a router on its path back to the source. That router connects to another on *its* path to the source, and so on until the new receiver is part of the multicast tree. We can see that the path through the tree to this receiver is therefore based on the reverse path: that is, the path from the receiver to the source. Most traditional multicast protocols use reverse-path tree construction, since it reduces the load on the source. The downside of reverse-path systems is that many networks have significant routing asymmetry: that is, the reverse path is often markedly different from the forward path between two nodes, as discussed by Paxson [59]. If a reverse-path join is being used for a multicast session in which all the data comes from the source, the delivery tree is being constructed in the opposite direction from that which is to be used for the actual delivery of data. This tree is likely to be less efficient for data delivery than one constructed in the forward direction.

A forward-path join is a join that bases a receiver's connection to the tree on the path in the forward direction, from the source to the receiver. This requires some collaboration from the source, but results in a multicast tree that is constructed in the same direction as the delivery of data.

1.1.4 Delivering Multicast Packets

Once constructed, the multicast tree is used to deliver packets of data from a source to the rest of the group. Each packet is transmitted by the source and forwarded to a multicast router, which forwards a copy of the packet out each of its interfaces that has known receivers attached, and so on. The direction of packet flow through the network can be characterised as upstream (towards the source) or downstream (towards the receivers), and multicast routers keep track of which interfaces are upstream and which are downstream interfaces. Two types of multicast distribution tree are generally used by multicast routing protocols: *shortest path trees* and *shared trees*.

In a *shortest path tree* approach, a multicast distribution tree is constructed using the model described above, with a source at the root of the tree and branches extending out towards receivers. Every multicast router in the group stores information about this source in its multicast forwarding table. In scenarios with more than one source, separate shortest-path trees must be constructed for each source and an entry stored for each tree in every participating multicast router. This approach is sometimes also called a *source-based* approach.

Unlike shortest path trees, *shared trees* use a single common node called the *rendezvous point* (RP), sometimes also called a *core node*, as the root node of the distribution tree. All sources send their traffic to the RP, which forwards it down the shared tree to all of the receivers. This reduces the state information required in routers, since only the RP needs to know the locations of all the sources, and only one multicast tree (rooted at the RP) needs to be constructed. However, since all data must be sent by the sources to the RP and then redistributed, this approach will almost never be as efficient in terms of total tree cost as the use of shortest-path trees. The only way for a shared tree to be as efficient is for the RP to lie on the shortest paths between every source and receiver in the multicast group. Hence, the selection of the RP's location in the network is critical to minimising the cost of a shared tree multicast group.

1.1.5 Dynamic Trees

Many multicast protocols are designed for static scenarios, where the set of receivers is known and will not change over the course of the transmission. One example is a corporate videoconference. In such a case the set of receivers is known ahead of time, so the delivery trees are constructed, the conference takes place, and then the trees are torn down. The receiver set is fixed for the duration of the session, and the trees could even be statically constructed ahead of time.

Many multicast applications, however, are more dynamic systems: the set of users can change over the course of the transmission, causing new paths through the tree and new branching points to become available. This can reduce the efficiency of the distribution tree, particularly in multicast protocols where the selection of a branching point for a new receiver is dependent on the paths chosen through the network by receivers that joined the tree earlier on.

As an example, consider the case of an "Internet TV" station, where a content provider is multicasting content 24 hours a day, 7 days a week – a news channel or a music video channel, for example. In such a case, the composition of the receiver set can change markedly over time for a number of reasons:

- Clients do not all watch the stream 24 hours a day. The vast majority of end users will join the session, watch for a while, then leave the session. This is referred to as *dynamic membership*.
- If the audience is worldwide, the locations of the majority of receivers will shift depending on the time of day (for example, as an audience in one country goes to sleep and an audience in another wakes up), creating very different tree topologies.
- Failures may occur in the network, removing individual receivers or entire branches from the tree.

In these scenarios with dynamic membership, it is possible in some multicast systems to perform maintenance of the tree, often called *tree rearrangement*. In these systems, branching points or receivers (or both) are re-connected to the tree when appropriate in order to prune older, less efficient branches in the tree or make use of new branches created by the addition of recent receivers. In some systems, this rearrangement is triggered by participants randomly probing other points in the tree to find better locations to connect. In others, rearrangement of the tree can be triggered by an event counter threshold or a periodic timer. This maintenance of the tree helps maintain efficiency in the face of changes to the tree's structure caused by the addition and departure of receivers.

1.2 Lorikeet

We propose a new multicast protocol called Lorikeet. Lorikeet is designed to be a practical, incrementally-deployable single-source multicast scheme that will operate on the current Internet without requiring IP multicast to be deployed. The target application for Lorikeet is single-source live audio and video distribution. In addition, we use routers in the network to perform branching, enabling more efficient use of the network and better performance than purely application-level protocols. Lorikeet is capable of rearranging its distribution tree to cope with changing receiver populations, and maintains minimal state information at individual nodes in the tree. As is shown in Chapter 5, Lorikeet builds distribution trees with less than half the cost of simultaneous unicast distribution when only 10% of the routers in the network support the protocol.

Lorikeet uses a source-based joining procedure and a hop-by-hop delivery mechanism similar to the recursive unicast tree approach proposed in REUNITE [71] and extended in HBH [18]. Unlike most other multicast protocols, including REUNITE and HBH, Lorikeet employs a true forward-path tree construction technique, always joining new receivers to the multicast tree via a unicast connection from an existing router in the tree. This ensures that the tree is optimised for data delivery, which occurs in the forward-path direction from source to receivers. In the join procedure, a new receiver contacts the source and the source begins a recursive search of the current tree to find a close parent node for the receiver, or connects the receiver directly if a close enough parent cannot be found. Routers in the network that support the Lorikeet protocol are termed *capable routers* and are able to provide branching points in a multicast tree. Receivers in Lorikeet are always leaf nodes in the tree and therefore support no children. This design decision was made in order to simplify leaving the tree, focus on branching in the core of the network where capacity is not limited to access bandwidth and facilitate tree rearrangement. Lorikeet's join operation is of low complexity compared to the joins proposed by other multicast approaches which share some of Lorikeet's features, such as ARIES [8] and DSG [32], while still generating trees of comparable cost.

Dynamic membership is supported by allowing the distribution tree to locally rearrange capable routers. This rearrangement is triggered by a topological event, rather than a timer or threshold, in order to reduce the likelihood of unnecessary modifications to the tree.

Thus, Lorikeet has a number of advantages over traditional multicast and many other proposed multicast protocols:

- it is designed specifically for single source transmissions to end receivers, removing the communications overhead required to perform sender discovery or manage multiple trees.
- it provides join and leave mechanisms that have low message complexity, enabling receivers to join and leave the tree rapidly.
- it is incrementally deployable: Lorikeet will operate even with no capable routers in the tree – receivers will simply join directly to the source, in a "simultaneous unicast" configuration.
- it has no dependency on IP multicast or special multicast addressing, using standard IP unicast addresses for source identification. It is still, however, able to use routers in the network to perform branching.
- it is able to make efficient use of the network: Lorikeet's trees are forward-path trees constructed to make efficient use of the underlying network topology.
- it is able to effectively rearrange the tree to cope with changes to the receiver set. In particular, these rearrangements are localised to the subtree under a single router and no global calculations are required to achieve them.

Many of these features are also present in the REUNITE [71] protocol described by Stoica *et al.* There are some significant differences between the work done on REUNITE and Lorikeet, however:

- REUNITE claims to construct the tree in the forward-path direction but clearly does not, instead using the reverse unicast shortest path from the receiver to the source to select parent routers in the tree. Lorikeet therefore creates more efficient trees in scenarios with asymmetric unicast paths. This issue is also addressed by the HBH protocol, which modifies REUNITE's join procedure to better cope with asymmetric paths.

- Lorikeet explicitly searches the current tree for parent routers when connecting a new receiver, while REUNITE only searches the reverse unicast path between the new receiver and the source. Hence, Lorikeet is able to more effectively leverage the routers already in the tree, providing more branching and a lower cost delivery tree.
- REUNITE uses soft state with periodic refreshment to maintain all forwarding information in the tree, while Lorikeet manages joins and leaves with explicit control messages, requiring much less control communication to maintain the structure of the tree.

With the exception of the changes to its join technique that improve performance in trees that traverse asymmetric paths, HBH shares these properties with REUNITE. A discussion of both protocols and a comparison between them and Lorikeet are presented in Section 5.2.4. Simulation data comparing REUNITE/HBH and Lorikeet is presented in Chapter 5, in which we show that Lorikeet is able to outperform REUNITE/HBH by over 30% in terms of total tree cost in scenarios with limited protocol deployment in the network. Even at higher levels of deployment, Lorikeet consistently generates cheaper trees.

1.3 Thesis Structure

In Chapter 2, we present a survey of other work on multicast. We begin by examining the current state of Internet Protocol (IP) multicast, as standardised by the Internet Engineering Task Force. We then discuss different approaches to multicast protocol design from the literature, including small-group multicast, application-level and overlay multicast, and protocols that leverage information about the underlying network topology.

Chapter 3 is a discussion of the Steiner Tree Problem in Networks, which is the graph-theoretic problem that underlies multicast tree construction. We investigate both exact algorithms for finding optimal solutions and heuristics for finding approximate solutions. An analysis of the performance of these heuristics is presented.

In Chapter 4, we describe Lorikeet, our new multicast protocol for single-source multimedia streaming. We develop the ideas behind Lorikeet from the characteristics of its target application and the network environment, then present a description of each operation performed by the protocol.

The performance of Lorikeet and several other multicast protocols is analysed in Chapter 5. Here, we describe our simulation environment and analyse Lorikeet's complexity and the behaviour of several different operations described in the previous chapter. Lorikeet's performance is compared to that of several other competitor protocols, including ARIES, DSG and REUNITE. Finally, we analyse Lorikeet's behaviour in an incremental deployment scenario and compare it directly with REUNITE (which is also capable of incremental deployment.)

In Chapter 6, we present an investigation into the use of directory nodes, a service designed to enhance Lorikeet's ability to discover Lorikeet routers when they are sparsely distributed throughout the network. We extend Lorikeet's join algorithm to use this service, and present some results on its efficacy.

Chapter 7 is a discussion of issues related to the development of a physical implementation of Lorikeet. In this chapter, we describe issues that are not immediately obvious from Chapter 4's description of Lorikeet's topological behaviour, such as security concerns, resource management and the handling of multiple simultaneous operations. We also describe the different ways in which Lorikeet can be deployed and the likely drivers for this deployment.

Chapter 8 presents a summary of our findings in this thesis, and discusses several possible avenues for future work on Lorikeet and research in multicast in general.

1.4 Major Research Contributions

This section briefly outlines the major research contributions made by this thesis.

- We have developed and specified a new multicast protocol called Lorikeet, designed for delivery of real-time content from a single source to a large set of receivers on current Internet networks.
- We have investigated the Steiner Tree Problem in Networks and developed Lorikeet's join and tree maintenance algorithms from an understanding of the underlying topological problem.
- Lorikeet is designed to be deployed incrementally and allow receivers to access content even in the absence of network routers that support the protocol. Lorikeet's performance has been analysed at a range of network penetrations ranging from zero to all routers in the network, and the level of deployment required to make significant performance improvements has been identified.
- We have developed a rearrangement operation for Lorikeet's distribution tree that is triggered by a topological event (a router becoming non-branching) rather than forcing a rearrangement according to a periodic timer or a threshold being reached. This rearrangement operation is localised to a single branch of the tree and does not require any global calculations.
- We have analysed the complexity of Lorikeet and several other algorithms for joining, leaving and rearranging the tree in terms of the control messaging required, in both static (Steiner heuristic) and dynamic (online multicast) scenarios.
- The performance of Lorikeet in a number of scenarios constructed to approximate receivers joining and leaving a tree on a number of Internet-like topologies has been simulated and compared to several other protocols, including REUNITE [71], ARIES [8] and DSG [32].

- We have examined the ways in which Lorikeet could be deployed in the network, either in the core or out near the edges, in ISP access networks. For the latter case, we have investigated the use of directory nodes as a capable router discovery service to improve performance when capable routers are concentrated at the edge of the network, as with Web proxy servers.

- We have examined the barriers to widespread implementation of IP multicast and other multicast schemes and described a plan for implementation of Lorikeet that would overcome these obstacles.

- We have suggested several ways in which Lorikeet can be extended to provide features that are difficult or not possible to implement with other multicast protocols, including
  - adaptation of the data stream as it flows down the tree to deal with receivers' different capacity requirements; and
  - local retransmission (between a parent router and child router or receiver) of packets when they are lost but can be retransmitted before they are due for playback.

This chapter has introduced the topic of multicast transmission over the Internet and presented a brief discussion of some concepts that are basic to most multicast protocols. We have also presented a summary of the major features of our protocol for multicast transmission, called Lorikeet, and outlined the major research contributions of our work. The following chapter will present a survey of existing research in the field and current Internet standards, and later chapters will describe Lorikeet in further detail.

Chapter 2

State of the Art

This chapter presents an overview of the research literature and the existing standards for Internet multicast communication. First, we describe the existing IP multicast protocols and their uses, the majority of which are standardised by the Internet Engineering Task Force (IETF) and have implementations available for use. Next, we present a description of Source-Specific Multicast (SSM), a simplified set of these protocols designed for single-source multicast only, and briefly give an account of the state of IP multicast deployment and the barriers preventing its use by many end users.

The remainder of this chapter we devote to alternative approaches to multicast that have appeared in the literature. First, we go through the small group multicast approaches, which encode the destination lists in the packets themselves and deliver them via unicast with router support. Second, we treat application-level and overlay network approaches. Application-level multicast schemes perform all distribution of data on the end-hosts themselves, building the distribution tree solely out of end-hosts and requiring only unicast transmission between member nodes. Overlay network approaches take this idea one step further, building a generalised overlay of participating nodes on top of an IP network (often introducing different addressing and routing schemes), then implementing multicast as a service on this overlay network framework. Finally, we examine a group of multicast protocols that explicitly leverage information about the underlying network topology to build efficient distribution trees.

This survey of different protocols is not intended to represent a complete, well-defined taxonomy of multicast techniques. We have grouped them into subsections for ease of comparison and brevity in description, rather than attempting to explicitly categorise each system.

2.1 Internet Protocol (IP) Multicast

The IETF maintains a set of standards that we group under the title "Internet Multicast", defining group communication over Internet Protocol (IP). Internet Multicast was initially defined in RFC 966 [20], which introduced the concept of "host groups", sets of hosts identified by a single IP address to which packets could be delivered. Further development led to RFC 1112 [19] in 1989, which defined version 1 of the Internet Group Management Protocol (IGMP) and became the recommended standard for Internet multicast transmission. These initial RFCs define a basic mechanism for group communication, based around the use of special-purpose multicast addresses. Every multicast group selects a single multicast address from the pool of Class D IP addresses, 224.0.0.0 through 239.255.255.255 in IPv4. Packets sent to a group's multicast address are delivered by the network to every host in the group. Many-to-many communication is supported; that is, a group can have multiple senders as well as multiple receivers. Joining a group is achieved by announcing a desire to receive packets on that group's multicast address (for example, by binding the host's ethernet interface to the multicast address and notifying the local multicast router). Local routers use IGMP messages to discover when they have members of multicast groups on their attached local networks, and use that information to selectively forward multicast packets via only those interfaces. Sharing of the multicast state information gathered by IGMP between routers in an organisation is achieved through the use of *multicast routing algorithms*, which are analogous to the routing algorithms employed for unicast. The current version of IGMP is IGMP version 3 [13]. In IPv6 networks, a newer protocol called Multicast Listener Discovery (MLD) [75] serves the same function as IGMP does in IPv4 networks.

As described in Chapter 1, Internet multicast protocols can be characterised as shortest-path and shared tree protocols. In a shortest-path protocol, the delivery tree is constructed with the source at the root of the tree. IP Multicast routers store information about this source as an (S, G) pair in their multicast forwarding tables, where S is the unicast address of the source and G is the multicast address of the group. In multicast groups with more than one source, routers must build a separate (S, G) shortest path tree for each one. In a shared tree protocol, the tree is rooted at a rendezvous point (RP). All sources send their traffic to the RP, which forwards it down the shared tree to all of the receivers. Shared tree multicast groups are commonly described with (*, G) notation, with the wildcard "*" representing the sources and G as the multicast group address.

A number of different multicast routing protocols have been developed over the years, beginning with the Distance Vector Multicast Routing Protocol (DVMRP) [76]. As its name suggests, DVMRP is a distance-vector routing protocol, similar to the Routing Information Protocol (RIP) [36] developed for unicast routing. Distance-vector routing protocols build routing tables for their participants in a distributed manner, based on the sharing of information about reachable hosts and their network distances between neighbours. The approach to multicast routing and delivery used by DVMRP is a "dense mode" approach, based on a flood-and-prune mechanism: when a multicast router receives a packet, it sends that packet out on all of its interfaces except the one it came in on (the upstream interface). If a router has no receivers in its connected networks for a packet it has received, it sends a "prune" message to the upstream router, requesting that it not be sent subsequent packets for that group. Periodic re-flooding is used to refresh state (in case new interested receivers appear on previously-pruned networks). Upstream and downstream paths are determined through Reverse Path Forwarding (RPF) checks against multicast route information maintained by routers. Routers running DVMRP maintain this information by exchanging distance vector updates with neighbouring routers.

A later protocol, Protocol Independent Multicast – Dense Mode (PIM-DM) [1] implements the same flood-and-prune approach, but does not build and maintain its own routing table for RPF checks. Instead, it is able to use the routing table provided by any underlying unicast routing mechanism. Both DVMRP and PIM-DM are shortest path tree approaches to multicast tree construction.

Dense protocols, while very simple and useful for group transmission where receivers are common (that is, where there is likely to be at least one receiver in every connected network), are very inefficient in cases where receivers are much more sparsely distributed. To serve this scenario better, sparse protocols like Protocol Independent Multicast – Sparse Mode (PIM-SM) [26] were developed. PIM-SM uses a shared tree approach, with a rendezvous point (RP) that keeps track of the locations of all the sources. Instead of flooding the network, sources send their data to the RP, which handles distribution to the group's receivers. The benefit of this approach when compared to PIM-DM is that no flooding is required to notify routers of the locations of active sources; instead, the RP is the only node that needs to know the sources' locations. Multicast routers in the group need only store one shared tree rooted at the RP, rather than storing separate shortest path trees for each source. The disadvantage of using a shared tree protocol is the inefficiency introduced by using a single tree rooted at the RP – if the RP is not optimally located (given the locations of the sources and the receivers), traffic will flow along longer paths through the RP than it would if it was travelling from the sources to the receivers directly. An additional protocol called Multicast Source Discovery Protocol [27] (MSDP) can be used to connect several PIM-SM distribution trees together, with each domain using its own independent RP.

Source-Specific Multicast (SSM)

Many of the issues described above (such as source discovery and the use of an RP) only apply to multicast sessions where there can be more than one source transmitting data to the group. In the case of single-source groups, much of the extra complexity associated with these standards is unnecessary. The Internet community has recognised this and developed a standard called Source-Specific Multicast (SSM) [9] for multicast groups with only a single source. Consequently, the original standards for many-to-many multicast communication have come to be grouped under the term *Any-Source Multicast* (ASM).

SSM represents a subset of the existing Internet multicast standards, simplified for the one-to-many model. It shifts the problem of source discovery to the application layer, and represents a multicast group with a single (S, G) tree: when a new receiver wishes to join a group, it informs its local multicast router of the unicast address of its source S, as well as the group address G. Routing of multicast packets under SSM is done with a subset of PIM-SM and IGMPv3, without the need for separate multicast routing tables (as in DVMRP), RPs or sender discovery. However, SSM still uses the basic IP multicast delivery model and Class D group addressing. SSM is derived from earlier work on EXPRESS [38], which originally defined the single source multicast channel represented by an (S, G) pair.

Deployment on the Internet

All of the protocols described so far constitute "interior" multicast routing protocols, designed to be used inside a single autonomous system (AS); that is, a single organisation. Additional protocols are used to share information between ASes, permitting multicast sessions to extend across larger areas of the network. The most common protocol used for this purpose is the Multiprotocol Border Gateway Protocol (MBGP) [6], an extension to the standard BGP protocol used for interdomain routing. MBGP allows BGP to share information about routes other than unicast

IP routes (in this case, multicast RPF information) between different organisations.

These "traditional" IP multicast protocols all have a number of properties in common that make them difficult to deploy on an Internet-wide scale: all the routers between the source and the receivers need to have multicast enabled; some protocols require additional memory for routing tables in multicast routers; Class D group addresses need to be assigned; and many organisations are not willing to bear the additional network and CPU load associated with multicast traffic.

For these reasons, IP multicast is not widely deployed across the Internet today. It is in use inside organisations for "local" applications (such as video-conferencing), but is not generally available to end users connected to the Internet via commercial Internet Service Providers. In the early 1990s, a network of multicast-enabled routers called the MBone was created, connecting together via unicast tunnels which allowed these "islands" of multicast connectivity to appear seamlessly connected. The MBone was used for multicast protocol research and a small number of applications, such as audio and video multicasting of IETF meetings. In 2006, many larger organisations have portions of their networks running native multicast, even across AS boundaries, but it is still not available to end users. Today, Internet 2 (a research network connecting a large number of predominantly American universities) is used as a multicast research platform.

2.2 Small Group Multicast

IP multicast is not a particularly "lightweight" suite of protocols, presenting the network user with a large amount of infrastructure that is required for group communication to occur. Several potential multicast applications, however, involve small groups of end receivers for which all this infrastructure can be considered unnecessary or over-engineered. For these applications, an area of research that has attracted some attention is *small group multicast*, also termed *explicit multicast*. These systems are designed for multicast transmission of data between small groups of hosts, facilitated by the carrying of lists of destinations in the data packets themselves. Examples of small group multicast protocols include Xcast [11], SEM [12] and LinkCast [3].

Xcast explicitly encodes the list of destinations in the data packets, rather than using a multicast group address or storing state information at member nodes. This list is provided by the multicast source, which must maintain a list of all receivers in the session. When a router receives such a packet it reads the header, partitions the destinations according to their next hops, and forwards a copy of the packet with appropriately rewritten headers to each next hop. This approach requires no additional control communications in the network and requires no state information to be stored at routers. However, having each packet contain a full list of receivers severely limits the size of the multicast group, making explicit multicast protocols inappropriate for large-scale multimedia transmission. Xcast+ [68] extends Xcast by adding an IGMP join at the receiver side to the system and using a list of multicast routers, rather than receiver addresses, in the packets. Distribution of the data packets to end-receivers is then handled by locally-scoped IP multicast delivery from these multicast routers. Similarly, SEM [12] uses IGMP to manage joins and has the source maintain only a list of participating multicast routers, with which it constructs a tree (storing some limited forwarding state in intermediate routers). Both these approaches still require that traditional multicast routing protocols be deployed in the network, though they do succeed in reducing the state information and control overhead required. LinkCast [3] improves upon XCast's storing of receiver lists in packet headers by encoding instead a set of link indices, generated from the receiver join messages which are appended to by the routers which they traverse. Though this approach can support a larger number of receivers than Xcast, it still does not scale to large groups and requires deployment on every router in the network in order for link index gathering to operate.

2.3 Application-Level Multicast and Overlay Networks

Even explicit multicast techniques have the requirement that routers support the protocol in order for them to operate: a router that does not support the protocol cannot replicate and forward the packets correctly. To overcome this obstacle, there has been considerable research in "application-level" or "end-host" multicast. In these systems, all processing is done at the edge of the network on the end-hosts: every receiver participates in the tree as a branching node, forwarding packets to other receivers further down the tree. Since all the branching is done at end-hosts, there is no necessity for routers to support the protocol. Hence, simple unicast forwarding is used for transmission from the source to receivers, and between receivers themselves. Many of these approaches also implement *rearrangement* of the distribution tree, to cope efficiently with changes to the receiver population – if a new receiver joins the tree, other receivers may be able to improve their performance by becoming children of this new receiver. When a receiver leaves the tree, its children must be re-parented to maintain their connection to the group.

Helder and Jamin's Banana Tree Protocol [37] is an application-level multicast protocol designed to be the underlying control mechanism for a peer-to-peer file sharing application. It has a very simple approach to joining the tree: nodes simply join the multicast distribution tree as children of the source of the tree (it is assumed there is a bootstrapping mechanism by which they can discover the location of the source). Optimisation of the tree is then taken care of through a periodic rearrangement procedure: a node periodically tests its siblings for closeness, and switches its parent to one of these siblings if it is closer than its current parent. Helder and Jamin found that this approach worked well in ideal situations, but did not perform adequately in more realistic scenarios: their conclusion was that a wider range of graph transformations was required to effectively optimise the distribution tree.

Chu *et al.*'s Narada protocol [40] creates a well-connected mesh of hosts and then constructs spanning trees from this mesh, rooted at each source of the transmission. Rearrangement in Narada is based on members randomly probing each other and adding new links based on a given utility function. Pendarakis *et al.*'s ALMI [60] uses a centralised session controller (like an RP) to coordinate the multicast tree – this session controller constructs the tree by periodically calculating a minimum spanning tree based on measurements from tree members. Similarly, El-Sayed and Roca's HBM protocol [65, 24] uses a centralised RP that builds the multicast distribution tree. This RP has complete information about all receivers in the group and is responsible for calculating the tree's topology and disseminating that information to all its participants. Data transmission between receivers happens directly, however, rather than through the RP as in traditional shared tree protocols. An analysis of the performance of the application-level approach has been presented by Chu *et al.* [39, 40]

In [51], Mathy *et al.* present an application-level tree construction protocol called TBCP (Tree Building Control Protocol). TBCP is designed to be an efficient tree construction mechanism that operates with partial knowledge of the multicast group membership and limited network topology information. It uses a tree joining algorithm that finds a parent node for a new receiver by recursively searching the existing tree (and, if necessary, re-parenting existing nodes to maintain a bound on the number of children a tree node can support). TBCP constructs a tree with very limited network information and no dependencies on IP multicast or special-purpose routers in the network. However, it only specifies a tree construction mechanism, designed to be combined with the delivery, tree maintenance and leaving operations provided by another overlay or application-level protocol.

The NICE application-level multicast protocol [5, 4] is designed for relatively large receiver set, low-bandwidth real-time applications that can tolerate some loss, such as news and stock tickers. NICE organises its member nodes into a hierarchical control topology based on the separation of groups of nodes into layers, further partitioned into clusters of nodes that are close to each other in network terms. All of the nodes in the group are in layer 0 and organised into clusters. Each cluster on layer 0 (zero) has a cluster leader that becomes a member of a cluster at layer 1 (along with other layer 0 leaders), and so on. Nodes only maintain state information about other nodes in their clusters, thereby limiting the amount of space required on each node. Data transmission is performed on a tree derived from this control hierarchy, where nodes retransmit received packets to all members of clusters for which they are the leader. NICE periodically performs maintenance on this control hierarchy; this maintenance consists of split and merge operations designed to maintain cluster size between two bounds. In addition, a node periodically probes the leaders of other clusters in a given layer looking for a more appropriate cluster allocation. Several other proposals (SAHC [52] and ZIGZAG [74], for example) present similar schemes based on dividing the receiver set into bounded-size clusters from which a multicast distribution tree can be constructed.

Several research projects have developed the idea of building *overlay networks* where participant nodes are organised in a structured manner and nodes can be addressed using, for example, their locations in a coordinate system. Much of this research is in the related area of peer-to-peer networking, but several projects have implemented application-level multicast as an application on a framework that provides an underlying overlay structure. The Narada system, described earlier, can be considered an overlay network approach, since it organises its hosts into a mesh before building distribution trees on it with a routing protocol. Ratnasamy *et al.* describe such a system built upon an overlay network framework called the Content-Addressable Network (CAN) [63], called CAN-based multicast [64]. Similarly, Zhuang *et al.* present Bayeux [83], a multicast system that leverages an underlying overlay framework called Tapestry [82]. In [14], Castro *et al.* present an evaluation of application-level multicast schemes built on several of these overlay network frameworks, namely CAN, Chord [70], Pastry [66] and Tapestry.

Chawathe's ScatterCast [15] protocol is an overlay multicast distribution system that combines concepts from overlay network protocols, application-level systems and traditional IP multicast. It builds an overlay network of "ScatterCast proxies" (special-purpose, strategically located servers) and receivers, using locally-scoped IP multicast for delivery where possible, and direct unicast connections where IP multicast is not available. The overlay network of proxies is built by creating a strongly connected mesh first, then running a standard routing protocol on top of the mesh to build shortest path trees (rooted at sources) for data distribution. Similarly, Yoid [28] and HMTP [81] are both application-level multicast proposals that leverage existing IP multicast capability where possible, and construct overlay networks over unicast elsewhere. HMTP builds a shared tree of members representing each "island" of local multicast connectivity (called Designated Members, or DMs) and assumes all end receivers are multicast-capable hosts within the same network as a DM. In addition to the shared distribution tree, Yoid also creates a mesh topology for control information and increased robustness.

Application-level multicast schemes solve the problem of requiring deployment of the protocol across the whole network, since they only require unicast transport between group members (which are all end-hosts). However, they do so at a performance cost: all branching of the distribution tree happens at the edges rather than in the core of the network. If branching in the core were possible, the system could reduce path lengths significantly and hence build a more efficient tree.

Another issue with traditional multicast protocols that is addressed by many application-level approaches is that of dynamic membership. IP multicast was largely designed for well-behaved applications with a static list of hosts in the group, where the list of receivers is fixed and does not change over the lifetime of the session. The delivery tree is therefore very stable, and no further maintenance is required after its construction. Many multicast applications, however, can invite potentially very dynamic membership, with sets of users connecting and disconnecting over the period of the transmission. Several application-level protocols try to maintain efficiency during these changes by periodically rearranging the distribution tree. Often

([37, 5, 42, 40]) this is achieved by having receivers periodically attempt to rejoin the tree at the source or a nearby node, searching for a better parent in the tree than their current parent.

2.4 Topology-Aware Multicast

Most overlay and application-layer multicast systems are built to operate at the edge of the network on end-hosts, and their tree construction and maintenance algorithms emphasise connectivity over performance. In this section we present an examination of research into multicast systems designed to more closely match the structure of the underlying network, generating higher performance multicast trees and even rearranging the tree when the current tree is inefficient due to changes in the network or the receiver population. We have chosen to describe these systems in greater detail since this goal of building and maintaining efficient trees is one shared by our work on Lorikeet.

In ARIES (A Rearrangeable Inexpensive Edge-based on-line Steiner algorithm) [8, 7], Bauer *et al.* develop a multicast protocol designed to cope with dynamic membership by rearranging the distribution tree. In an ARIES tree, regions of the tree maintain "damage counters" which are incremented when nodes are added or removed from them. When the counter for a region exceeds a set threshold, that local region is rearranged using a Steiner heuristic[1] that only modifies links in that region, and its counter is reset to zero.

In several respects, ARIES is impractical for use on the current Internet, but a valuable piece of research in multicast tree construction and maintenance. The joining method used is a greedy join, which has a new node join the tree by connecting to the nearest tree node to it in the network – this is an ideal operation that would be much too expensive to perform on a current network, due to the necessity of

[1]Steiner heuristics are algorithms designed to find good approximate solutions to the Steiner Tree Problem in Networks, described in detail in Chapter 3.

measuring the cost of connecting via every node in the tree. ARIES' rearrangement algorithm also adds significant complexity and communications overhead to its protocol. Each region's counters must be kept up to date by having newly-modified nodes broadcast counter updates to their surrounding regions. ARIES' authors suggest the use of the Kruskal Shortest Path Heuristic (K-SPH) as the Steiner heuristic to use for rearrangement: K-SPH operates by deleting all the links in a region, then treating the individual nodes as 'fragments' to be re-connected, starting with the two fragments that are closest together. It continues in this way until only one fragment remains; that is, all the nodes in the region have been connected again. This operation requires a great deal of topological knowledge and measurement, even though it is restricted to operating within a single region. Furthermore, we found that when ARIES was implemented in our simulation environment, the shortest paths between two fragments being connected by K-SPH could occasionally travel through a third node currently in the tree, which would result in a loop. The only way to avoid this result is by treating the rest of the tree outside the region as an additional fragment, extending the rearrangement of the tree from a local region-based calculation to one that involves the complete tree.

Goel and Munagala's Delay-Sensitive Greedy (DSG) algorithm [32] also uses a greedy join. Every node that joins the tree measures its *stretch*, the ratio of the delay via its path through the tree to the delay on the shortest path from the source. If the stretch for that node exceeds a set threshold, that node (and all of its parent nodes that fail a second, tighter bound) is re-parented directly to the source, thereby minimising its delay. This approach is designed to construct efficient trees (through use of the greedy join), while satisfying a second constraint on the relative delay: in a DSG tree, all receivers can be guaranteed a minimum delay relative to their optimal shortest path delay. The stretch threshold check occurs during the join operation, ensuring that the delay constraint is always met for all receivers; DSG does not specify a leave operation, focusing solely on the tree construction problem. As for ARIES, the use of a greedy join in DSG makes it impractical for large-scale deployment on the Internet. It does provide, however, an example of a tree construction and maintenance system that balances several conflicting resource requirements: in this case, total tree cost and the delay to individual receivers.

Stoica *et al.* have proposed a protocol called REUNITE [71] (an acronym representing "REcursive UNicast TrEe") based on hop-by-hop unicast transmission of data between routers that support the protocol. This approach solves a number of the problems slowing widespread use of IP multicast: it is incrementally deployable on the network; it stores multicast forwarding state only on routers acting as branch points (rather than all routers through which the multicast transmission passes); and it has no requirement for separate multicast addresses, since the group is a single-source group identified by the source's unicast address. REUNITE uses a passive tree construction method, relying on shared portions of unicast shortest paths from the source to achieve branching (and therefore, more efficient network usage). The tree is maintained by a constant exchange of messages between the source and the receivers; the source sends TREE messages downstream, and the receivers send JOIN messages upstream (though these are discarded when they reach the first multicast router in that direction). Leaving the tree is accomplished by simply stopping the transmission of JOIN messages, which results in a timeout at the parent router and the tearing down of that branch of the distribution tree. This approach is called a *soft-state* approach, since state information in routers will "disappear" after a timeout if it is not maintained through the periodic exchange of messages. While simple to describe, this approach does require the constant exchange of messages simply to maintain state information (even if that state is unchanging). Participating routers must also maintain timers on all information stored in order to timeout stale information.

The Hop-by-Hop (HBH) [18] protocol extends REUNITE's delivery mechanism by proposing its combination with the single-source channel abstraction from EXPRESS [38], using class D multicast IP addresses to refer to its multicast groups. It also refines the tree construction algorithm, focusing on the construction of forward path trees and dealing with some pathological cases that arise in asymmetric networks that REUNITE does not treat as efficiently. Most of the properties described above for REUNITE also hold true for HBH, and the two protocols build identical multicast distribution trees in symmetric networks. REUNITE and HBH are discussed further in Section 5.2.4.

Jannotti *et al.* have developed a single-source, application-level multicast protocol called *Overcast* [42], aimed at large-scale delivery of video content. Overcast is designed to work with customised Overcast routers forming an overlay network, and unmodified HTTP clients as end-receivers. Tree construction is designed to place nodes in the tree as far downstream from the source in the tree as possible while not sacrificing bandwidth from the source, measured on-line by transferring small quantities of data. This approach is designed to ensure that as much branching as possible takes place in the tree. Nodes periodically re-evaluate their position in the tree and relocate themselves if appropriate.

In [46], Kwon and Fahmy present a new application-level multicast approach, called Topology Aware Grouping (TAG). TAG constructs its (single-source) multicast tree by exploiting information about the underlying network – namely, using the overlap among underlying unicast paths from the source to the group members to construct the tree. A new node joining the tree will become a child of the current tree node that shares the longest overlapping unicast shortest path from the source through the network. The resultant distribution tree is a forward-path tree designed to match the underlying network topology quite well, assuming that the underlying unicast paths are of high quality. TAG is, however, an application-layer approach and thus cannot make use of routers in the core of the network. In addition, a mechanism for determining the underlying unicast paths between the source and all participating nodes is necessary – the approaches used in their implementation are the use of `traceroute` and from publicly-accessible topology sources (such as OSPF topology servers, monitors or Internet topology discovery projects). These sources of route data are often incomplete, out of date or very coarse-grained, compromising the efficiency with which TAG can construct its distribution tree. In addition, underlying unicast routes may be more a reflection of routing policy than the optimal shortest paths that would provide the most efficient paths from the source to receivers.

Research in multicast protocol design is clearly a large field, with a multitude of different protocols expressing different application and network requirements. IP multicast was designed initially to be a generic solution, offering a platform for many-to-many group communication with quite heavy network requirements, such as the necessity of complete deployment and special-purpose addressing. These requirements, and other issues, led to a lack of widespread deployment of "native" IP multicast and research into other approaches to solve these issues. Small group multicast systems require no state information at routers but have significant limitations on group size and throughput. Application-level and overlay multicast protocols address the problem of requiring complete deployment across the network by doing all processing on end-hosts, but cannot match the performance of systems that are able to do branching in the core of the network. Other protocols address the issues of dynamic changes to the tree and directly exploiting the underlying network topology.

Our multicast protocol, Lorikeet, is designed to address a single application, that of the delivery of "live" streaming data (such as audio or video) across the Internet to large groups. From this application arises a number of basic requirements, which are described in Chapter 4. From the many different systems described above, however, it is obvious that many different ways of constructing a delivery tree are possible: the join operation is critical to the efficiency of a multicast protocol. For this reason, we examine the underlying problem of constructing a tree containing a subset of the nodes in a network, the *Steiner Tree Problem in Networks.* The following chapter presents a description of the problem and an analysis of several approaches for solving it, as a basis for designing a multicast join operation.

Chapter 3

The Steiner Tree Problem in Networks

3.1 Introduction

This chapter describes the underlying graph-theoretic problem in multicast tree construction, the *Steiner Tree Problem in Networks.* Briefly, this problem concerns the creation of a graph that connects a set of nodes with the smallest possible cost. These nodes are a subset of the nodes that make up a larger network. The nodes in this subset are called *terminals,* and the graph that connects them with minimal cost is called the *Steiner Minimal Tree.* This problem is similar to that of the construction of a minimum spanning tree, with the difference being that not all of the nodes in the network need to be present in the resulting tree; only the terminals are required, although other nodes may be included. The Steiner Tree Problem in Networks has been shown to be NP-complete [41] and finding exact solutions to problems quickly becomes intractable as the network size grows. Many heuristics have been proposed for finding good solutions to the problem with tractable complexity.

We begin by defining the Steiner Tree Problem in Networks and the Steiner Minimal Tree. After that, we describe its relationship to multicast distribution tree construction and its application in static (a fixed set of receivers) and dynamic (a changing set of receivers) situations. Two techniques for finding exact solutions to the Steiner Tree Problem in Networks are presented, along with a set of *reductions* that reduce the size of the problem space when applying these techniques. Several heuristics for finding approximate solutions to the problem are also described. Finally, we develop a further heuristic that is appropriate for low-complexity construction of an efficient tree as the basis for a multicast tree construction algorithm.

3.2 The Steiner Tree Problem in Networks

Consider an arbitrary graph $G = (V, E, c)$, where V is the set of nodes[1] in the graph, E is the set of edges between nodes, and $c : E \longrightarrow \mathbb{R}$ is an edge length function. The Steiner Tree Problem in Networks is the problem of finding a subgraph of minimum cost containing *at least* a subset $N \subseteq V$ of nodes (called *terminals*.)

Hwang *et al.* [41] formulate the Steiner Tree Problem in Networks as follows:

- **GIVEN**: An undirected network $G = (V, E, c)$ and a non-empty set N, $N \subseteq V$ of *terminals*.
- **FIND**: A subnetwork $T_G(N)$ of G such that:
  - there is a path between every pair of terminals,
  - total length $|T_G(N)| = \sum_{e_l \in T_G(N)} c(e_l)$ is minimised.

The subnetwork $T_G(N)$ is called the *Steiner minimal network* for N in G. If all edges in G have positive length, $T_G(N)$ is called the *Steiner minimal tree* (SMT) for N in G.

The Steiner Tree Problem and its variants have many applications in the sciences, including group communication network routing in computer networks [8, 23]; circuit

[1]Hwang *et al.* use the word *vertices*. We have chosen to use *nodes* instead for consistency with other work on multicast and the rest of this thesis.

layout in electrical and chip design [33, 50]; and infrastructure layout, such as the design of networks of water pipes [80]. Garey *et al.* [30] have proven that the Euclidean Steiner Tree Problem is NP-Complete. Consequently, a great deal of research has been focused on the development of *heuristics* for the problem which are able to operate much more quickly and produce good results for problems that are too large to solve optimally in reasonable time.

In multicast tree construction, we build a logical distribution tree that connects a set of receivers to a source (or multiple sources) in a larger network. The paths through the network between these nodes may traverse additional intermediate nodes. If we consider the receivers and sources to be the terminals in the Steiner Tree Problem, then the optimal way to connect them so as to minimise network load (using the costs of the network paths between nodes as the metric) will be by finding the Steiner minimal tree for those terminals.

Furthermore, we may categorise multicast scenarios into two different groups, *static* situations and *dynamic* situations. In a static situation, the set of nodes to be connected and the underlying topology is fixed for the duration of the multicast session. In this case, the tree can be constructed at the start of the session and will not change for the duration of the multicast transmission. The minimal tree need only be calculated once. A good example of a static situation is a pre-arranged private video transmission, from the head office of an organisation to all of its regional offices over a private, fault-tolerant network. In this scenario, the source and set of receivers is known at the beginning of the session and will not change – so the tree only needs to be calculated once, used for the transmission, and then torn down.

In a dynamic situation, however, several of those parameters that are fixed in a static situation may change:

- The receiver set may not be fixed: users might join and leave the session for different periods of time over its duration.

- The underlying network topology might change: links may be added or removed, or nodes in the tree may fail.

An example of a dynamic situation is that of an Internet-based radio station. The station operates constantly and listeners connect when they wish to, listen for as long as they wish, and disconnect. The size and distribution of the receiver population in this scenario may vary considerably with time. One way to approach this scenario would be to construct a new distribution tree every time a change occurred – a receiver leaving or joining the tree, for example. This would ensure that the tree remained efficient, but would be very likely to involve significant control overhead and disrupt existing users' transmissions while the old tree was torn down and the new one constructed.

Another approach is to have the tree remain in place and then modify it locally as receivers join and leave. New receivers can be grafted on to the tree at an appropriate location, and leaving receivers can be removed. This reduces the complexity of the system and minimises disruption to existing receivers, but may result in a less efficient tree than one constructed directly for the current receiver set at a particular point in time. This is due to the continuing use of the pre-existing structure of the tree, generated over time by other join and leave operations. Near-optimal decisions made in the past may turn out to lead to significantly sub-optimal situations for future receivers joining the tree.

In *online* situations, where the distribution tree must be calculated on demand, exact methods to find the SMT cannot be used because of their prohibitive complexity. Calculation of the tree must happen quickly, so the receivers can join the session and begin receiving data. Multicast is one such online application of the Steiner Tree Problem in Networks, and for this reason heuristics for finding approximate SMTs are used instead of exact methods. Additionally, only limited information about the underying network topology or even the presence or absence of other nodes in the tree may be available, making finding an exact solution very difficult or impossible.

3.3 Exact Solutions

Many techniques for finding optimal solutions to the Steiner Tree Problem in Networks have been developed. In this section, we present two such techniques which scale according to different attributes of the graph being operated on. The first technique, Hakimi's Spanning Tree Enumeration algorithm [35] is a complete enumeration approach, solving the problem by considering every possible minimal spanning tree of every subset of the network's nodes that includes the terminal set. The second technique is a dynamic programming approach proposed by Dreyfus and Wagner [22] that generates the SMT by finding and combining the SMTs of smaller sub-graphs.

When describing the worst-case complexity of various algorithms in the following sections, we use v to represent the number of nodes, e to represent the number of edges and n to represent the number of terminals in the network.

Further approaches to solving the Steiner Tree Problem in Networks to optimality are described by Hwang *et al.* [41] and Winter [79].

3.3.1 Spanning Tree Enumeration

Hakimi's Spanning Tree Enumeration method [35] finds the SMT for a graph G by finding the minimum spanning tree for every set of nodes that contains at least the terminal set. Every possible set of nodes is tested, and the connected minimum spanning tree with minimum cost is an SMT for this terminal set on graph G.

There are several algorithms available for finding the minimum spanning tree of a graph. We selected *Prim's Algorithm* [61] for this implementation, which operates as follows (given a set of nodes as described above):

- begin by adding an arbitrary node from the set to the output graph;
- add the smallest edge that would connect a currently unconnected node in the set to the output graph;

- repeat the previous step until all nodes in the set are connected in the output graph.

Assuming that shortest paths between all nodes have already been calculated, the complexity of the algorithm as described here is $O(n^2 2^{v-n})$. The spanning tree enumeration method is polynomial in the number of terminals and exponential in the number of non-terminals, and hence it is most suited to solving problems where the majority of nodes in the graph are terminals.

3.3.2 Dynamic Programming

This approach, formulated by Dreyfus and Wagner [22], builds the SMT by considering small subsets of the terminal set, finding and storing the SMTs for those subgraphs, and iteratively forming larger Steiner minimal trees from minimal length unions of these smaller SMTs until the SMT for the complete terminal set is found.

Assuming that shortest paths between all nodes have already been calculated, the complexity of the algorithm is $O(3^n v + 2^n v^2)$. Hence, this approach is polynomial in the number of nodes, but exponential in the number of terminals. The dynamic programming approach is therefore suited to solving problems where the number of terminals is small compared to the total number of nodes in the larger graph.

3.4 Reductions

The task of finding the SMT can be made significantly easier with the use of pre-processing *reductions* that reduce the size of the calculation. These reductions can be broadly classified as *inclusion* reductions and *exclusion* reductions.

Inclusion reductions reduce the complexity of the problem by identifying nodes and edges that *must* be included in the SMT. Therefore, these nodes and edges can be placed in the output graph before the main calculation begins, and removed from consideration in the input graph. A simple example of an inclusion reduction is a test for terminal nodes of degree 1 (leaf nodes that are terminals). Since such a terminal must be in the output graph, so must the single node connected to it and the link between them.

Exclusion reductions identify nodes and edges that could not possibly be in the SMT, thereby enabling their removal from consideration by the main calculation. A simple example is a test for non-terminal nodes of degree 1. Such nodes cannot ever be on the shortest path connecting two terminals and hence may be removed without affecting the SMT being calculated.

These reductions can have an enormous impact on the running time of exact algorithms, such as the spanning tree enumeration and dynamic programming approaches described earlier. In many cases, simple Steiner tree problems can be solved through the use of reductions alone. In other cases, the set of nodes and edges in the input graph can be drastically reduced, decreasing the number of calculations necessary to (for example) enumerate all possible minimum spanning trees containing the terminal set, or calculate SMTs on smaller subgraphs of the terminal set.

We employed a number of reductions to facilitate the faster calculation of exact SMTs in our simulations. The list of reductions used is as follows:

Non-Terminals of Degree 1 Remove any node of degree 1 that is not in the terminal set, along with its connecting edge.

Non-Terminals of Degree 2 Remove any node of degree 2 that is not in the terminal set which can be "bypassed" by a shorter path between its neighbours. If it cannot be bypassed, remove it and add an equal length direct path between the two neighbours.

Paths with Many Terminals Any edge (v_i, v_j) with cost greater than the *bottleneck Steiner distance*[2] between v_i and v_j may be removed.

[2]The path P between two nodes v_i and v_j in graph G is made up of one or more elementary paths, where an elementary path is the subset of P that connects v_i and the next terminal (or v_j), two terminals, or a terminal and v_j. The *Steiner distance* is the length of the longest such

Non-Terminals of Degree 3 For a given non-terminal v with degree 3, if the minimum bottleneck Steiner distance between a pair of v's neighbours is less than or equal to the sum of the v's edge costs, then v can be removed along with its edges, and replaced by direct links of identical cost between its three neighbours. This is a specific case of a generalised "Non-terminals of degree k" reduction.

Terminals of Degree 1 Any terminal node of degree 1 *must* belong to every SMT, and so may be removed (along with its connecting edge) from calculation and added to the final tree immediately. This is the only *inclusion* reduction we use in our implementation.

Cut Reachability This exclusion reduction uses a tree spanning the terminal nodes (such as the output from a fast Steiner heuristic) to identify non-terminals that can be removed from consideration.

Detailed descriptions of these reductions (and several others) can be found in [41]. In our simulations, we performed the above reductions in the given sequence, repeating the sequence until none of the reductions modified the graph. Such repetition is necessary as some of the reductions (such as the Non-Terminals of Degree 3 reduction) introduce additional edges that may be subject to exclusion by another reduction.

3.5 Suboptimal Heuristics

Since finding the optimal solution for most graphs is so computationally intensive, even with reduction preprocessing, various heuristics are used in practice to find a good suboptimal solution. This is particularly true of multicast tree construction algorithms, which must be able to run online very quickly.

elementary path for a path P. The *bottleneck Steiner distance* is the minimum Steiner distance taken over all possible paths from v_i to v_j in G.

Other heuristics have also been developed for the Steiner Tree Problem in Networks - several are described in [41]. We chose to use a small subset of these, limiting our analysis to the faster algorithms available, since we are focusing on algorithms that could be used online to build multicast trees.

3.5.1 Shortest Paths (SP) Heuristic

The Shortest Paths heuristic operates as follows:

- Begin with an arbitrary terminal in the output graph.
- Find the closest terminal to the output graph that has not yet been added. Connect this terminal to the output graph via its shortest path to the nearest node in the output graph. Note that this node could be a terminal or a non-terminal.
- Repeat until all terminals have been added.

This algorithm requires the shortest paths from every terminal to every other node in the graph to be known. Consequently, its complexity is dominated by the calculation of these shortest paths. If Dijkstra's algorithm using heaps is used to calculate them, then the complexity of the SP heuristic is $O(n(e + v\log v))$ [41].

3.5.2 Minimum Spanning Tree (MST) Heuristic

The Minimum Spanning Tree heuristic operates by calculating the MST for the complete graph, and then removing each non-terminal of degree one until no more can be removed. In our implementation, Prim's algorithm was used to find the MST. The complexity of the MST heuristic is $O(e + v\log v)$ [41].

3.5.3 Shortest Paths Terminals (SP-T) Heuristic

This operates similarly to the SP heuristic described above, but only connects new terminals to terminals in the output graph, rather than to any node. Thus, it only requires shortest paths between pairs of terminals to be calculated. The SP-T heuristic operates as follows:

- Begin with an arbitrary terminal in the output graph.
- Find the closest terminal to the output graph that hasn't yet been added. Connect this terminal to the output graph via its shortest path to the nearest *terminal* in the output graph.
- Repeat until all terminals have been added.

This heuristic requires less information than the SP algorithm, as it only needs the shortest paths from every terminal to every other terminal. In practice, however, Dijkstra's algorithm must still be calculated on the whole input graph, as these shortest paths traverse intermediate non-terminal nodes which must be considered in the calculation. Hence, the complexity of the SP-T algorithm is the same as that of the SP algorithm, $O(n(e + v\log v))$.

3.5.4 Shortest Paths with Origin (SP-O) Heuristic

The Shortest Paths with Origin heuristic further limits the SP algorithm by requiring that all terminals connect to the tree via their shortest paths to an arbitrarily selected terminal. This terminal is the *origin*, or root of the tree. The algorithm proceeds as follows, assuming an origin has been selected and added to the output graph:

- Select an unconnected terminal T and determine the shortest path P from it to the origin.
- Add T and all the nodes and links in the path P to the output graph (unless they are already present).
- Repeat until all terminals have been added to the output graph.

This algorithm is similar to the SP and SP-T heuristics described above, but only connects new terminals via their shortest path to the *origin*, rather than selecting the nearest node or terminal. The complexity of the SP-O heuristic is $O(e + v\log v)$ [41].

3.6 Performance Analysis

All of the heuristics and both of the exact Steiner tree algorithms described in the previous section have been simulated on a variety of input data. The simulation software used is a custom-written package developed in the Python programming language.

The first set of input data (graphs and sets of terminals) used to compare these techniques came from the SteinLib Testdata Library[3] [45], data sets B and C. Both of these data sets contain sparse graphs with random edge weights and varying numbers of terminals. Data set B consists of eighteen problems with 50-100 nodes, and data set C consists of twenty problems with 500 nodes.

The second set of input data are the "Waxman" topologies that are used later for simulating dynamic multicast scenarios. These topologies consist of 500 nodes, 350 of which are leaf nodes, and are described in detail in Section 5.3.1. In these simulations all of our terminals are leaf nodes, to more closely model receivers in an Internet multicast scenario.

3.6.1 Exact Methods

The two algorithms for finding optimal solutions both scale according to different attributes of the network and terminal set, and are thus applicable to different classes of problem.

The spanning tree enumation algorithm requires very little memory (since it need only store one tree and the current minimum cost at any given time) but takes a

[3]Available on the Web at `http://elib.zib.de/steinlib/steinlib.php`.

great deal of time to compute, and scales exponentially with the number of non-terminals in the graph. Thus, it is suitable for denser graphs where the number of terminals is large relative to the total number of nodes.

The dynamic programming approach requires considerably more storage than the spanning tree enumeration algorithm, since it must compute and store a large number of subgraphs as it executes. In terms of computation time, however, it scales exponentially with the number of terminals rather than with the number of non-terminals. Thus it is most suitable for finding the solution to large, sparse graphs where the number of terminals is small relative to the number of non-terminals. We found that the memory requirements of this algorithm were such that for larger graphs we had to resort to the use of temporary disk storage, which considerably slowed the execution of the program.

3.6.2 Heuristics

Four heuristics for finding approximate solutions to the Steiner Tree Problem in Networks have been presented. All four complete in polynomial time, with their complexity dominated by the shortest-paths calculations that they require.

Three of the heuristics (the Shortest-Paths, Shortest-Paths-Terminals and Shortest-Paths with Origin heuristics) are *path heuristics*, building a tree spanning the terminals by connecting unconnected terminals to nodes already in the tree until all terminals are connected. The three heuristics differ in the number of nodes in the tree available for connection and consequently in the size of the search required.

The fourth heuristic (the Minimum Spanning Tree heuristic) is a *tree heuristic*, which builds a tree spanning the network first, then prunes non-terminals to form a smaller tree that is an approximation of the SMT. This is similar to the approach taken by the spanning tree enumeration technique for finding the exact solution, except that only one spanning tree is considered (the MST of all nodes) and then reductions are applied to prune that tree.

3.6.3 Results

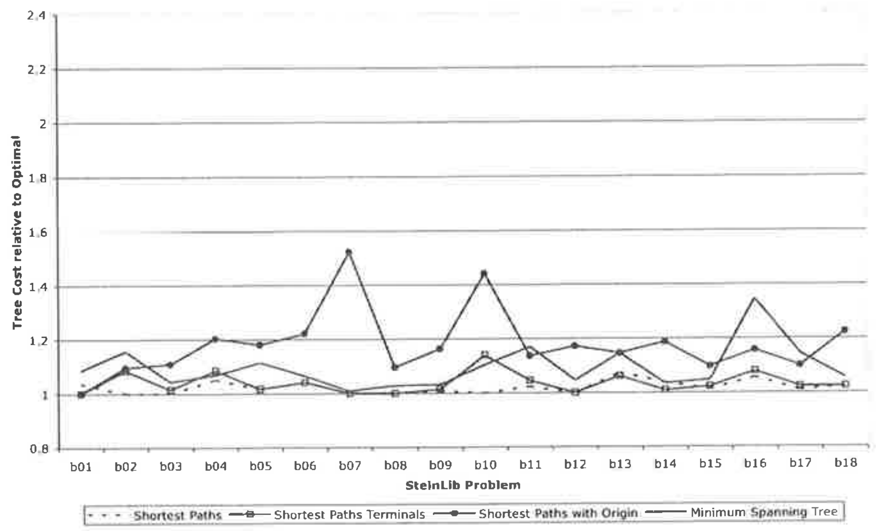
Two charts illustrating the competitiveness of the heuristics when run on the SteinLib B and C data sets are included as Figures 5(a) and 5(b) respectively. These charts show the cost of the final tree for each heuristic divided by the cost of the optimal SMT against each network in the data set. Note that Figure 5(a) is plotted on the same vertical axis as Figure 5(b) to simplify comparison.

For small graphs, the heuristics shown performed fairly similarly and quite well, occasionally even finding the optimal solution. As the problems grew larger, however, their performance tended to fall off, particularly that of the MST heuristic. The SP and SP-T heuristics performed remarkably similarly, despite the extra constraint that the SP-T heuristic could only connect new terminals to terminals, rather than to any node. Consistent with the requirement that it only join terminals to the "source" of the graph, the SP-O heuristic performed less well, but still produced a graph that was within 1.5 times the cost of the optimal graph in almost all cases.

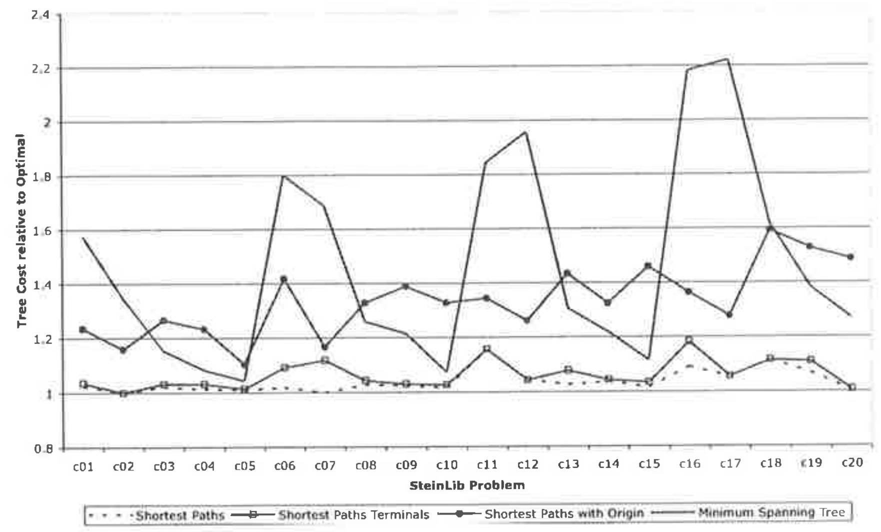
The MST heuristic performs quite differently, since it operates by constructing a larger tree and then removing nodes from it, rather than building the tree iteratively as in the other heuristics. Its performance in these scenarios is quite variable, suggesting that it is sensitive to properties of the input data that do not have so much of an effect on the performance of the other heuristics.

The MST heuristic performs much better on dense graphs, where the majority of nodes are terminals, than sparse graphs, where there is a small number of terminals relative to the total number of nodes. This is shown graphically by Figure 5(b), where the three peaks in the MST heuristic's chart (data sets C06, C07, C11, C12, C16 and C17) are all very sparse graphs, with 500 nodes and 5 terminals each.

Figure 6 is a graph of the heuristics' competitiveness on the larger Waxman network topologies. These results are the mean tree costs generated by each algorithm averaged over five different Waxman network topologies. For each heuristic, three different terminal sets were used, composed of 20, 50 and 100 terminals.



(a) SteinLib data set B: Tree costs for heuristics, relative to optimal SMT cost



(b) SteinLib data set C: Tree costs for heuristics, relative to optimal SMT cost

Figure 5: Tree costs relative to optimal trees for heuristics

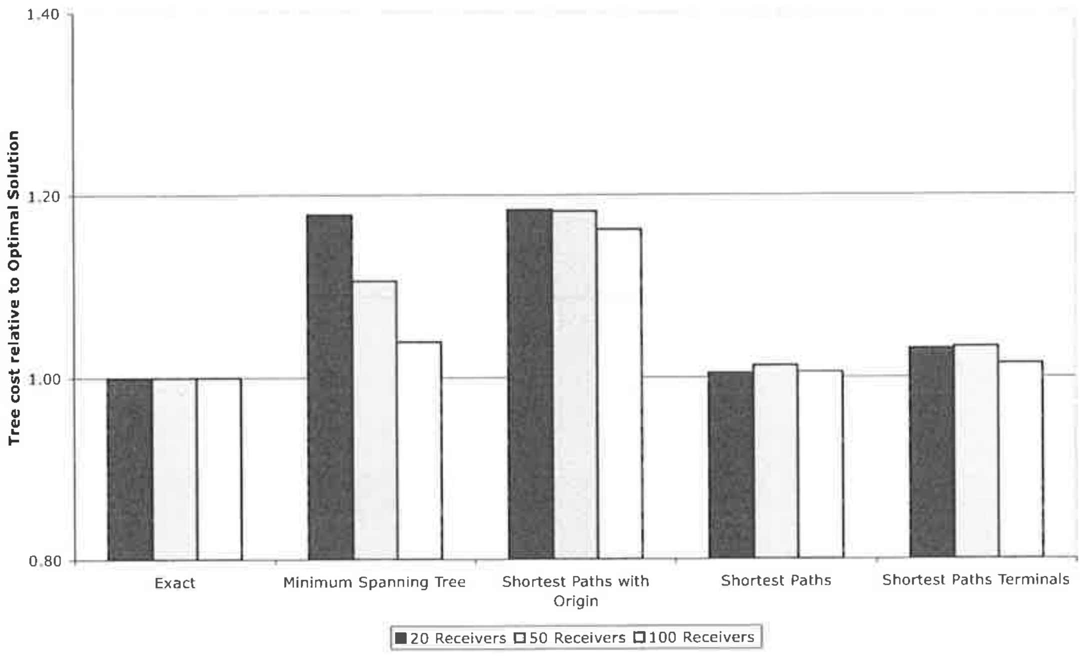


Figure 6: Waxman data set: Tree costs for heuristics, relative to optimal SMT cost

In these results, all four heuristics maintain a tree cost within 1.2 times the optimal tree cost. As in the SteinLib results, the SP and SP-T heuristics do very well, with the other algorithms producing more expensive trees. This is consistent with the larger amount of freedom to choose connection points on the tree available to those two algorithms. In contrast, the SP-O heuristic may only connect terminals to one point (the source) in the tree, leading it to generate longer paths and hence a more expensive tree. The MST heuristic once again shows that it performs better in denser scenarios than it does in sparse ones, with its performance improving as the number of terminals in the graph increases. It does not, however, outperform the SP and SP-T algorithms and is only marginally better than the SP-O heuristic in these examples.

3.7 The Steiner Tree Problem in Multicast

The Steiner Tree Problem in Networks is the underlying problem in the development of algorithms for the construction of multicast distribution trees on a network. The "terminals" in the Steiner Tree Problem represent the multicast group's members, connected together by paths through other intermediate nodes (non-terminals) in the network. The specific application of large-scale dynamic multicast over the Internet also adds a number of other constraints to the problem: in particular, the construction of the tree must be fast and operate with minimal information about the network topology. Centralised information about the topology of the whole network (as assumed in this chapter) is not available, and gathering information from large numbers of nodes online during a join is not feasible if the join is to complete quickly.

In this chapter, we have presented two exact algorithms for solving the Steiner Tree Problem in Networks and four heuristics for finding approximate solutions. The exact algorithms are not appropriate for use in a real-time multicast system, for two reasons: they require complete topological information, and they scale exponentially, requiring a great deal of computational time to find the solution. Neither of these requirements can be met in an online multicast tree construction scenario.

The four heuristics described earlier are the Shortest Paths (SP) heuristic, the Shortest Paths Terminals (SP-T) heuristic, the Shortest Paths with Origin (SP-O) heuristic and the Minimum Spanning Tree (MST) heuristic. The first three of these are *path heuristics* which construct a tree gradually by connecting unconnected terminals to nodes already in the tree until all terminals are connected. The MST heuristic is a *tree heuristic* which constructs a spanning tree first and then prunes unnecessary nodes. All four heuristics are able to produce approximate solutions within 1.2 times the exact solution's cost on the Waxman topologies used to simulate a multicast scenario, although the MST heuristic performs less well on the more sparse data sets in the SteinLib tests. These heuristics operate in polynomial time, much faster than the exact approaches.

However, there are still obstacles to the use of such heuristics in an online multicast situation. These approaches require considerable topological information – complete information about the network in the case of the MST heuristic, and a great many shortest-path measurements in the case of the others. Although it is possible to implement an Internet-based tree construction algorithm based on the SP heuristic, for example, such an algorithm would need to contact each node in the tree for every new receiver that joined it, in order to find the closest parent node. Such large amounts of control messaging are inappropriate for anything but small groups. At the other extreme, the SP-O heuristic requires no such calculations, but limits the efficiency of the tree by connecting every receiver to the origin; this approach relies on the presence of shared portions of these paths to provide efficiency savings through branching.

We feel that the use of a path heuristic is an appropriate way to design a multicast construction algorithm, but an appropriate compromise must be made between control messaging and performance. The algorithm must consider more possible parent nodes than the SP-O heuristic in order to promote branching in the tree and thereby improve performance, while not exhaustively searching the existing tree as is done in the SP and SP-T heuristics.

In the following chapter we will describe a multicast tree construction technique that develops from these ideas. We begin by describing the characteristics of our target application and the network on which it operates. From these attributes we create a series of requirements, which are used to develop a complete protocol for multicast tree construction and maintenance.

Chapter 4

The Lorikeet Protocol

4.1 A New Multicast Protocol

In previous chapters we have described some of the principles of multicast distribution and presented an overview of current Internet standards and research work in the area of multicast protocol design. We have also analysed the graph-theoretic problem that underlies multicast tree construction, the Steiner Tree Problem in Networks. It is clear from the large variety of multicast systems in the literature that there are many ways to approach the problem of tree construction and maintenance, and that all of these approaches have both advantages and disadvantages depending on the target application.

In this chapter we present a new multicast protocol, called Lorikeet. Lorikeet is targeted at single-source distribution of live multimedia content over the current Internet. It uses a hierarchical tree of unicast connections between the source, routers and receivers to deliver this data, requiring neither deployment on every router in the network nor traditional IP multicast infrastructure. Since this application is likely to be used in quite dynamic situations, receiver join and leave are low complexity operations designed to complete quickly. Additional support is provided for rearrangement of the tree to maintain efficiency as the receiver set changes.

We commence our discussion with a description of the characteristics of the application and the environment that we envisage for this protocol. From these characteristics, we build a set of requirements for Lorikeet. Finally, we describe the protocol itself and specify the behaviour of each operation it performs.

4.2 Design Goals

4.2.1 Application Characteristics

As described in Chapter 1, we seek to create a new protocol designed to efficiently deliver streaming multimedia content to home users using multicast over the current Internet. At present, streaming multimedia is predominantly served by simultaneous unicast applications, resulting in very inefficient use of network bandwidth which could be significantly reduced if a multicast system were available.

This application has a number of properties that directly affect the design of a multicast protocol built to facilitate it:

1. The data transmission can tolerate some loss, since the data being transmitted is multimedia data and, depending on the application requirements, some small loss of information will not be perceived by the user.

2. The application has a single source and only transmits content in one direction, from the source to the receiver.

3. People will request content manually and expect delivery to commence quickly, hence join operations must complete in reasonable time.

4. The transmission may be continuous (such as for a 24 hour news broadcast) or short-lived (for a live event, such as a music concert).

5. The set of receivers is unlikely to be static; it will probably change over time as new receivers join the session and other receivers leave.

6. Content providers are likely to want to be able to authenticate receivers and collect statistics on receivers.

4.2.2 Environmental Characteristics

Our protocol is targeted at current Internet technology and is thus subject to the properties of Internet networks and delivery mechanisms. Some of these properties that directly influence the protocol design are as follows:

1. The current Internet generally only provides best-effort delivery of packets, using a variety of underlying routing protocols. For a given stream of packets transmitted from node A to node B, there is no guarantee that all packets will be delivered, no guarantee that they will arrive in order, and no guarantee that they will all traverse the same path through the network from A to B. Quality of Service mechanisms like DiffServ [57] are available on some networks, generally for corporate use, but they are not yet widespread enough to rely on for providing guaranteed delivery to home users.

2. It is unrealistic to assume that the protocol will be deployed on all routers, everywhere. In order for a protocol to be practical on today's Internet, it must be incrementally deployable.

3. It is assumed that Internet routers can calculate an appropriate measure (hop-count, for example) of the "cost" of the path between two nodes (routers or receivers) in the network.

4. The current Internet does not necessarily provide symmetric links (with the same capacity in both directions) or symmetric routes (through the same set of links in both directions) through the network.

5. Access bandwidth (the capacity of the last link, between an end-user and their Internet service provider) is generally very limited, and high-bandwidth multimedia applications can often consume a significant fraction of that capacity.

End-users usually only have a single link to the rest of the network, through a consumer Internet service provider.

4.2.3 Requirements

From these application and environmental characteristics arise a number of system requirements, which we present and discuss in the following points.

1. *The system should assume only best-effort transmission of data and is not required to deliver all packets, or all packets in sequence.*

This differentiates the system from a *reliable multicast* system (for example, those presented in [47, 48, 58]), in which *all* of the data transmitted *must* be received by all of the receivers. We are focusing specifically on streaming multimedia data, which can tolerate some loss, and hence unreliable transport is acceptable, within certain constraints. Damage to streams through packet losses can also be mitigated through the use of application-layer or video coding mechanisms, like Forward-Error Correction (FEC) and layered coding.

2. *The system is designed for one way, single-source transmission.*

Multicast transmission should make more efficient use of network resources in delivering non-interactive content, such as streaming video. In such a case, the data transmission flows only from the sender to the receivers.

This application has a compelling commercial driver, since large amounts of network capacity are currently expended on single-source audio and video transmission [69]. Examples include webcasts of live events, movie trailers, adult entertainment and Internet radio stations.

Concentrating on the single-source case significantly simplifies the protocol, since there is no need to do sender discovery (as in many traditional IP multicast protocols) and the tree can be optimised for data coming from the single sender. A similar idea has been applied to traditional multicast in the form of Source-Specific Multicast (SSM) [9], currently under consideration by the IETF.

3. *Tree construction should be "good" in the sender-receiver direction.*

Most traditional multicast algorithms are *receiver-driven*, where the receiver joins the tree by contacting its nearest multicast-capable router. This means that the tree is constructed from the bottom up (from the receiver to the source), while the actual data flows in the opposite direction – from the sender(s) to the receivers. Such a tree construction mechanism is called a *reverse-path* join. This is inherently inefficient in today's Internet, where routing can be (and often is) quite asymmetric due to phenomena like hot-potato routing in the core [72] and the presence of asymmetric links (such as ADSL and satellite links) in the access network.

We therefore construct our trees in the downstream, or *forward-path*, direction, to match the direction of data transmission.

4. *Receivers must have no children in the tree.*

As stated earlier, one of the properties of the current Internet is that the access bandwidth (the capacity of the link between an end user and their Internet Service Provider) is generally quite limited. In many cases, this link is also asymmetric, with larger capacity available for transmissions *to* the user than *from* the user. This is one of the major issues faced by end-host multicast systems: the data rate of the transmission is necessarily constrained by the upstream bandwidth available to receivers, since receivers must re-send the data stream to other receivers. A second problem faced by systems in which end-host receivers must support other receivers is the fact that end-hosts generally have a single link to the network which must be traversed *twice* by the same data, whereas routers in the core generally have several links to the network.

We therefore set the requirement that receivers must be leaf nodes of the tree. All branching takes place further upstream, which more accurately reflects the capacity distribution and topology of the network and allows receivers to receive higher-bandwidth streams. In addition, requiring receivers to be leaf nodes simplifies leaving the tree and tree maintenance, since a receiver can simply be pruned from the tree without having to reconnect its children elsewhere.

5. *People will request content manually and expect delivery to commence quickly, hence join operations must complete in reasonable time.*

In order for the system to be usable by people in real-time, it must be designed to perform tree construction with enough speed to be responsive. For example, a user should not need to wait for a significant amount of time between submitting a request to join the tree and beginning to receive data.

6. *The system must efficiently handle dynamic membership.*

As described in Section 1.1.5, many modern applications for multicast transmission can have dynamic membership, with nodes joining and leaving the tree continually. The multicast protocol should be able to cope with these changes to the distribution tree and perform maintenance on it when required, in order to maintain its efficiency.

7. *The use of a multicast session (rather than a unicast transmission) should be transparent to receivers.*

Receivers should not be aware that they are participating in a multicast session, only that they are receiving data from the network. To put it another way, participating in the multicast session should appear to the receiver to be no different from receiving a unicast stream. Specifically:

- there should be no need to allocate a special multicast address (as in traditional multicast);
- sessions can be identified using URLs that are acquired out-of-band (for example, via a link on a Web page) – the URL identifies the source and the "name" of the stream;
- there is no need to worry about address collisions, because of the above two points;
- a connection is established with a server, as in a client-server application like the World Wide Web. This server may or may not be the source, as will be discussed in Section 4.5.2.

8. *Not all routers in the network need to support the protocol being used.*

Not all routers in the network need to support the protocol – the system should degrade to simultaneous unicast in the case of no routers that support the protocol being present (this is the worst-case scenario.) This is to maximise the utility of such a protocol, making sure it is usable even in the case where no routers that support the protocol are available in the network. In addition, the protocol should be capable of incremental deployment in the network without requiring changes to all of the routers across the Internet all at once.

4.3 Network Assumptions

As described in the requirements above, this protocol is designed to be deployed on today's Internet network without requiring large-scale changes to existing routing and transmission protocols. Hence, the use of standard UDP or TCP packets would be the most appropriate method for carrying control signalling, transmitted over standard unicast paths provided by existing Internet routing protocols. We assume that the network consists of a group of interconnected smaller networks, with limited capacity on links to end users (corresponding to home DSL links or small business links, for example) and large capacity in the core.

Tree construction and rearrangement operations will involve the calculation of the *cost* of the path between two nodes in the network. Both of these nodes will be capable of supporting the protocol, since both will be either receivers, participating routers or the source. Intermediate nodes on the path between the two nodes may not necessarily support the protocol – these nodes simply perform standard Internet forwarding. The cost metric used to select between different paths through the tree must be calculable at individual routers without requiring the involvement of a large portion of the multicast tree or the use of significant additional state information. Ideally, we would like the cost metric to be chosen so that the protocol's multicast trees are constructed with minimal total bandwidth usage.

Since we wish our protocol to be incrementally deployable, the only pieces of infrastructure we can modify are the participating routers themselves. This requirement rules out the creation of modified routing protocols that must be run on every (capable or not capable) node in the network. In addition, the only metrics that are available are those that can be calculated between capable routers and receivers, using the underlying Internet infrastructure. Traditionally, the available metrics have been *hop count* (the number of intermediate nodes) and *delay* (the round trip time, or time taken for a packet to be transmitted from one node to the other and back).

Our application, as described earlier, is a one-way streaming application, sending data from the source to the receiver set. Since there is no communication in the other direction, it is not generally sensitive to delay: delay is only an important factor if the application is time-sensitive or interactive. In our situation, promoting branching in the network and minimising the bandwidth used is much more important than minimising the delay to receivers. Therefore, we suggest the use of hop count as the initial metric to use in tree construction and rearrangement operations: it is easily calculable on the current Internet and requires no extension of existing infrastructure. In addition, it allows the paths used by the protocol to match those used for today's unicast traffic. Note that although hop count is an additive metric, additivity is not a required property for our metric: all that is necessary is a way to select the minimum cost path through the network from a set of candidate paths without requiring information from a large number of nodes.

Although hop count provides an immediately available (though coarse) measure of the efficiency of a path between two nodes, it has at best only a limited correlation with available capacity. If we wish our protocol to select its paths so that it minimises the proportion of available bandwidth used by the tree, it is necessary to discover the bandwidth of intermediate links in the network directly and use that as a cost metric. Such discovery is not possible on a wide scale today. However, this functionality may be developed in the future due to its utility in helping control the next generation of applications that have stricter quality of service and capacity requirements.

In such a scenario, we would propose the use of a cost metric based on the *bottleneck bandwidth* (the bandwidth of the smallest capacity link on the path) or *available bandwidth* (the proportion of the bottleneck bandwidth that is available at a given instant) of the path between the two nodes.

In this thesis we use positive integer costs in our simulations, representing a general cost, rather than restricting ourselves to the use of hop count in which the cost of every link is unity.

4.4 Control and Delivery

As described in Chapter 2, many different approaches to multicast control and delivery have been developed. All of these approaches share the central idea of constructing a logical tree of nodes, then using that tree to replicate data packets as they pass downstream from the source, in order to supply them to all of the multicast group's receivers. However, the mechanisms for constructing and maintaining the tree and for managing the delivery of data vary considerably, depending on the requirements of the intended application and the constraints imposed by the network environment.

The concept of *control* of the multicast tree encompasses all of the mechanisms used to manage the tree's structure: where state information about the tree is stored, how it is modified and which nodes should respond to events that occur in the multicast group. XCast [11], for example, is a small group multicast system in which the source manages all information about the tree, maintaining a list of receivers and transmitting that list in data packets to be replicated along with the data payload by downstream routers. In this case, routers in the network maintain no state information about the tree at all and merely "follow orders" from the source. In contrast, REUNITE [71] has no centralised list of receivers at all. Instead, each node in the tree maintains information on its parent and its children and forwards data from the former to the latter. These lists are *soft state* information, meaning that they must be periodically refreshed by regular communications from children or discarded after a timeout. Traditional IP multicast in its simplest form relies on an implicit tree, constructed from the bottom up by having receivers contact their local multicast routers to register interest in a particular group. Routers then share information with each other about the presence of group members and forward packets to all neighbours connected (directly or indirectly) to known group members. This approach is deliberately very decentralised in order to support multiple senders and large, dense groups. However, it requires that it be universally deployed in order to operate, and provides very limited capabilities for management and measurement of the multicast tree and its members. Multicast groups also require mechanisms to identify them – in the case of traditional IP multicast, this is a specially assigned IP address from the Class D address space dedicated to multicast, while in many other systems it is the address of the source or some other unique identifier.

Given the requirements enumerated in Section 4.2.3, it is clear that our protocol *cannot* require universal deployment and need only support single-source groups. Hence, the system must operate on top of current Internet protocols without requiring large-scale changes, and no complex sender-discovery protocols are required. Since we require tree construction to be in the downstream direction, the source must have some involvement in joining new receivers to the tree, rather than having new receivers join just by contacting their local router. In order to scale effectively to very large trees, the multicast tree cannot be stored in a central node, but must be stored in a distributed fashion. We also require that the protocol efficiently handle dynamic membership and complete join and leave operations quickly. To satisfy these constraints, we propose the use of a hierarchical multicast tree.

In a hierarchical multicast tree, each participating node maintains as little information as possible about the tree's structure – it need only know the locations of its children and its parent (and the source address in order to identify the particular multicast group.) This is similar to the scheme employed by REUNITE [71]. However, unlike REUNITE and most other algorithms, we have elected to connect receivers to the tree using their forward paths from the source. This requires the source to accept join requests from new receivers and search the tree for a nearby parent router. This search must be performed quickly, surveying a path through the current tree for a good match without exhaustively searching the tree (which would be too slow) or maintaining large amounts of topological information at the source (which would require a great deal of communications overhead to maintain). To this end, we propose using a limited search of the tree structure, starting at the source and examining a path through the tree, selecting the best parent from that limited search. This method allows us to retain only limited information about the tree at individual member nodes.

Since our protocol is limited to the use of current Internet protocols (as we cannot require universal deployment), our choices for delivery mechanisms are very limited. Traditional IP multicast is not universally deployed and hence cannot be relied upon as a delivery system. Therefore, we have decided to use a hop-by-hop unicast delivery model, similar to the recursive unicast approach used by REUNITE. A simple example of this mechanism is illustrated in Figure 7.

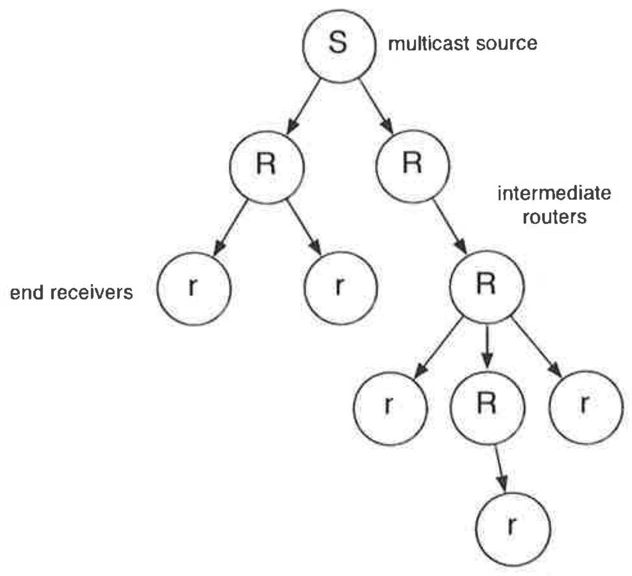


Figure 7: Hierarchical Multicast Delivery

This multicast group's participants are shown as a delivery tree, rooted at the source, with end-receivers as the leaf nodes of the tree. Data is transmitted hop-by-hop down the tree from the source, using direct unicast connections between each parent and child on the tree. When a node with children (a router) receives a packet from its parent, it replicates the payload of the packet and transmits a new packet with the same payload to each of its children, with the source set to its address and the destination set to the appropriate destination address for each child. This hop-by-hop approach, while "heavier" than traditional group-address-based multicast protocols which can simply forward packets, has a number of advantages:

- No group addressing is necessary – the tree is identified purely by the source address (and a path for selecting between different multicasts from the same source)
- Packets can be explicitly operated on for the benefit of receivers downstream of a router – for example, packets can be cached until acknowledged to provide local recovery, or streams can be stripped of detail to accommodate clients with lower capacity.
- Tree maintenance and rearrangement can be implemented, since we have control over the tree's topology.
- Aggregate statistics can be easily gathered – for example, counting the number of receivers in the network is straightforward and can be aggregated and passed upstream by routers.
- Authentication and charging is possible, since a join request must always reach the source of the tree.

It is important to note that each packet's data payload is unmodified at branching routers – replication of packets is done by simply copying packets and rewriting their headers, with no further processing required.

4.5 The Lorikeet Protocol

Our protocol, Lorikeet, is a hierarchical multicast system. Data is transmitted by the source to its children, who relay the data packets to their children. All receivers in a Lorikeet multicast tree are leaf nodes – hence there is no requirement for receivers to relay data to children. The relaying takes place in routers in the network that implement the Lorikeet protocol, here referred to as *capable routers*. All routers maintain a very small amount of state information; for a given multicast tree, each router knows the addresses of all capable routers on the path back to the source and the addresses of its children only.

Aside from the transmission of data by the source, a single-source multicast tree has a number of distinct events that can occur. In all multicast systems, a new receiver can *join* the tree and a participating receiver can *leave* the tree. In rearrangeable multicast systems, local portions of the tree (or, in some cases, the complete tree) can be *rearranged* periodically to improve efficiency. The following sections describe how these three events are implemented by the Lorikeet protocol.

4.5.1 Notation

We consider a network represented by a graph $G = (V, E)$ where V is the set of nodes and E is the set of edges connecting pairs of nodes. Overlaid on this network G a multicast distribution tree $T = (V_T, E_T)$ is constructed, where $V_T \subseteq V$ and $E_T \subseteq E$. The tree T is rooted at a source node $S \in V_T$. Every node in V_T except S must have an edge in E_T connected to a parent node in V_T, and may have other edges connected to nodes in V_T.

This multicast distribution tree has a single source node S. The other nodes are either *routers* or *receivers*. Receivers must be leaf nodes – that is, receivers cannot have children in the tree. The cost of the shortest path from node X to node Y is the total cost of all the links in the shortest path (X, Y) and is denoted by $\mathbf{cost}(X, Y)$. The cost of the tree path from X to Y, where $(X, Y) \in V_T$ and X is upstream from Y, is the total cost of the path (X, Y) using only links in E_T and is denoted by $\mathbf{tree}(X, Y)$.

The degree of a node is the number of links it has to other nodes (either to its parent or its children). Every node in the tree has one (and only one) parent node, with the exception of the source node, which has no parent. Routers, other than the source, with degree > 2 are described as *branching routers*, since they distribute data to more than one child node. Routers, other than the source, with degree 2 or less are referred to as *non-branching routers.* As stated earlier, routers that support the protocol are called *capable routers.*

4.5.2 Joining the Tree

A new receiver r joins the tree T via the following procedure:

1. The new receiver r contacts the source S using a standard unicast connection and tells S that it wishes to join the multicast tree for a given session (sources can potentially serve multiple sessions/trees).

2. S finds r a parent router R and returns that address to r.

3. R and r establish a unicast connection between them, and R begins to transmit the data being distributed by the multicast session on that unicast connection to r.

The path to a new receiver consists of an existing path through the tree from the source to the selected parent, to which is appended a unicast connection (using the underlying routing framework) from the selected parent to the new receiver. Before the receiver is connected to the parent, a probe packet using a defined Lorikeet port number is sent along this ultimate unicast path. Any capable routers along this path may sense this packet (by identifying its port number), announce their existence to the parent router, and join the tree. The first capable router encountered via this mechanism becomes a child of the parent router and the parent of the new receiver, and the process continues until the probe packet finally reaches the receiver.

Note that the source S may be a "repeater" that accepts streams from one or more other sources and redistributes them to a multicast tree using Lorikeet. This arrangement is similar to the idea of a rendezvous point (RP) described in Section 1.1.4, and allows several sources to transmit data over a single, centrally-managed tree. These sources are also "hidden" from the tree, freeing them from having to handle tree construction or deliver more than one stream.

Step 2 listed above (finding a parent for new receiver r) operates using one of the two join algorithms described in the following sections.

Simple Join

The Simple Join algorithm is designed to select a good parent node by using a decentralised, non-exhaustive searching procedure. It is conceptually similar to a single iteration (adding a single terminal) of a Steiner Tree Problem heuristic like those described in Chapter 3. This algorithm searches more potential parents in the tree than the Shortest Paths with Origin heuristic, but does not search the entire tree, like the Shortest Paths heuristic. Instead, it extends a single path downstream from the source until a router that is closer to the new receiver than all of its children is found.

Note that once the source hands off the search to one of its children, it is no longer involved in the rest of the join.

The Simple Join algorithm operates as follows:

2a. The current router $R_{current}$ is set to the source, S and the initial tree path contains only S.

2b. $R_{current}$ calculates $\mathbf{cost}(R_{current}, r)$

2c. If $R_{current}$ has no children in T which are capable routers and not receivers, then $R_{current}$ is made the parent router of r and Step 2 terminates.

2d. $R_{current}$ asks those of its children in T which are capable routers (and hence not receivers), denoted $C_1, C_2, ..., C_n$, to each calculate $\mathbf{cost}(C_k, r)$.

2e. $R_{current}$ receives these costs from its children and determines the child with minimum cost, identified as C_{min}.

2f. If $\mathbf{cost}(R_{current}, r) < \mathbf{cost}(C_{min}, r)$ then $R_{current}$ is selected as the parent router for r, terminating Step 2. If not, $R_{current}$ is added to the tree path and C_{min} becomes the new $R_{current}$. The process above repeats from step 2c, now being executed on the new $R_{current}$.

In implementation of this algorithm, of course, it is likely that some admission control would be necessary. A potential child router C_k may wish to reject a new receiver for a variety of reasons, including router load or insufficient downstream bandwidth on the next hop's interface. In these scenarios, the router C_k would reject its parent's request for a cost calculation and would thus be removed from consideration as a parent for that receiver. In the case of failed authentication for the multicast connection, the receiver would be rejected by the source upon making its initial connection.

Path-Greedy Join

The Path-Greedy Join algorithm builds on the Simple Join algorithm and trades further complexity for cost. It also searches a single downstream path through the tree, but does not terminate until a router with no further routers for children is reached. At this point, the *entire search path* is examined and the lowest-cost parent chosen from it. This approach requires slightly more management of the join procedure (such as passing along the complete search path) but has two advantages over the simple join: (1) it searches more tree nodes; and (2) it avoids terminating at a local minimum along the search path when a cheaper parent could be found further downstream.

In this algorithm, the join message that is passed down the tree also contains the search path so far, comprising the list of capable routers traversed and their associated costs to the new receiver.

The Path-Greedy Join algorithm operates as follows:

2a. The current router $R_{current}$ is set to the source, S, and the initial tree path contains only S and its associated cost, denoted by $\mathbf{cost}(S, r)$.

2b. If $R_{current}$ has no children in T which are capable routers (that is, its children are all receivers), then go to Step 2g.

2c. $R_{current}$ asks those of its children in T which are capable routers, denoted $C_1, C_2, ..., C_n$, to each calculate $\mathbf{cost}(C_k, r)$.

2d. $R_{current}$ receives these costs from its children, and determines the child with minimum cost, identified as C_{min}.

2e. $R_{current}$ and $\mathbf{cost}(R_{current}, r)$ are added to the tree path.

2f. C_{min} becomes the new $R_{current}$. The process above repeats from Step 2c.

2g. The final $R_{current}$ examines the tree path passed down to it and determines the router R_{min} in the tree path for which $\mathbf{cost}(R_{min}, r)$ is minimised. This router is returned to the source S, which notifies the new receiver r that its parent in the tree is R_{min}.

4.5.3 Leaving the Tree

The process for a receiver r leaving the tree is as follows:

1. The leaving receiver r contacts its parent router R and informs it that it wishes to terminate its connection to the tree.

2. The parent router R disconnects r from the tree.

3. If the parent router R has no remaining children after r's departure, it notifies its parent that it wishes to leave the tree and is subsequently disconnected by the parent. This process repeats recursively towards the source until a parent router with other children is encountered.

In addition to the above join and leave processes, Lorikeet can also perform rearrangement of the tree. Rearrangement in the Lorikeet protocol is described in more detail in the next section.

4.5.4 Rearrangement

Rearrangement of the tree is necessary to maintain efficiency when changes to group membership occur. As described earlier in Requirement 6 (see Section 4.2.3), large-scale multicast sessions may experience significant changes in the locations of their receivers. Since the construction of the tree is determined by the locations of receivers as they join, when those receivers leave and others join the tree will be unlikely to be as efficient as a tree constructed for the *new* group of receivers. To allow the tree to improve its structure to suit new receivers, we employ a rearrangement scheme that adapts the tree to cope with changes as receivers leave.

Rearrangement in Lorikeet is triggered when (as a result of a leave operation) the parent router of the departing receiver changes status from a *branching router* (router with two or more children) to a *non-branching router* (router with one child). We believe that this trigger for rearrangement based directly on a topological event is a novel technique. Other approaches in the area include: (a) triggering rearrangement by counting the number of join or leave events in an area and rearranging when a threshold is reached, as in ARIES; (b) triggering rearrangement periodically with a timer mechanism; (c) triggering rearrangement as part of a join event based on a performance criterion, as in DSG.

Two rearrangement algorithms were developed in the creation of this protocol: namely, the *Path* rearrangement strategy and the *Rejoin* rearrangement strategy. Both are described below.

Path Rearrangement

This rearrangement strategy focuses on the consolidation of long chains of routers with one parent and one child, by considering their replacement with direct unicast paths between the "top" and "bottom" branching routers in the chain. The algorithm operates as follows:

1. Router R detects that it has become non-branching as a result of the departure of one of its children.
2. R sends a message upstream via its parent towards the source looking for the nearest upstream branching router (router with two or more children), referred to as R_{up}. This message is passed from router to router (hop-by-hop) until a candidate is found. If no branching occurs upstream of R, the source S is selected as R_{up}.
3. R also sends a similar message downstream via its single remaining child looking for the nearest downstream branching router in that direction, referred to as R_{down}. If no branching occurs downstream of R, the final capable router on the path (which will be supporting a single receiver) is selected as R_{down}.
4. If $\mathbf{cost}(R_{up}, R_{down}) < \mathbf{tree}(R_{up}, R_{down})$, then R_{down} is re-parented to R_{up} via the unicast path, and intermediate routers (including R) are removed from participation in the tree.
5. If a rearrangement has taken place, new parent router R_{up} transmits a rearrange message down the new path to R_{down}, containing its path from the source. Each downstream capable router uses this path to update its tree path (used for loop detection), appends its own address to the path in the message and forwards it on to its children.

Note that in the case that no branching routers are discovered downstream of R, the furthest downstream capable router is selected as R_{down}, rather than the receiver it supports. This is due to the likelihood that the access bandwidth available to the receiver is likely to be considerably smaller than the upstream bandwidth of its parent router. Therefore, the parent router will be able to perform a rearrangement with less of an interruption to the flow of data, since it may maintain both the old and new connections while the rearrangement takes place. In addition, this allows the receiver's implementation of the protocol to remain simple, requiring only join and leave operations.

Figure 8 illustrates the Path Rearrangement approach.

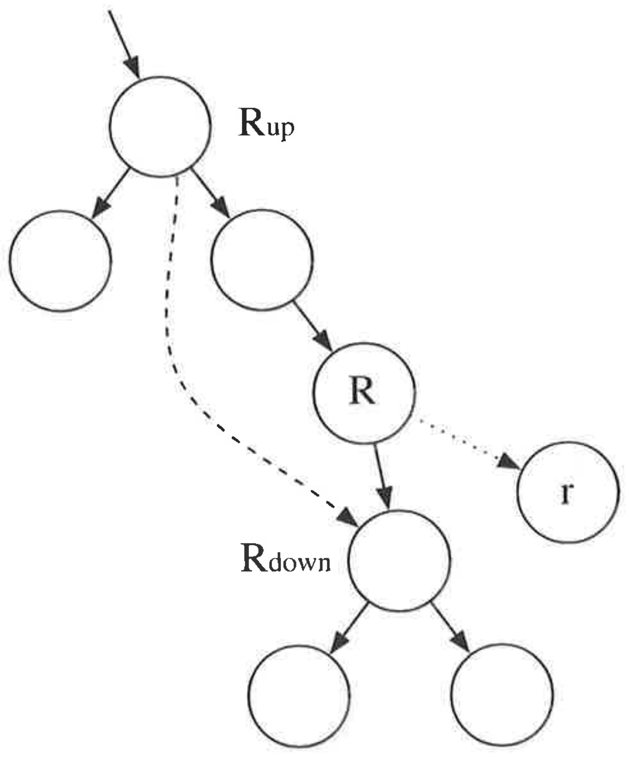


Figure 8: **Path Rearrangement**

Receiver r leaves the tree, turning its parent R from a branching router into a non-branching router. The current tree path (R_{up}, R, R_{down}) is then replaced with the shortest path from R_{up} to R_{down} if a shorter path is available.

Rejoin Rearrangement

Rejoin rearrangement allows a branching router to be reconnected to the tree at a parent with a lower parent-to-branching-router cost. It also allows the path to be shortened as in Path rearrangement, described above. The algorithm operates as follows:

1. Router R detects that it has become non-branching as a result of the departure of one of its children.

2. R sends a message upstream via its parent towards the source looking for the nearest upstream branching router, referred to as R_{up}. If no branching occurs upstream of R, the source S is selected as R_{up}.

3. R also sends a message downstream via its single child looking for the nearest downstream branching router, referred to as R_{down}. If no branching occurs downstream of R, the final capable router (a router supporting a single receiver) is selected as R_{down}.

4. R_{up} calculates the cost of the shortest path to R_{down}, $\mathbf{cost}(R_{up}, R_{down})$ and the cost of the current path through the tree, $\mathbf{tree}(R_{up}, R_{down})$ and returns both results to R_{down}.

5. R_{down} contacts the source of the multicast tree and finds a new parent router, R', using the join procedure described in Section 4.5.2.

6. R_{down} considers two types of rearrangement:

    (a) If $\mathbf{cost}(R', R_{down}) < \mathbf{cost}(R_{up}, R_{down})$, then the path $(R_{up}, ..., R_{down})$ is removed and R_{down} is connected to R' instead.

    (b) Otherwise, if $\mathbf{cost}(R_{up}, R_{down}) < \mathbf{tree}(R_{up}, R_{down})$, then we set R_{down}'s parent to R_{up} and remove the intermediate links and routers from the tree.

   (c) Otherwise, no change is made.

7. If a rearrangement has taken place, the new parent router (R' or R_{up}, depending on which rearrangement is performed) propagates a rearrange message down the new path to R_{down}, containing its path from the source. Each downstream capable router uses this path to update its tree path (used for loop detection), appends its own address to the path in the message and forwards it on to its children.

Figure 9 illustrates this approach.

Loop Detection and Subtree Inversion

Unfortunately, several problems are encountered when rearrangement is implemented on suboptimal trees, where paths through the tree may not be the underlying shortest paths. Since there is no global state information maintained in Lorikeet (for scalability reasons), routers have very limited knowledge of which nodes are in the tree – a given router knows of the source, its parent and its direct children, but has no further knowledge of the tree. This creates the possibility that loops may be created, whereby a router r tries to rejoin to the tree (as part of a rearrangement operation) and the underlying shortest path between the tree and r goes through a router which is one of r's descendants. Figure 10 (a) and (b) illustrate this behaviour: the new path chosen for the router r goes through v, which is a child node of r.

Note that in the Lorikeet protocol loops cannot occur during join or leave operations, as all Lorikeet receivers are leaf nodes and do not therefore have child nodes in the tree. Loops are only possible when nodes that are already in the tree are discovered on the shortest paths between capable routers.

This problem is solved by having routers (not receivers, of course) maintain a little more state information. Rather than simply storing the identity of its parent node, a router must store the complete tree path back to the source, that is the

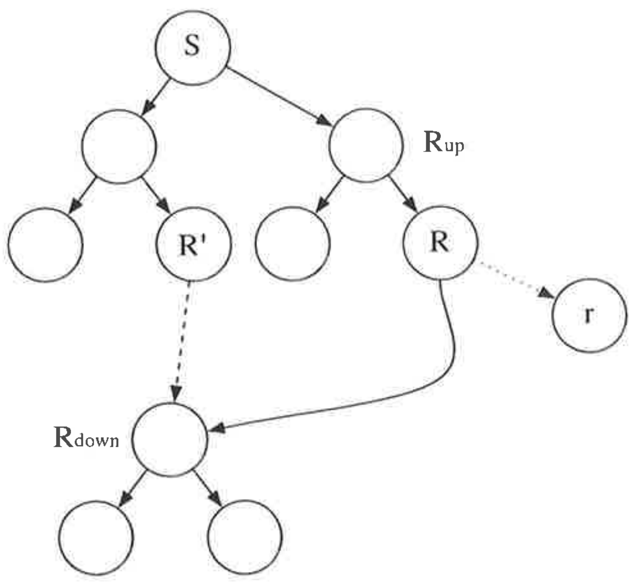
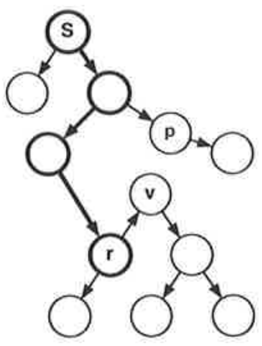
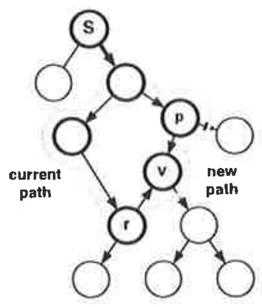


Figure 9: **Rejoin Rearrangement**

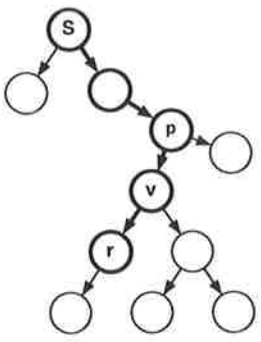
Receiver r leaves the tree, turning R from a branching router into a non-branching router. We search downstream from R for the first downstream branching router, R_{down}; and upstream for the first upstream branching router, R_{up}. **cost**(R_{up}, R_{down}) is calculated. R_{down} then contacts the source S, which finds a potential new parent R'. If **cost**(R', R_{down}) is less than **cost**(R_{up}, R_{down}), R_{down} becomes a child of R' and R is pruned from the tree. Otherwise, if **cost**(R_{up}, R_{down}) is less than the current value of **tree**(R_{up}, R_{down}), R_{down} becomes a direct child of R_{up} and R is pruned from the tree.



(a) Original Tree, with r to be rearranged. The node marked r is R_{down} as described earlier. Only routers (not receivers) are shown.



(b) New path to r from new parent p goes through v, a child of r in the original tree. Data is already flowing from r to v, so a loop occurs at the network level.



(c) Perform a subtree inversion, with v becoming the parent of r and a child of p. The link from r's old parent to r is removed, and the old parent of r will also be removed if it has no other children.

Figure 10: Loop Detection and Subtree Inversion

ordered list of upstream capable routers in the tree. This information is propagated down the tree as routers join it, constructed by taking their parent's path and appending their parent. The tree paths of affected routers are also updated on modification of the tree due to rearrangement.

With this extra information, router v is able to detect when one of its ancestors r wishes to become one of its children and perform a *subtree inversion*. This situation only occurs when v is discovered on the underlying unicast path between a selected new parent node p and r. In this case, the process that occurs is as follows:

1. v disconnects from its parent and becomes a child of p.

2. r then becomes a child of v.

The end result of this operation is shown in Figure 10(c).

4.5.5 Data Delivery

Data transmission in Lorikeet is performed by forwarding copies of packets after rewriting their headers. When a capable router in the tree receives a packet from its parent, it rewrites the packet's source address to match its own address. One copy of the packet is then sent to each of its children, with the destination address set to each child node's address accordingly.

This approach only requires the rewriting of fixed size headers and is therefore relatively cheap computationally. Since connections are only maintained between parent and child nodes, local acknowledgement of packets can be performed on a link-by-link basis if necessary without acknowledgements travelling further up the tree and swamping upstream routers or the source.

Since Lorikeet maintains explicit control of the tree by having its routers maintain direct connections with their parents and children, routers could potentially perform operations on the data being distributed through the tree. Such modification of content is difficult to achieve in other systems where topological information about the tree is not available at routers. Some examples of this functionality include:

- Local retransmission of packets on a link-by-link basis: for example, across a parent-child link. The parent may cache packets and require positive acknowledgement of every packet from the child. Then, the parent can retransmit a packet if it is not acknowledged within a set time period, without having to require the source to retransmit to the whole group or swamping the source with acknowledgements.

- Selective modification of the data as it traverses the tree. For example, the data being transmitted by the source could consist of a layered video stream, consisting of a low-bandwidth base layer and additional lower-priority layers that add detail to the base layer (as described in [55]). Each layer would be identified by a tag in the packet's header. It would be possible to have routers discard layers according to the demands of their children: a router whose children only have the capacity for 256kb/s of the video can only forward enough layers of the stream to satisfy that requirement, while another router further upstream might forward the complete 1Mb/s stream. Obviously, this system would require some minor modifications to the join algorithm in order to treat capacity as another component of the metric, thereby constructing trees with higher-capacity nodes placed nearer the source.

A more detailed discussion of Lorikeet's implementation and additional features that it potentially enables is presented in Chapter 7.

In this chapter we have described our target application, large-scale single-source multimedia transmission, and our network environment, the current Internet. From the properties of this application and the constraints imposed by the environment, we developed a set of requirements for a multicast protocol. The Lorikeet multicast protocol is designed to meet these requirements, providing a single-source group communication protocol that can support a large number of receivers and operate using unicast transmission on the current Internet, thus allowing it to be deployed incrementally. In addition, it provides support for rearrangement of the tree to maintain efficiency with a changing receiver set, using a novel topological trigger for these rearrangement operations.

The following chapter describes the simulation environment developed for the analysis of Lorikeet, and presents an analysis of Lorikeet's performance. Lorikeet's performance is analysed at various levels of deployment in the network and compared to the performance of other multicast protocols.

Chapter 5

Performance Analysis

5.1 Introduction

In Chapter 4, we presented a definition of the Lorikeet protocol for multicast tree construction, maintenance and data transmission. In this chapter, we analyse Lorikeet's performance in simulations of a number of different multicast scenarios.

First, we describe several other competitor algorithms that have been implemented in topological simulation for the purposes of comparison. These include two simple algorithms as baselines for comparison (the Source-Join and Greedy algorithms) and several other approaches from the literature (ARIES [8, 7], Delay-Sensitive Greedy (DSG) [32] and REUNITE/HBH [71, 18]), as discussed in Chapter 2. We investigate worst-case and average case message complexity of these algorithms in order to quantify the overhead they impose on the network.

We begin the performance analysis by examining the different approaches to Lorikeet tree construction (the Simple and Path-Greedy join algorithms) and tree rearrangement (the Path and Rejoin rearrangement algorithms) in order to determine which combination of these is most appropriate for general-purpose use.

The selected version of Lorikeet is then compared against other multicast algorithms in equivalent environments. Specifically, the simulations are of networks where all of the routers present are capable routers, rather than a mix of capable and non-capable routers. This property of universal deployment is a requirement of most of the other algorithms being simulated.

Finally, we analyse the performance of Lorikeet and the REUNITE/HBH protocol (neither of which require universal deployment) at different levels of deployment in the network, from zero capable routers to 100% capability. This allows us to confirm their performance in an incremental deployment situation, and predict what level of deployment yields a significant benefit from the use of multicast compared to simultaneous unicast.

5.2 Other Algorithms

The collection of algorithms described in this chapter are discussed here in enough detail to describe their behaviour in tree construction and maintenance. In implementing them for simulation, we found that in several cases their descriptions in the literature were incomplete or ambiguous – where this is the case, we have discussed these issues and their resolution. In particular, the ARIES protocol did not account for loops occurring as a result of its rearrangement process, an issue we addressed by modifying its rearrangement heuristic. We also present a more detailed discussion of the REUNITE algorithm and its extension HBH, as we feel that they are the most appropriate competitor protocols to compare to Lorikeet. Both REUNITE/HBH and Lorikeet share several basic properties, such as targeting unicast delivery over the current Internet, minimising the storage of state information and permitting incremental deployment. However, we show that the approaches employed by these protocols for tree construction and maintenance are quite different, as is their message complexity.

5.2.1 Source-Join and Greedy algorithms

The Source-Join algorithm is the simplest algorithm implemented. It performs a join operation by always connecting a new receiver to the source of the multicast by the shortest path (as selected by the underlying routing algorithm). Branching occurs when several of these paths share a router, which will request that it only receive one copy of the stream from the source. This algorithm is analogous to the *Shortest Paths with Origin* heuristic described in Chapter 3.

The Greedy algorithm performs a join operation by always connecting the new receiver to the nearest node in the existing multicast tree. This is done by exhaustively searching the tree, calculating the cost of the path between the new receiver and each capable router, and selecting the minimal cost path. This algorithm is the decentralised analogue of the *Shortest Paths* heuristic described in Chapter 3.

Both the Source-Join and Greedy algorithms in our implementations perform leave operations in the same way, by removing the leaving receiver and pruning upstream nodes recursively until a router with other children is found.

5.2.2 ARIES

ARIES (A Rearrangeable Inexpensive Edge-based On-line Steiner Algorithm) [8, 7] is a rearrangeable multicast algorithm designed to maintain efficiency in the face of changes to the multicast tree. Join operations are performed in the same way as the Greedy algorithm described above, using an exhaustive search of the current tree. Tree maintenance is done by rearranging a localised *region* of the tree whenever that region has enough receivers join or leave the tree within it. The definition of a region is quite complex – it is a portion of the tree that contains at least one modified node, or M-node (a node that has joined or left since the last rearrangement) and is bounded by stable nodes, or Z-nodes (nodes that have not been modified). Every Z-node keeps a counter for each region in which it participates which is incremented whenever a node joins or leaves that region. When this counter reaches a set threshold, a rearrangement of the region is triggered. When a node leaves the tree, it is marked as a deleted node, but not removed – this removal happens on the next rearrangement. Since both join and leave operations increment the counter, rearrangements can occur as a result of either type of event.

The following section describes the issues that arose while implementing ARIES in simulation, and the modifications made to address them.

Loop Detection in ARIES rearrangement

To rearrange a region, ARIES employs the Kruskal Source-Join Heuristic (K-SPH), although any static Steiner tree heuristic could be used, such as those in Chapter 3. K-SPH operates by deleting all the links and deleted nodes in the region, and then joining the remaining fragments together, joining the two closest fragments by the underlying shortest path in each iteration until only one fragment remains. To start with, the fragments are all individual Z-nodes, since M-nodes become Z-nodes on rearrangement. After the rearrangement the region ceases to exist, since it no longer contains any M-nodes.

We found that this approach presented a problem. Upon implementing it, we found that our trees would develop loops. This occurs because although K-SPH constructs a local connected graph, that graph is part of a larger overlay tree. Some of the paths between fragments being joined may contain nodes that are part of the larger tree, but not part of the region being rearranged.

Solving this problem requires expanding the initial fragments so that nodes in the region that are connected to the rest of the tree form one large fragment (or several large fragments, if the tree is partitioned by the region) that consists of the remainder of the tree. Unfortunately this operation requires a *search of the entire tree*, which somewhat reduces the advantage of limiting rearrangement to individual regions in the first place. Although changes to the tree only take place in this smaller region, the operation itself requires the participation of the whole tree.

For these reasons, we have limited our analysis of ARIES to a centralised, topological model, where its performance in terms of tree cost can be compared to that of other algorithms.

5.2.3 Delay-Sensitive Greedy (DSG)

The Delay-Sensitive Greedy (DSG) algorithm [32] is another more complex tree construction algorithm designed to also maintain the efficiency of the tree over time. It uses a Greedy join as described above, but allows for the re-connection of a node along the shortest path to the source if a delay constraint is not met by its initial location in the tree. This delay constraint is described in terms of the *stretch* of a node v, which is defined as $\mathbf{stretch}(v) = \mathbf{tree}(S, v)/\mathbf{cost}(S, v)$, where $\mathbf{tree}(\ldots)$ and $\mathbf{cost}(\ldots)$ are the multicast tree cost and shortest-path cost as defined in Section 4.5.1. If the stretch of node v exceeds a threshold, the first upstream node v' that satisfies a tighter bound is found and that that node is rerouted to the source along the shortest path from S to v'. Thus, DSG only performs tree maintenance as part of its join operation. Goel and Munagala [32] do not address the behaviour of leaving nodes, so we have implemented it in our simulations as described earlier for the Source-Join and Greedy algorithms.

5.2.4 REUNITE and HBH

In [71], Stoica *et al.* proposed the REUNITE (REcursive UNIcast TreE) protocol for multicast over recursive unicast trees. Later, in [18], Costa *et al.* developed HBH (Hop-By-Hop), a protocol that improves upon some of REUNITE's behaviour in asymmetric networks. In this section, we discuss first the salient features of the REUNITE protocol and subsequently the modifications proposed by HBH's authors.

As described in Chapter 2, Lorikeet shares many of REUNITE's advantages over both traditional IP multicast and many application-level approaches; the following points are paraphrased from the introduction to the REUNITE article [71] and are common to both REUNITE and Lorikeet:

- Reduction of forwarding state: information about nodes participating in the multicast tree is only maintained at a small number of nodes;
- No need for class D addresses: REUNITE uses unicast forwarding for both control and data transmission, identifying the multicast group by a source address (and port, or path);
- Incrementally deployable: the protocol will operate even if only a subset of network nodes deploy the protocol, rather than requiring complete deployment or tunnelling (as for IP multicast);
- Load balancing and graceful degradation: routers may choose to ignore control messages and the protocol will automatically use other routers to handle new joins or groups;
- Support for access control: since the source handles joins, access control can be implemented by authenticating receivers at the source.

However, there are a number of differences between the approaches used for tree construction and maintenance in REUNITE and Lorikeet. REUNITE's multicast tree for a group is stored (in a distributed fashion) in Multicast Forwarding Tables (MFTs) in routers participating in the tree. Each MFT contains a list of receivers that are downstream of that router, to which the router will send duplicate copies of any packets for the group that it receives. These MFTs are *soft state*: that is, they are maintained through the periodic reception of JOIN messages from these receivers. If a receiver does not send a JOIN message for a specified amount of time, its entry in the router's MFT is marked as "not alive" and eventually removed. A second table stored by routers in the tree is the Multicast Control Table (MCT). The MCT is used to keep track of which trees a non-branching router (one that merely forwards packets, rather than duplicating them) is a member of.

Lorikeet works differently in that its management of the tree is more explicit; state in routers is not soft, but is instead modified through explicit join and leave messages communicated between a node and its parent. State information is limited in a similar way, in that a router only maintains information about its children and its path back to the source, which is necessary to facilitate rearrangement of the tree.

We now discuss REUNITE's join and leave behaviour in more detail.

Joining the Group

REUNITE's join mechanism is similar to the Shortest Path approach described in Section 5.2.1. A receiver r wishing to join the group sends a JOIN message through the network, towards the source S. If there are no existing routers in the tree along the path (r, S), S creates an entry in its MFT for r and begins sending r data along the path (S, r). However, if there *does* exist a router R that is already a member of the tree on the path (r, S) traversed by the JOIN message, R will intercept this message and join r to itself, creating an MFT entry for r at router R. The authors of REUNITE describe this approach as constructing the multicast tree based on the *forward direction unicast* routing towards the receiver. This is correct for the first receiver in the tree, since it is by necessity connected directly to the source. However, subsequent receivers may be connected to tree routers found on their *reverse* unicast path back to the source using the process described above, rather than through a forward-path search originating from the source. We illustrate this behaviour with two examples, given below.

The example diagram shown in Figure 11 is taken from page 4 of the paper by Stoica *et al.* [71] that proposes the REUNITE protocol. In this scenario, the following asymmetric unicast routes are given:

- $S \rightarrow N1 \rightarrow N3 \rightarrow R1$;
- $R1 \rightarrow N2 \rightarrow N1 \rightarrow S$;
- $S \rightarrow N4 \rightarrow R2$; and

- $R2 \rightarrow N3 \rightarrow N1 \rightarrow S$.

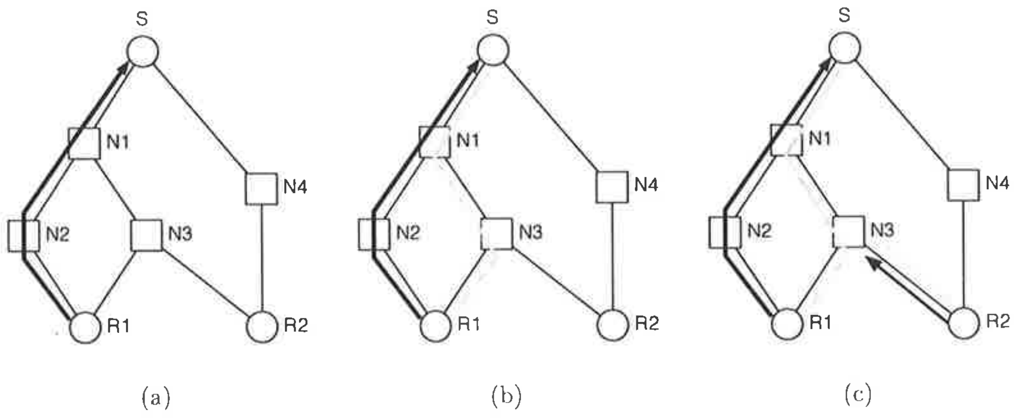


Figure 11: Example illustrating REUNITE's tree creation protocol, from Stoica *et al.* [71], page 4.

In this example, the receiver $R1$ is the first receiver to join the group (Figure 11(a)). Since no router in the network is aware of the group, the JOIN message sent by $R1$ is propagated all the way to the source S, which adds an entry for $R1$ to its MFT and begins to send data to $R1$. In addition to sending data, S also sends periodic TREE messages down the delivery tree (Figure 11(b)). When these messages arrive at routers $N1$ and $N3$, they update their MCTs to indicate that they are part of the multicast tree. These routers do not duplicate packets at this stage; they merely forward packets to $R1$.

In Figure 11(c), a second receiver $R2$ joins the group by sending a JOIN message towards S. When this message reaches $N3$ (the first router on the *reverse* path towards S that is a member of the delivery tree), $N3$ becomes a branching node: it removes its MCT entry for the group and creates an MFT entry for $R2$. Thus, a data packet that arrives at $N3$ (on its way to receiver $R1$) will be duplicated and also sent to $R2$ by $N3$.

This join has clearly been made on the *reverse* path from $R2$ to S through $N3$, rather than on the forward path $S \rightarrow N4 \rightarrow R2$. If $N4$, the only router on the forward path from S to $R2$, were a member of the group it would still not be used to supply $R2$ since the forward path from S to $R2$ is never examined during the JOIN procedure.

Consider the case where the forward and reverse paths between S and $R2$ are exchanged with each other, giving the following routes:

- $S \rightarrow N1 \rightarrow N3 \rightarrow R1$;
- $R1 \rightarrow N2 \rightarrow N1 \rightarrow S$;
- $S \rightarrow N1 \rightarrow N3 \rightarrow R2$; and
- $R2 \rightarrow N4 \rightarrow S$.

In this scenario, $R1$ joins as it did in the first example, and $N1$ and $N3$ once again become non-branching routers in the tree. This time, however, the forward path to $R2$ is via $N3$, which is already a participant in the tree. However, in this case, the JOIN message from $R2$ passes via the reverse path through $N4$, and (since $N4$ is not in the tree) $R2$ joins directly to the source, just as $R1$ did. Here, the use of the reverse path to join the second receiver has meant that the source must maintain two separate unicast connections directly to end receivers, rather than branching at $N3$ as would be done if a true forward-path delivery tree were constructed.

Broadly, REUNITE's join mechanism will only allow a receiver to join the tree at a router currently participating in the tree that is on the reverse path from that receiver to the source. If there are no such routers, the receiver will be joined to the source directly. This means that in multicast groups where receivers are sparsely distributed, the likelihood of discovering a *participating* router on the shortest path back to the source from a given receiver is low. In such a situation, not as much branching will take place and the total cost of the tree will be greater than in systems that promote branching by using the existing tree to join new receivers.

Leaving the Group

A receiver $R1$ leaves a REUNITE multicast group by simply stopping the sending of JOIN messages. This causes the router which has $R1$ in its MFT to timeout the entry for $R1$ (since it is not being refreshed any longer by periodic JOIN messages) and conclude that $R1$ has left the tree. The router cannot immediately stop sending packets to $R1$, however. Consider the first example described above, when $R1$ and $R2$ have joined the tree. If $R1$ now decides to leave, S cannot stop sending packets to $R1$ without interrupting the transmission to $R2$ as well, since $R2$ is supplied by $N3$ *without S's direct knowledge.* The approach to solving this issue that is used by REUNITE is to allow these receivers (that are dependent on the path to the leaving receiver) time to discover new branch points in the tree to receive packets from.

When $R1$ leaves, therefore, S marks its entry in its MFT as *not alive*, and keeps sending data and TREE messages down the delivery tree as before. These TREE messages are marked with a *stale* bit, indicating that the branch is to be removed in the near future. When downstream routers receive these stale TREE messages, they mark their MFTs as stale also (in the case of branching routers) or remove their MCTs for the group (in the case of non-branching routers). In our example, taking place after the situation shown in Figure 11(c), router $N1$ would remove its MCT and router $N3$ would mark its MFT as stale. This has the effect of causing later JOIN messages from downstream receivers to propagate further up the tree, to be intercepted by either the nearest non-stale participating router, or the source itself. These receivers effectively rejoin the tree further upstream on the same path to the source, at a participating router that is not on the stale branch. After a further timeout, the stale MFT entries at routers on the stale branch are removed, and the leaving receiver is removed from the tree. The result in our example scenario would be that the JOIN messages from $R2$ would propagate further up the shortest path, past the routers $N3$ and $N1$, which are stale or no longer participants, and reach S. $R2$ would become a child of S along its forward path, $S \rightarrow N4 \rightarrow R2$, and router

$N4$ would create an MCT indicating it was part of the group.

This approach to handling receiver leaves is necessary because REUNITE's design makes the stream being delivered to a particular router dependent on the presence of the receiver that established that branch. In Figure 11, for example, routers $N1$ and $N3$ are in the tree because they are supporting $R1$, which is a child of S. Later receivers that join those branches, however, join to "lower" routers in the tree, such as $N3$, without S's knowledge. This way of storing the tree leads to interdependencies between the receivers supported by a given branch of the tree: if this "first" receiver leaves and its branch is pruned, all the other receivers that depend on that branch must be re-parented.

Since REUNITE limits receivers to connecting to the delivery tree via sub-paths of their shortest paths to the source, it does not allow for rearrangement of the tree to improve its efficiency.

In contrast, Lorikeet's receivers have no interdependencies: a receiver can be simply pruned from the tree without affecting any other receivers, since its parent always maintains a direct connection with its child. If this pruning of a receiver leaves its parent with no further child nodes, we also prune the parent recursively. This approach requires no constant exchange of messages or timer-based mechanisms. Lorikeet provides mechanisms for rearrangement of the tree to improve efficiency as its participants change, such as the Path and Rejoin rearrangement techniques described in Section 4.5.4.

5.2.5 HBH

In [18], Costa *et al.* describe HBH (Hop-By-Hop), an extension of the REUNITE protocol that is designed to address several deficiencies in REUNITE's tree management. Their work makes a number of modifications to the the protocol:

- The authors suggest the use of Class-D multicast IP addresses (as in traditional IP multicast) for identification of groups, using the source-specific channel abstraction introduced in EXPRESS [38].

- HBH stores the next branching node in the MFT table, rather than the destination receiver. This modification makes the tree more stable than those constructed by REUNITE with respect to leaving receivers.

- HBH adds a third message type in addition to REUNITE's JOIN and TREE messages, the FUSION message. This message is sent by a router that receives several different TREE messages referring to different end receivers, and enables it to take over control transmissions to these receivers from the source.

The integration of IP multicast addressing into the protocol is not treated in detail in [18], and the authors list the formal definition of the interface between HBH and IP multicast addressing as future work. The other modifications to the protocol allow HBH to cope better with receivers for which the reverse path is different from the forward path (as in asymmetric networks), using the FUSION message to move management of these receivers further down the tree to the appropriate branching routers. In symmetric networks, HBH produces the same multicast trees as REUNITE does.

Since our simulations use only symmetric networks, the REUNITE and HBH protocols generate identical results in these scenarios; therefore the two protocols are referred to collectively for the remainder of this thesis.

5.3 Simulation

In order to analyse the behaviour and performance of Lorikeet and the other algorithms described in this chapter, we wrote implementations of each of them designed to operate on a simulation of a network. This simulation is a discrete event simulation of the network topology only, since we have chosen to focus on the critical issues of construction and maintenance of an efficient multicast tree, rather than on other aspects of the protocols.

Our simulation is a complete software package written in the Python programming language. A typical simulation follows the following sequence of steps:

1. Load a pre-calculated network graph from disk and initialise a multicast tree containing the source alone.
2. Calculate (or load from disk) the shortest-path information for the network, consisting of the shortest paths between all nodes.
3. Calculate (or load from disk) a sequence of join and leave events for the receiver set. A sequence is an ordered set of events, where each event consists of a single identified node joining or leaving the tree.
4. Iteratively run the sequence of events according to the selected multicast algorithm. For each event, we:
   (a) Determine the new multicast tree by simulating the operation of a multicast protocol for this event.
   (b) Write the resulting (intermediate) multicast tree to disk for analysis.
   (c) Calculate and write to disk a number of measurements (for example: tree cost, capable router count, receiver count).

All of the parameters used in simulation may be saved and re-run in order to ensure that (for example) the same sequence of events is used to simulate different protocols for comparison. The following section describes the two basic environments used for our simulations.

5.3.1 Simulation Experiments

To generate our results, two types of experiment are used:

- "Waxman" topologies (generated by the BRITE tool [56]) with 500 nodes, and 650 links. 350 of these nodes are leaf nodes (receivers), and the other 150 are routers. The sequences used on these topologies are randomly generated: for each event, a node is selected. The event is a join event if the node is not currently a member of the multicast tree and a leave event if it is. In all cases shown here, the results are averaged over five different "waxman" topologies (with the same number of nodes and edges).

- The "Windowed" topology is also generated by the BRITE tool (using its "2-level" topology support), with 2500 nodes and 3020 links. 2000 of the nodes are leaf nodes, and 500 are routers. The sequence used is generated by giving every node a geographical location on a 2D grid measuring 1000x1000 units (this data is provided by BRITE), then having a "join window" travel from left to right across the grid selecting nodes to join to the tree. A second window, travelling behind the first, selects nodes to leave the tree. This approach is used to simulate the correlation of network joins in some situations, such as a 24hr broadcast where receivers are likely to be correlated by timezone. The windowed simulation has several parameters: (a) the initial count of receivers in the tree; (b) the rate at which the window moves across the plane; (c) the window size; and (d) the departure lag, or lag between the joining and leaving windows.

5.4 Complexity Analysis

The complexity of distributed algorithms like these multicast tree construction algorithms is very difficult to quantify because of their dependence on the network and overlay topologies. Algorithms may perform very differently on networks where receivers are clustered close to each other compared to when they are more sparsely distributed, for example. In addition, the ordering of join and leave operations is quite significant: the addition of new routers to the overlay tree can affect the selection of parent nodes in later joins, as can the pruning of routers that are not needed at the time.

We feel that the best way to describe these protocols is in terms of *message complexity*, where we analyse the number of messages between nodes that an algorithm takes to perform an operation. All algorithms presented in this chapter have two basic operations: the join operation and the leave operation. Tables 1 and 2 present the lower- and upper-bounds for the message complexity of these operations. As shown in the tables, the Lorikeet algorithms have much smaller lower-bounds on complexity than the Greedy-join based algorithms, while maintaining the same upper-bound.

Two special cases in analysing these algorithms for complexity are the Source-Join and REUNITE algorithms. As shown, both of these have very low message complexity for both join and leave operations, since all receivers join at and leave from the source via the (already known) underlying shortest path. However, this approach trades this reduced complexity for increased tree cost: it relies on receivers sharing portions of their paths from the source, rather than explicitly making use of routers that are in the tree but not on that receiver's shortest path from the source. An implementation of the Source-Join algorithm would require additional complexity to achieve this path sharing, either by having the source maintain a copy of the topology, or by having routers detect and consolidate duplicate streams. REUNITE is effectively an implementation of the Source-Join algorithm: it operates using additional periodic messaging to maintain its soft state forwarding tables at routers, thereby facilitating multicast branching. For this reason, the complexity figures given for Source-Join and REUNITE are significant underestimates, since either additional functionality or periodic messaging (which is difficult to compare to more explicit control systems) is required in real implementations of these algorithms.

These lower- and upper-bounds only present the behaviour of the algorithm in the trivial and worst-case scenarios; thus, they do not represent the behaviour of the system in the vast majority of cases. To give a more complete picture of the behaviour of Lorikeet against other algorithms we counted the messages passed over the course of a simulation and present the results as averages in Table 3. The simulation used to generate these results is the "Windowed" experiment described in

Section 5.3.1, with the following parameters: the initial count of receivers is 100, the rate at which the window moves is 0.5 units/event, the window size is 200 and the departure lag is 100. Results for ARIES are omitted from this table because of the difficulty in developing a distributed implementation of ARIES, as described in Section 5.2.2. However, the number of messages passed by the ARIES algorithm will be at least as great as the number passed by the Greedy algorithm, plus the additional cost of the messages required to maintain ARIES' counters and perform rearrangement. Results for REUNITE are also omitted due to its dependence on periodic messaging to maintain its soft state information; this time-dependence makes REUNITE's message complexity difficult to compare to the other event-driven protocols presented here.

It can be seen from the table that the exhaustive join technique's higher complexity gives rise to a much larger average number of messages passed on join events for both the Greedy and DSG algorithms, and also for ARIES (as noted above). This makes these algorithms unsuitable for real-time use on multicast transmissions with more than a trivial number of receivers, since this control overhead will grow at least linearly with the number of receivers. It is also worth noting that the join complexity is more significant than the leave complexity in multicast protocols, as a receiver must wait for a join to complete before it can begin receiving data. Hence, we feel that only algorithms which do not perform exhaustive searches on join, but instead limit the size of their searches further, are candidates for real-world use.

Although Lorikeet has the same worst-case complexity as the Greedy algorithm, its best case is an order less complex, and in the average case each join only takes a small number of messages. This is due to the way that Lorikeet performs its join operations, only expanding one branch of the tree at each branch point. Table 3 also shows that the difference between Simple and Path-Greedy join in terms of the number of messages passed is marginal, with only an increase of between three and four messages on average for Path-Greedy join in this scenario. Rearrangement in the Lorikeet variants is only ever performed on leave operations, because of the

Algorithm	**Lower Bound**	**Upper Bound**
Simultaneous Unicast	$\Omega(1)$	$O(1)$
Greedy	$\Omega(N)$	$O(N)$
Source-Join	$\Omega(1)$	$O(1)$
REUNITE	$\Omega(1)$	$O(1)$
DSG	$\Omega(N)$	$O(N)$
Lorikeet Simple	$\Omega(1)$	$O(N)$
Lorikeet Path-Greedy	$\Omega(1)$	$O(N)$

Table 1: Join Event Complexity

trigger that is used (a router becoming non-branching).

The values given for the standard error in these results are quite large for most of the algorithms. This is due to the fact that the number of messages passed per operation vary a great deal from one operation to the next and are highly dependent on the topology of the tree at the time. For example, a join operation that takes place shortly after the creation of a tree requires fewer messages to find a good location compared to a join on a tree which already has several hundred participating nodes.

Table 4 shows the number of attempted and successful rearrangements for the simulation used to calculate the data given in Table 3. It shows that rearrangement is more often successful with Path-Greedy join than with Simple join, and moreover that Rejoin rearrangement is slightly more often successful than Path rearrangement.

It is obvious from the data that Rejoin rearrangement is more expensive than Path rearrangement, since it requires an additional join operation. However, we feel that the extra cost for Rejoin rearrangement is acceptable since rearrangements are performed in the background; in other words, "hidden" from the user.

Algorithm	**Lower Bound**	**Upper Bound**
Simultaneous Unicast	$\Omega(1)$	$O(1)$
Greedy	$\Omega(1)$	$O(N)$
Source-Join	$\Omega(1)$	$O(N)$
REUNITE	$\Omega(1)$	$O(1)$
DSG	$\Omega(1)$	$O(N)$
Lorikeet (either) without Rearrangement	$\Omega(1)$	$O(N)$
Lorikeet (either) with Path Rearrangement	$\Omega(1)$	$O(N)$
Lorikeet Simple with Rejoin Rearrangement	$\Omega(1)$	$O(N)$
Lorikeet Path-Greedy with Rejoin Rearrangement	$\Omega(1)$	$O(N)$

Table 2: Leave Event Complexity

Algorithm	**Join**	**Leave**	**Rearrange**
Simultaneous Unicast	1.00 ± 0.00	1.00 ± 0.00	
Source-Join	6.98 ± 2.45	1.94 ± 1.32	
Greedy	178.41 ± 33.74	1.64 ± 0.96	
DSG	182.16 ± 33.63	1.67 ± 1.00	
Lorikeet Simple	16.88 ± 4.49	1.86 ± 1.17	
Lorikeet Simple, Path rearrangement	16.88 ± 4.49	2.23 ± 1.23	2.09 ± 1.17
Lorikeet Simple, Rejoin rearrangement	16.85 ± 4.49	2.21 ± 1.20	17.75 ± 4.73
Lorikeet Path-Greedy	19.52 ± 5.57	1.74 ± 1.24	
Lorikeet Path-Greedy, Path rearrangement	19.06 ± 4.97	1.98 ± 1.19	3.79 ± 4.71
Lorikeet Path-Greedy, Rejoin rearrangement	19.04 ± 5.29	1.97 ± 1.18	19.48 ± 6.82

Table 3: Messages Passed Per Event

In this table, the Join column lists the message count for join events, the Leave column lists the message count for leave events where no rearrangement takes place, and the Rearrange column lists the message count for leave events where rearrangement takes place. The mean and standard error are given for each count.

Algorithm	Attempts	Successful
Lorikeet Simple, Path rearrangement	291	0
Lorikeet Simple, Rejoin rearrangement	291	9
Lorikeet Path-Greedy, Path rearrangement	296	42
Lorikeet Path-Greedy, Rejoin rearrangement	297	54

Table 4: Rearrangement Counts

Number of attempted and successful rearrangements. The total number of leave events in this simulation was 964.

5.5 Tree Cost

In this section, we analyse Lorikeet's performance in terms of the total cost of the tree; that is, the sum of the costs of the individual links connecting nodes in the tree. We begin by analysing the different variants of Lorikeet described in Chapter 4: the Simple and Path-Greedy join algorithms and the Path and Rejoin rearrangement algorithms. The next section compares Lorikeet's performance to that of the other algorithms described earlier, in a complete deployment scenario (where all routers in the network support the protocol.) Finally, we analyse Lorikeet and REUNITE at different levels of router capability, to determine how they respond to incremental deployment in the network.

5.5.1 Lorikeet Join and Rearrangement Operations

This section presents a comparison of the different variants of the Lorikeet protocol. These can be divided into two categories: different join algorithms and different rearrangement algorithms.

Joining the Tree

As presented in Section 4.5.2, we have examined two different approaches to joining a Lorikeet multicast tree, the *Simple* and *Path-Greedy* joining algorithms. The Simple join algorithm is the initial algorithm developed to selectively search the existing tree for a parent for a new receiver. The Path-Greedy approach extends the Simple join to terminate the search later and avoid local minima. Both algorithms were simulated, and Figure 12 shows the total tree cost over an event sequence, averaged over five sequences on the same topology, with all routers being capable routers. The Greedy joining algorithm is included as a baseline for comparison.

We can see that the Path-Greedy algorithm provides approximately a 5% improvement over the Simple algorithm in the total cost of the tree once the system reaches a stable number of receivers. This improvement comes at a small additional cost in terms of message passing and implementation complexity, as shown in Table 3. Both algorithms are within 12% of the cost of the Greedy algorithm, which has much higher complexity than either Lorikeet approach.

Rearranging the Tree

Section 4.5.4 presents two different techniques for performing rearrangement of the tree in order to maintain efficiency with a changing population of receivers. The simplest approach is *path rearrangement*, which optimises the path between two branching routers in the tree, and the more complex approach is *rejoin rearrangement*, which rejoins a branching router to the tree to take advantage of new parent routers, falling back to path rearrangement if a rejoin would not improve the cost of the tree.

Table 5 shows the tree cost averaged over the stable portion of an event sequence (events 500-2000) measured for all of these rearrangement techniques. The cost is shown relative to the tree cost with no rearrangement. In this simulation, the "Windowed" experiment (described in Section 5.3.1) is used, as rearrangement is more effective when node joins and leaves are correlated. The parameters used here are as follows: the initial count of receivers is 100, the rate at which the window moves is 0.5 units/event, the window size is 200 and the departure lag is 100.

These results show that rearrangement of the tree provides a cost improvement

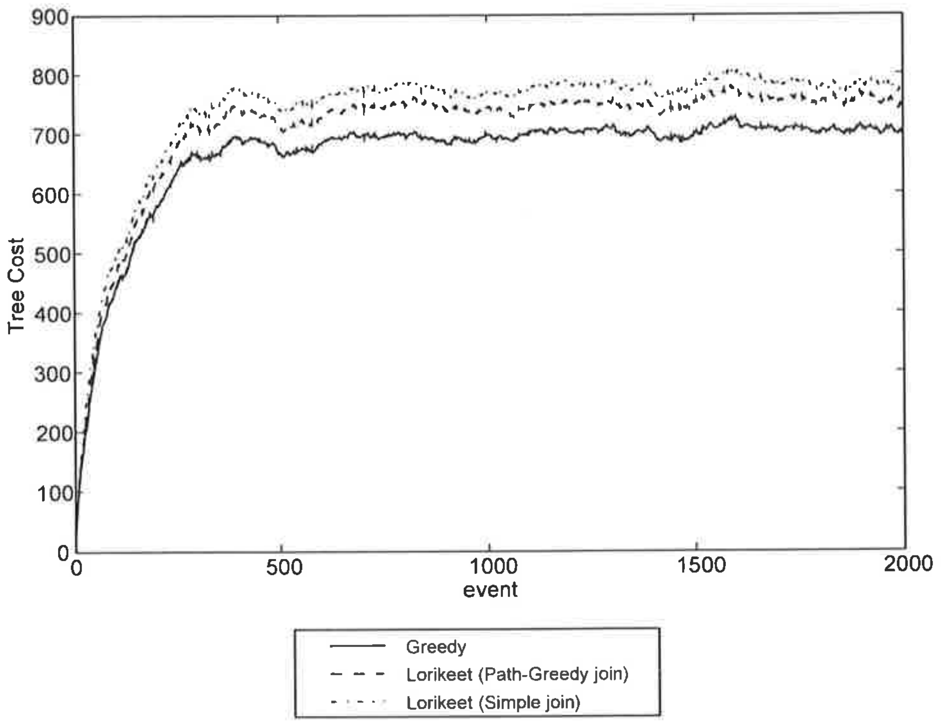


Figure 12: Comparison of Join Techniques: Simple Join vs. Path-Greedy Join

Simple Join

Algorithm	**Mean**	**Min**	**Max**
No rearrangement	1.00	1.00	1.00
Path rearrangement	1.00	1.00	1.00
Rejoin rearrangement	1.00	0.98	1.01

Path-Greedy Join

Algorithm	**Mean**	**Min**	**Max**
No rearrangement	1.00	1.00	1.00
Path rearrangement	0.94	0.85	1.00
Rejoin rearrangement	0.91	0.80	0.98

Table 5: Comparison of Rearrangement Techniques (events 500-2000)

of up to 20% over no rearrangement in this scenario. The addition of rearrangement to the Path-Greedy join algorithm provides a significantly greater reduction (up to 20%) in the cost of the tree than the addition of rearrangement to the Simple join (up to 2%). This improvement in tree cost is more evident in the Rejoin rearrangement results, since the Rejoin algorithm inherits the join mechanism as part of its operation: using the more effective join algorithm improves the performance of the rearrangement.

From these results, we find that the combination of the Path-Greedy join algorithm with the Rejoin rearrangement mechanism offers the best performance in terms of total tree cost. The Path-Greedy join does not add significantly to the message complexity of the system compared to the Simple join algorithm, and the complexity of performing rearrangement is no more expensive than performing an additional join. Since these rearrangements occur infrequently and can be performed asynchronously without affecting receivers directly, we feel that this additional cost is justified.

For the remainder of this chapter, we use the Path-Greedy join with Rejoin rearrangement for all Lorikeet results.

5.5.2 Comparing Lorikeet and Other Algorithms

All of the other algorithms except REUNITE referenced in this chapter make the assumption that the protocol is supported by every router in the network; effectively, every node in the network is a capable router. In addition, they make no distinction between end-receivers and routers, thus allowing end receivers to support child nodes.

Thus, to make a "fair" comparison, we test a simulation of Lorikeet against simulations of the ARIES, DSG, REUNITE, Greedy and Source-Join algorithms with all routers in the network marked as "capable" for Lorikeet's purposes. Our network topology is constructed in such a way that receivers are nodes with degree 1 (leaf nodes) so as to address the second restriction. The simulation used is the "Waxman" experiment described in Section 5.3.1.

The graph in Figure 13 shows the tree cost relative to the Greedy approach for each algorithm during a sequence of events, where an event constitutes a new receiver joining the tree or a currently-joined receiver leaving the tree. The graphs of the cost of all of the algorithms except ARIES follow a similar shape to each other, with the Greedy algorithm consistently having the smallest cost and the Source-Join algorithm having the largest cost. This is consistent with the expected behaviour of both algorithms, since the Greedy algorithm uses an exhaustive search of the multicast tree for the optimal parent node on join, while the Source-Join algorithm always joins to the source. The Source-Join and REUNITE algorithms produce identical results, since they both perform a join along the shortest path to the source for each receiver.

Note that the difference between the best- and worst-case algorithms on this graph is approximately 15%, and that (as shown in Figure 2 in Chapter 1) all of these algorithms are far more efficient than using simultaneous unicast, which has a

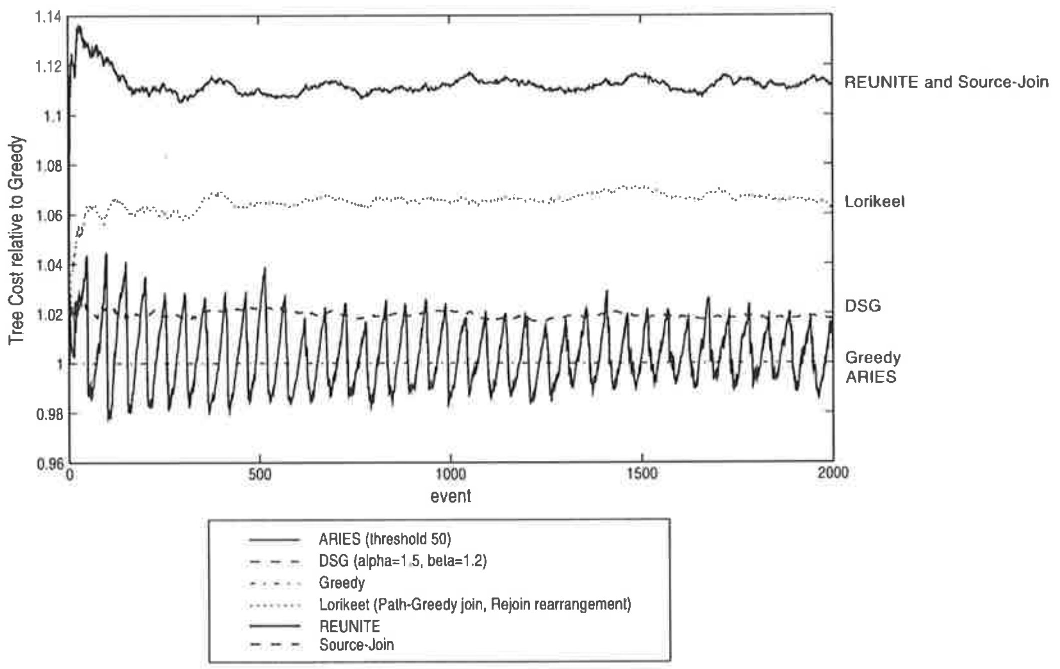


Figure 13: Comparison between Lorikeet and other Tree Management Heuristics

tree cost of more than *five times* the Source-Join algorithm's cost in this simulation.

Lorikeet and DSG fall between these two algorithms, which is also consistent with their expected behaviour. Both of them consider more options when choosing a parent node for a new receiver than the Source-Join approach, but are not considering as many nodes as the Greedy algorithm. In DSG, this is because DSG needs to rejoin nodes directly to the source when they fail the stretch test, and in Lorikeet this is due to the use of a join algorithm that does not perform exhaustive search. However, Lorikeet's join algorithm does perform significantly better than simply joining all receivers directly to the source, as in Source-Join or REUNITE.

The ARIES algorithm uses an exhaustive join as described earlier and prunes deleted nodes not when they leave the tree, but as part of a periodic tree rearrangement process. This process also rearranges parts of the tree to form local minimal Steiner trees. In this example, the rearrangement threshold is 50, meaning that 50 join/leave events must happen in a region of the tree before that region is rearranged. This behaviour gives the "sawtooth" pattern shown on Figure 13, which straddles the Greedy line since ARIES essentially uses the Greedy joining approach. ARIES is able to improve on the cost of the Greedy algorithm's tree through the use of rearrangement.

This figure shows that Lorikeet's performance falls between that of the Source-Join algorithm and the algorithms based on exhaustive joining heuristics (Greedy, DSG, ARIES). This demonstrates the trade-off between algorithm complexity and tree cost – Lorikeet's complexity is higher than that of the Source-Join algorithm, which only considers one parent node for a new receiver, but considerably lower than that of the exhaustive algorithms, which consider all possible parent nodes during a join operation. It is also interesting to note that Lorikeet with Path-Greedy joining is competitive with the Greedy algorithm (within 8%) despite its considerably lower complexity.

These figures show that Lorikeet is able to approach the performance in terms of total tree cost of algorithms that have much higher complexity and perform exhaustive searches of the tree on join operations. Due to this lower complexity, Lorikeet scales much better to large groups of receivers and ensures that joins to the tree (for which the user has to wait) are completed quickly. This makes Lorikeet far more appropriate for large-scale use.

5.5.3 Incremental Deployment

One of Lorikeet's design requirements, as described in Section 4.2.3, was that not all routers need know the protocol being used. In other words, the protocol should work even in a network that has only some capable routers present, or in a network with none at all. This property gives the system the ability to be *incrementally deployed*, a property that is effectively required to achieve broad usage across the Internet.

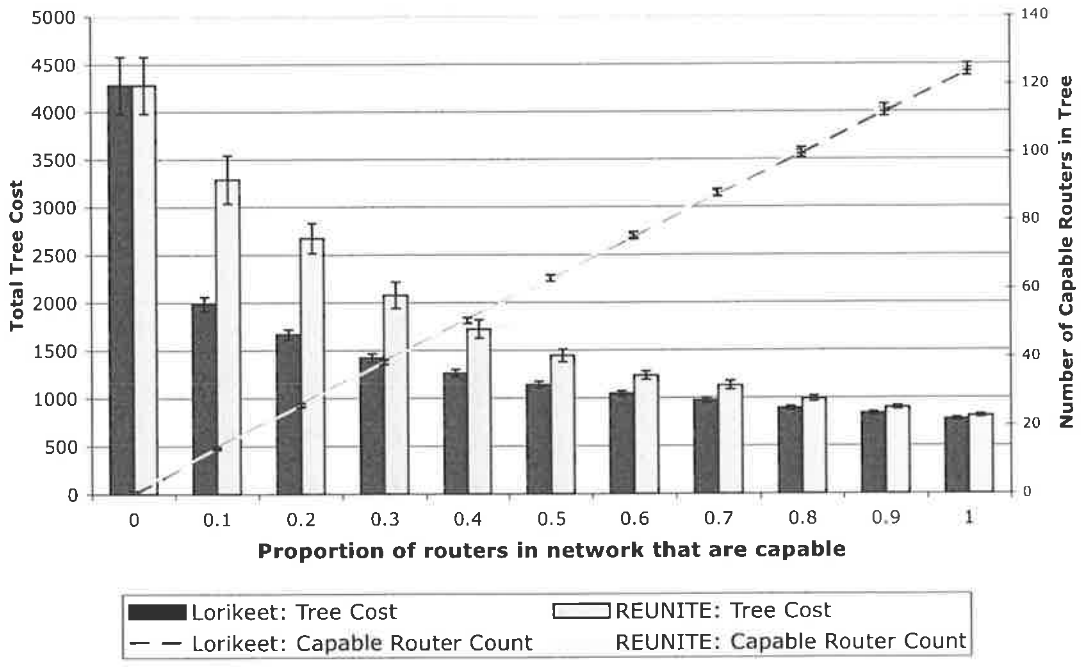


Figure 14: Comparison of incremental deployment performance for Lorikeet and REUNITE/HBH.

Figure 14 shows two sets of data from a simulation of the "Waxman" experiment described in Section 5.3.1 for both the Lorikeet and REUNITE/HBH protocols. The first (the column graph) shows the cost of the tree after 2000 events for a selection of values representing the proportion of capable routers in the network. The second (the line graph) shows the number of those capable routers being used in the tree after 2000 events.The simulation was run with ten different event sequences on five different Waxman topologies and the results averaged to generate the graphs shown. The error bars on both graphs represent 95% confidence intervals.

As the graph shows, REUNITE/HBH and Lorikeet generate distribution trees of exactly the same total cost when no capable routers are present; in this situation, all receivers are connected to the source by direct unicast connections. With the introduction of capable routers in the network, however, Lorikeet makes much better use of them, outperforming REUNITE/HBH by more than 30% when routers are sparsely distributed. Even at higher penetrations, Lorikeet constructs significantly cheaper trees.

For Lorikeet, the cost of the multicast tree when no routers are capable is more than double the cost of the tree when 10% of the routers in the network are capable. The efficiency of the total system clearly increases with capable router deployment, but the graph suggests that even a small amount of capability in the network makes a large improvement over the use of simultaneous unicast (which is exactly the same as Lorikeet with no capable routers). Approximately 75% of the improvement possible at full deployment is achieved when only 20% of the network's routers are capable, in this simulation.

The graph of the number of capable routers in the tree is linear, representing about 85% of the capable routers in the network at each point. This is unsurprising as capable routers, once discovered, remain in the tree while they still have child nodes to support, and there is a significant population of receivers in the tree by the end of the simulation. Some routers, of course, do not get discovered or do not remain long in the tree, as they are on the edge of the network on rarely used paths. The count of capable routers in the multicast tree is very similar for both algorithms. This is due to the fact that both Lorikeet and REUNITE/HBH discover capable routers in the same way, along the shortest paths between the source and receivers.

In a REUNITE/HBH multicast tree, the paths between nodes that are used are *always* sub-paths of the unicast shortest paths between the receivers and the source. Since Lorikeet places no such restriction on the path used by a receiver to join the tree, additional branching may be enabled. For example, a receiver may choose to join the tree through a router off the shortest path to the source whose path from the source is slightly longer than the receiver's direct path, but whose path to the receiver is very short. This ability to select from a wider range of potential parents allows Lorikeet to construct more efficient (in terms of total tree cost) multicast trees.

5.6 Summary

In this chapter, we presented a series of results and simulations designed to quantify the performance of Lorikeet's different components and compare that performance to other multicast algorithms. Since our focus in this work is on tree construction and maintenance, a topological simulation was designed to facilitate analysis of Lorikeet (and other systems) in terms of the trees it constructs.

To begin with, we described several competing multicast algorithms, namely ARIES, DSG and REUNITE/HBH. Two simple algorithms (the Source-Join and Greedy algorithms) were also included as baselines for comparison. The message complexity of these different approaches was discussed, showing both analytically and with empirical measurements the difference between algorithms that exhaustively search the tree (using Greedy joins) and algorithms that perform more limited searches, like Lorikeet.

The performance of Lorikeet's different join and rearrangement algorithms was examined through simulation. From our results, we concluded that the Path-Greedy join with Rejoin rearrangement was the most effective variant of Lorikeet for general purpose use, and we selected that version of the algorithm for further comparison. Next, we compared Lorikeet to the other algorithms discussed earlier, simulating all of the different systems on a network in which all the routers supported the protocols being used. This comparison showed that Lorikeet performs well in comparison to other algorithms, improving on REUNITE/HBH and the naive Source-Join algorithm and approaching the performance of the algorithms based on exhaustive join techniques, which are not practically deployable for large-scale use due to their message complexity.

Finally, we examined the performance of the only algorithms in our comparisons above that support use with only partial deployment in the network, Lorikeet and REUNITE/HBH. These results showed that Lorikeet makes much more effective use of sparsely-distributed capable routers, outperforming REUNITE/HBH by more than 30% in situations with limited deployment. Lorikeet's advantage was reduced at higher levels of deployment, but it was still able to generate cheaper trees in all cases.

The following chapter will describe our attempt to improve Lorikeet's performance further in networks where capable routers are sparsely distributed, through the use of a directory service for locating capable routers.

Chapter 6

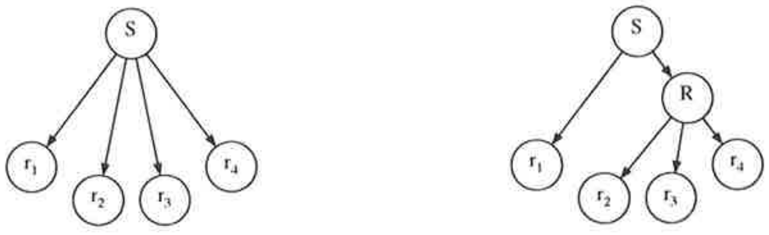
Directory Nodes

6.1 Motivation

To efficiently distribute the data being transmitted by the multicast source, a Lorikeet multicast tree requires capable routers (routers that support the Lorikeet protocol) to be present in the tree. These capable routers facilitate *branching*, or copying of data packets to more than one downstream node. Capable routers are discovered when new receivers join the tree, by searching the underlying shortest path between the new receiver and its point of connection to the rest of the tree.

If capable routers are scarce or not present on these shortest paths, then the multicast tree ends up as effectively a "simultaneous unicast" scenario, with every receiver receiving a separate unicast stream from the source. A simple diagram illustrating this situation is shown in Figure 15.

In Figure 15(a), no capable routers are present and the source must therefore connect to all of the receivers directly. Consequently, the source maintains four connections, one for each receiver, and sends the same data to each. In Figure 15(b), a single capable router R has been discovered and is being used to support three of the receivers. Therefore, the source need only transmit two copies of the data, one to the first receiver (which is still directly connected to the source) and the other to the router R, which copies the data to its three child receivers. Such branching reduces the load on the network and allows the network to support many more receivers without overloading the source.



(a) No capable routers in the tree; every receiver has its own connection from S.

(b) R is a capable router and shares the link (S, R) between its children, reducing network load.

Figure 15: Capable Routers in the tree improve efficiency through link sharing.

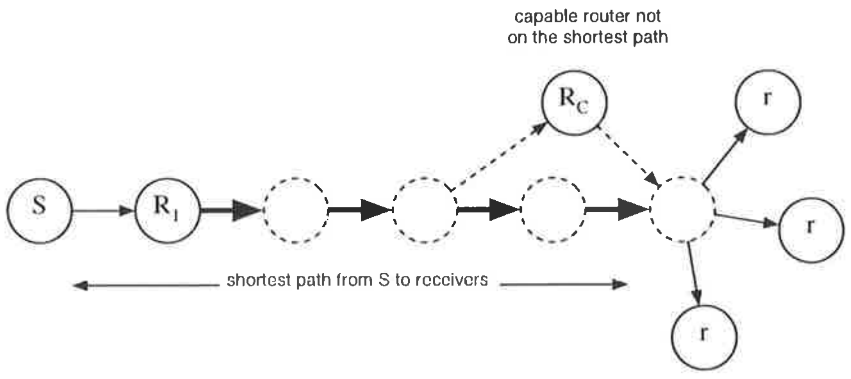
The Lorikeet protocol, as described in earlier chapters, makes an important assumption: that capable routers are present on the shortest paths between parent routers and receivers, where they can be discovered and used for branching. However, it is possible that capable routers may be present in the network but *not* on the shortest paths to receivers; in this situation it is desirable to *discover* them through some other mechanism and add them to the tree. Some example scenarios in which these conditions may arise include:

- multicast trees where the source has just started transmitting and the tree contains no capable routers at all;
- networks where capable routers are few in number and sparsely distributed;
- networks where routers on the shortest path are large backbone routers that may not be capable or do not want to "sniff" for Lorikeet join packets for performance reasons. Hence, any capable routers present will not be on the shortest path. A simple diagram showing this situation is presented in Figure 16.

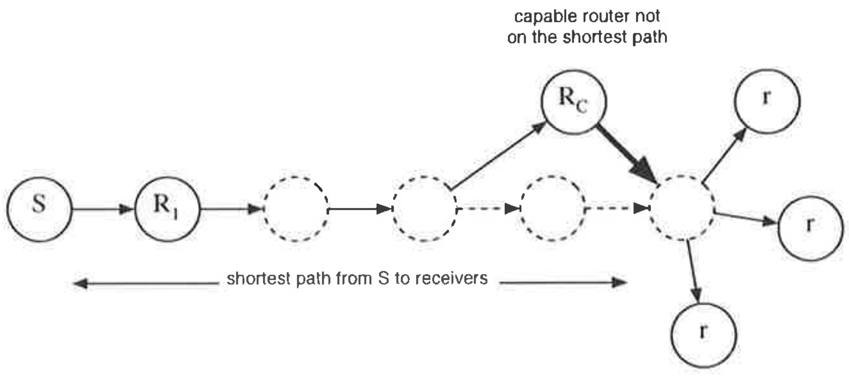
Once a capable router is added to the tree to support a receiver, however, it can be reused by other receivers that join the tree after its addition. Such reuse may lead to a tree that is of lower cost than it would have been had that capable router not been added, thus justifying the capable router's addition – even if its use is initially more expensive than directly connecting the first receiver to the source. In these situations, an alternative method must be used to find capable routers in the network that are not present on the shortest paths to receivers, but are close enough in network terms to reduce the cost of the tree through increased branching.

For example: in the situation shown in Figure 16(a) the multicast stream is being transmitted from the source to a capable router R_1. Router R_1 is then transmitting the stream to three receivers via shortest paths that are largely the same but contain no further capable routers. Hence, the path through the non-capable routers (shown as dashed nodes) is not being shared, but is instead carrying three copies of the stream, one for each receiver. Those links carrying multiple copies of the stream are shown in bold. Capable router R_C is close enough to the three receivers to provide a net decrease in the cost of the tree if it were used, by reducing the burden of the path to R_C to a single copy of the stream.. This scenario is illustrated in Figure 16(b), and it can be seen that the number of links carrying multiple copies of the stream is reduced to just one. Since R_C is not on the shortest path to any of the three receivers, however, it will not be discovered by Lorikeet's join mechanism and will remain unused.

It is important to note that multicast transmission using the Lorikeet protocol should *not* be dependent on any additional discovery procedure to operate – such a system should only provide a way to improve performance in some scenarios. As discussed in earlier chapters, Lorikeet is designed to operate with very few (or even no) capable routers in the network, in order to permit incremental deployment of the protocol. We feel that this support for incremental deployment is a strong requirement for new Internet protocols designed for wide use.


capable router not on the shortest path
R_C
S
R_1
r
r
r
shortest path from S to receivers

(a) Capable router R_C is present, but not on the shortest paths to receivers.


capable router not on the shortest path
R_C
S
R_1
r
r
r
shortest path from S to receivers

(b) If R_C is discovered and used for branching, the number of links carrying duplicate streams is reduced.

Figure 16: Capable routers may be not on the shortest paths to receivers, but can still reduce the cost of the tree if used.

6.2 Directory Nodes

The solution proposed for the problem of finding capable routers that are off the shortest path is the concept of a *directory node*. A directory node is a node within the network that can be queried by Lorikeet sources and routers. When it is queried, it returns a set of capable routers from a database that it maintains – thereby providing a way for the algorithm to introduce more capable routers (and hence more potential for branching) into the multicast tree.

Such directory services for discovering the addresses of hosts providing desired information are used in many other Internet protocols, the most obvious example of which is the use of the Domain Name Service (DNS) to support name resolution and email delivery. Many peer-to-peer protocols also use directory services to discover the locations of other nodes or data within the network overlay, such as the trackers used by the BitTorrent [10] protocol and the GWebCache [34] software used by some clients to bootstrap a connection to another node in the Gnutella [31] network. These applications rely on directory techniques to operate.

In our proposal for Lorikeet, directory nodes will exist at well-known addresses within the network (perhaps defined in a specific DNS zone, for example) and can share information with each other about known capable routers. Capable routers should be administratively configured to periodically notify a directory node about their location, information which can then be propagated to other directory nodes and used in responses to querying routers.

When a join operation is underway, a router may query a directory node for a list of *"adopted children"* (additional capable routers) to consider in addition to its own children in the tree. Paths through these adopted children will, by definition, have an equal or higher network cost than the shortest path to the receiver. In order to bias the system towards the addition of new capable routers, the cost of a path through an adopted child is reduced, or "sweetened", for the purposes of comparison in order to make it more likely to be selected. Using a sweetened path is a decision to use a higher-cost path through a new capable router in the short term, with a view to recovering the cost difference by sharing it in subsequent operations.

We have not addressed the system by which directory nodes would maintain or share with each other the information that they store – such topics have been addressed in the literature, particularly in the design of peer-to-peer network protocols, such as CAN [63] and Tapestry [82], and content replication systems such as those described by Kangasharju *et al.* [44], and used by the commercial firm Akamai [2] and the research project Coral [17, 29].

The focus of our interest in this area is the improvement of Lorikeet's performance, with the service provided by directory nodes being a logical approach to achieving that goal.

6.3 Joining the Tree

Directory nodes are used to enhance Lorikeet's existing join operation, described in Section 4.5.2. They are designed to provide additional potential parent routers for consideration during the join. As the join operation proceeds, routers in the tree (including the source) can query a directory node and ask for a list of capable routers (adopted children) to be considered for inclusion in the tree.

When a list of capable routers is returned to a querying router by the directory node, these adopted children are queried to ensure that they are not already participants in the tree (which could otherwise cause the formation of loops). Adopted children that are not yet participants in the tree are then used as additional children in the join's path calculation. The cost of using one of these adopted children is the marginal cost: the cost of adding the new capable router to the tree, plus the cost of the link between that router and the new receiver. In order to bias the system towards the inclusion of new capable routers, since the cost can be no less than that of the shortest path, we decided to use a "sweetener" to reduce the cost of paths containing adopted routers, dividing the marginal cost by a factor α greater than one.

It is important to note that this approach requires the cost metric used by Lorikeet for selecting links be *additive*, since this addition to the algorithm relies on being able to add two partial path costs together and use the result for comparison.

Although initially we applied the sweetener to the entire marginal cost, we soon found that a more appropriate way to apply the sweetener was on a link-by-link basis, rather than to the whole marginal cost. Therefore, only those links for which the destination node is a capable router not currently in the tree have their cost divided by the sweetener value, while other links in the path remain at full cost. The motivation behind this approach is to only bias the system towards the inclusion of nodes that can be reused (namely, new capable routers) and not the cost of the link to the new receiver, which is a link that cannot be shared with other receivers. A more detailed discussion of path sweetening is given in Section 6.3.2.

6.3.1 Algorithm

In this section, we present a description and example of our mechanism for the use of directory nodes in Lorikeet's join operation.

This algorithm builds on the Path-Greedy join technique described in Section 4.5.2. At every step of the search through the tree for a parent, each selected router is able to query a directory node for adopted children. The marginal costs of these adopted children are calculated, and the cheapest of them is stored in the join message that is passed down the tree as the tree path is calculated by the Path-Greedy join algorithm. Adopted children are not used in the tree path itself, as selecting an adopted child would mean terminating the search early (since adopted children have no children themselves).

When the Path-Greedy search terminates, having found the tree path, the cheapest router in the tree path is compared to the cheapest adopted child, and the option with the smallest cost is chosen to be the parent of the new receiver. If the tree path router is chosen, the system behaves as the regular Path-Greedy algorithm does. If the adopted child router is chosen, the final path to the new receiver will consist of the path through the tree to the parent of the adopted router, then the adopted router, and then the new receiver itself.

Pseudocode for the algorithm is shown below. The following state variables are passed down the tree with the join message, and their initial values are as shown. The sweetener value is denoted by α.

```
$P :=$ [] {the tree path, initially an empty list}
```

```
$R_{T_{best}} := S$ {the current best candidate parent router in the tree}
$C_{T_{best}} :=$ **cost**$(S, r)$ {the cost of the path from $R_{T_{best}}$ to the new receiver $r$}
$R_{A_{best}} :=$ None {the best adopted child router}
$C_{A_{best}} := \infty$ {the marginal cost of using $R_{A_{best}}$}
$Parent_{A_{best}} :=$ None {the tree router which is the parent of $R_{A_{best}}$}
$R_c := S$ {the router on which the processing is currently taking place}
**loop**
  {this processing occurs on the current router, $R_c$}
  $R_c$ calculates **cost**$(R_c, r)$.
  append $R_c$ and its cost to the tree path $P$.
  **for** each child $R_k$ of $R_c$ **do**
    $R_c$ contacts $R_k$ and instructs it to calculate **cost**$(R_k, r)$.
    $R_k$ sends its calculated cost to $R_c$.
  **end for**
  $R_{child} :=$ child $R_k$ with minimum **cost**$(R_k, r)$.
  $R_c$ requests adopted children for receiver $r$ from a directory node.
  **for** each adopted child $R_a$ **do**
    $R_c$ queries $R_a$ and discards $R_a$ if it is already a participant in this multicast
    tree.
    $R_a$ calculates marginal cost: **marginal**$(R_a) := \frac{1}{\alpha}$**cost**$(R_c, R_a) +$ **cost**$(R_a, r)$
    $R_a$ sends its calculated cost to $R_c$.
  **end for**
  $R_{adopted} :=$ adopted child $R_a$ with minimum **marginal**$(R_a)$.
  **if marginal**$(R_{adopted}) < C_{A_{best}}$ **then**
    {there is a new best candidate adopted router}
    $R_{A_{best}} := R_{adopted}$
    $C_{A_{best}} :=$ **marginal**$(R_{adopted})$
    $Parent_{A_{best}} := R_c$
  **end if**
```

```
    if $R_c$ has tree children then
      if cost$(R_{child}, r) < C_{T_{best}}$ then
        {there is a new best candidate tree router}
        $R_{T_{best}} := R_{child}$
        $C_{T_{best}} :=$ cost$(R_{child}, r)$
      end if
      $R_c := R_{child}$ {pass control to the minimum cost child}
    else
      {there are no further tree children; now we choose the parent for $r$}
      if $C_{A_{best}} \leq C_{T_{best}}$ then
        {the best parent is the best candidate adopted router}
        return (subset of $P$ from $S$ to $Parent_{A_{best}}$)+$[R_{A_{best}}, r]$
      else
        {the best parent is the best candidate tree router}
        return (subset of $P$ from $S$ to $R_{T_{best}}$)+$[r]$
      end if
    end if
  end loop
```

Using directory nodes in this way adds only a very small amount of extra state information to the join message used by the Path-Greedy Join. In addition to the data required by the Path-Greedy algorithm already, the join message must carry $R_{A_{best}}$, the location of the best adopted child, $C_{A_{best}}$, the marginal cost of that adopted child, and $Parent_{A_{best}}$, the location of the router that "found" the best adopted child.

An example of this procedure operating is shown in Figure 17. The node marked r is the new receiver, S is the source of the multicast tree and $R_1, \ldots, R_6$ are capable routers that are already in the tree. The shaded nodes are nodes that were asked to calculate their cost to the new receiver, while the unshaded nodes were completely uninvolved in the tree search.

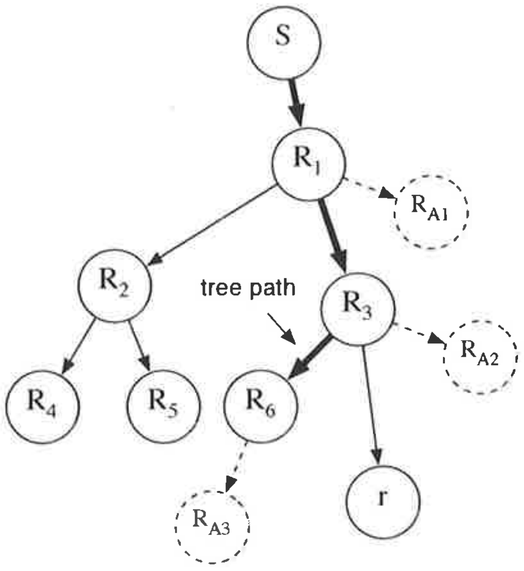


Figure 17: Path-Greedy Join with Directory Nodes

The tree path after performing the first part of the join operation (searching the tree) is (S, R_1, R_3, R_6), as shown in bold. As the tree path was constructed, each router contacted a directory node and queried for adopted children. R_{A1}, R_{A2} and R_{A3} are the best adopted children returned by directory nodes to queries from R_1, R_3 and R_6 respectively. S did not receive any adopted children in this case.

As the join traverses the tree path, the marginal costs of the adopted children are calculated and used to determine the best adopted child. The marginal costs for the three in our example are as follows:

$$\begin{aligned}
\mathbf{marginal}(R_{A1}) &= \frac{1}{\alpha}\mathbf{cost}(R_1, R_{A1}) + \mathbf{cost}(R_{A1}, r) \\
\mathbf{marginal}(R_{A2}) &= \frac{1}{\alpha}\mathbf{cost}(R_3, R_{A2}) + \mathbf{cost}(R_{A2}, r) \\
\mathbf{marginal}(R_{A3}) &= \frac{1}{\alpha}\mathbf{cost}(R_6, R_{A3}) + \mathbf{cost}(R_{A3}, r)
\end{aligned}$$

These values are compared as the tree path is traversed, and the adopted child with the minimum marginal cost is chosen as $R_{A_{best}}$. If we assume that in this case the minimum cost adopted child is R_{A2} and the minimum cost tree router is R_3, then the final part of the join operation proceeds as follows:

When the tree path calculation reaches the final router R_6, which has no further tree children, a comparison is made between the two possible cases; either tree router R_3 becomes the parent of r, or adopted child router R_{A2} does. If the path from R_3 has the lowest cost, the path to the new receiver will become (S, R_1, R_3, r). If the adopted router R_{A2} has the lowest cost, it will be added to the tree as a child of R_3 and the final path to the new receiver will become (S, R_1, R_3, R_{A2}, r).

6.3.2 Discussion

In our application of directory nodes to the Path-Greedy join algorithm above, we defer the use of adopted children until after the end of the tree search, similarly to how we traverse the complete tree path before deciding on a router to be the new receiver's parent. Initially, we did investigate the immediate use of an adopted router if its cost is the minimum cost found during a step in the search process. This approach, however, created unexpected problems. Since an adopted child is a router that was not present in the tree before it was discovered, it has no tree children. This results in the join procedure terminating when one of these routers is selected in a round of the join operation. If the sweetener value is high and/or a large number of adopted children are used, it becomes very likely that an adopted child will be selected, rather than a child from the existing tree. This has the effect of terminating the join operation quickly, when a particularly cheap adopted child is encountered early in the search. Consequently, this reduces the number of nodes considered in each search and over time results in a higher cost tree. The effects can be illustrated topologically as well: if all join operations are terminated early then the resultant tree is short and fat, consisting of a large number of receivers connected to the tree by short paths through adopted children. Such a tree will have less branching (and hence be less efficient) than a tree generated by a join process that exploits routers already in the tree more often.

To address this issue we designed the technique described here, where the use of adopted routers is deferred until the complete tree path has been calculated. This approach allows the existing tree to be exploited as much as possible before adopted children are considered and biases their use towards the bottom of the tree, rather than the top. Adding additional nodes at the bottom of the tree ensures that existing branching in the tree is leveraged, rather than simply adding additional branches from the source or other routers high up in the tree.

The Cost of an Adopted Child

In the previous section, we briefly described the way in which the marginal cost of a path involving an adopted child is sweetened in order to promote the addition of more capable routers (and therefore, more potential for branching) to the tree.

Early in our work on directory nodes, we applied the sweetener to the entire path through the adopted router, sweetening both the link between the adopted router and its parent *and* the link between the adopted router and the new receiver. Later, we refined this approach, sweetening only those links that end in new capable routers, while other links (such as the one between the adopted router and the new receiver) were considered at "full cost". Note that a "link" in this description is a link in the *tree*, which may consist of a path through a number of non-capable routers, rather than a link in the underlying physical network.

A simple example illustrating the difference between the two approaches to path sweetening is shown in Figure 18.

In this example, the router R is a router in the tree, while routers R_{D1} and R_{D2} are adopted child routers. We let the sweetener be 2 and the link costs be as shown.

When new receiver r joins the tree, the whole-path approach treats both paths as identical, with the same sweetened path cost of $(10 + 1)/2 = 5.5$. However, the

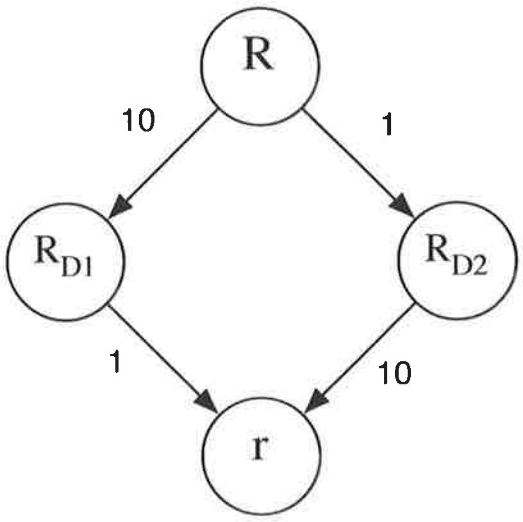


Figure 18: Two Approaches to sweetening the path

links (R, R_{D1}) and (R, R_{D2}) are reuseable, whereas the links (R_{D1}, r) and (R_{D2}, r) are not. It is possible to recover the cost of the reusable links through later join operations to those routers. This is what our later approach does: in this example, it gives the left-hand path a sweetened cost of $10/2 + 1 = 6$ and the right-hand path a sweetened cost of $1/2 + 10 = 10.5$, thereby selecting the left-hand path as cheaper.

The rationale for our selected approach to sweetening paths is that it draws a distinction between paths through the network that can be reused in the future (links to capable routers) and paths that offer no possibility of reuse (links to receivers). Therefore, given a choice of different paths to a receiver with similar costs, it is logical to choose the one with the least cost final link, since that link cannot be reused in subsequent joins.

The optimal value of the sweetener itself is very dependent on the topology of the network and the properties of the application. The selection of the sweetener is based on the probability that that router will be used by enough receivers joining later to make its addition to the tree cheaper than connecting those receivers elsewhere. This probability is affected by the proximity of receivers to each other and to capable routers in the network and is thus difficult to determine *a priori* without extensive knowledge of the network and the popularity of the multicast transmission.

An analysis of the relationship between the selection of the sweetener value and its effect on the total cost of the tree is given below.

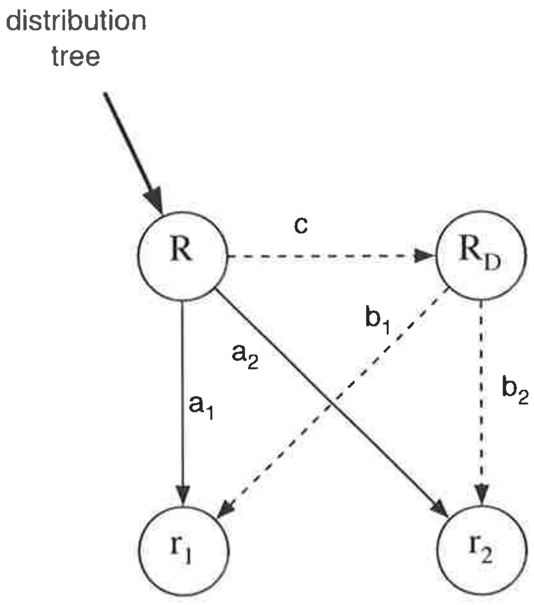


Figure 19: Selecting a sweetener

In the diagram shown in Figure 19, we have a Lorikeet multicast tree containing a capable router R with no children. That router is selected as the final tree router in the join algorithm for new receiver r_1, and has also received adopted child R_D from a directory node. The costs on the paths between these nodes are as shown.

For a sweetener value α, the cost of connecting r_1 to the tree via a direct connection to R is a_1, while the (sweetened) cost for connecting it through R_D is $\frac{c}{\alpha} + b_1$.

Hence, R_D will be added to the tree if:

$$\frac{c}{\alpha} + b_1 < a_1 \tag{6.1}$$

Assuming that the above inequality holds and R_D is added to the tree, we now add receiver r_2 in a subsequent join operation where both R and R_D are candidate parent nodes. R_D will become the parent of r_2 if $b_2 < a_2$.

Since Equation 6.1 allows the longer path through R_D to be chosen even if its actual contribution to the cost of the tree is greater than that of the direct path ($c + b_1 > a_1$), the cost of adding R_D must be recovered through subsequent joins to R_D that would have otherwise joined elsewhere on the tree. If we take the value a_i to represent the cost of joining the receiver r_i to the tree by a path other than that through R_D, and the value b_i to represent the cost of joining the receiver r_i by the path through R_D, then the cost of adding R_D will be recovered when:

$$c + \sum_{i=1}^{n} b_i \leq \sum_{i=1}^{n} a_i, \tag{6.2}$$

for n receivers r_1 to r_n. Rearranging for c yields,

$$c \leq \sum_{i=1}^{n} (a_i - b_i). \tag{6.3}$$

As is plain from the diagram, this tells us that the cost of adding R_D, the link with cost c, will be recovered when we have added enough new receivers r_i for which the sum of the differences in their costs exceeds c.

It is difficult to make decisions based on the values of a_i and b_i, since we do not know *a priori* the locations or path costs of new receivers. Let k represent the average cost recovered per new receiver by joining n receivers through capable router R_D:

$$k = \frac{\sum_{i=1}^{n} (a_i - b_i)}{n} \tag{6.4}$$

Then the minimum number of receivers required to recover the cost of adding R_D is given by n, which (by Equation 6.1) is equivalent to the smallest integer greater than or equal to $\frac{c}{k}$, denoted $\lceil \frac{c}{k} \rceil - 1$.

Hence, $\lceil \frac{c}{k} \rceil - 1$ is the number of additional receivers that must join the tree once r_1 has introduced the adopted child router R_D in order to recover the cost of adding

R_D.

6.4 Results

The extended Path-Greedy join algorithm described in Section 6.3 was implemented in the Lorikeet simulation environment. In order to analyse its performance, we created simulations on a Waxman topology (see Section 5.3.1) with sequences of 2000 events. These simulations are configured so that capable routers can either be leaf nodes (nodes with degree one) or non-leaf nodes (nodes with degree greater than one), and the proportions of capable leaf nodes and capable non-leaf nodes can be controlled independently. Tree routers in these simulations are configured to ask a directory node for *all* of the capable routers present in the network and then consider them as adopted children. This exhaustive behaviour was chosen in order to show the best possible performance improvement through the use of directory nodes. In a real implementation, of course, query results would need to be limited to a much smaller set of adopted children to avoid too large an increase in the message complexity of the join.

Simulations were performed over a large range of different levels of capable router penetration, with sweetener values varying from 1.0 through to 5.0. Figure 20 presents the results of three of these simulations, as follows:

1. A Waxman topology with no capable non-leaf routers and 10% of leaf nodes as capable routers;
2. A Waxman topology with 10% of capable non-leaf routers and 10% of leaf nodes as capable routers;
3. A Waxman topology with 50% of capable non-leaf routers and 50% of leaf nodes as capable routers.

The figure shows a graph of the mean tree cost over the final 1000 events against the sweetener value. Each of the three simulations was run 25 times, using different sequences of events on the same topology, and the results shown are the means calculated over all 25 runs.

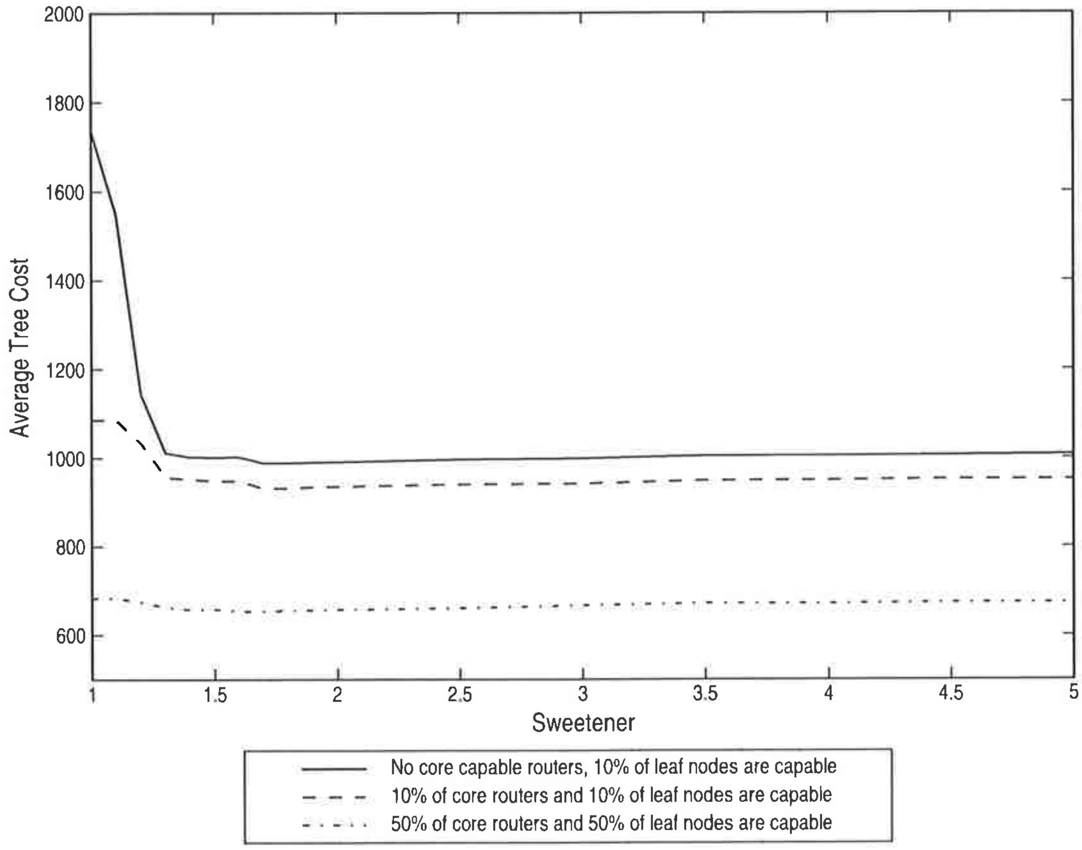


Figure 20: Average tree cost against sweetener value for Directory Nodes simulations

The three situations simulated are very different. In our first scenario, all of the capable routers present are configured as leaf nodes. Hence, they will not be discovered on the shortest paths from the source (or other routers) to receivers; instead, Lorikeet must rely on the use of directory nodes to discover new routers. In this case, the only capable routers available are a randomly chosen 10% of the leaf nodes in the network, with no capable routers at all in the middle of the network. Our second scenario shows the result of introducing an equal proportion of capable routers into the middle of the network, where they can be directly discovered on the paths between routers and receivers. In the third case, we simulate a situation where a large number of capable routers are present, comprising 50% of non-leaf routers and 50% of leaf nodes.

Consider the first case, where the only capable routers present are leaf nodes. With a sweetener of 1.0, no paths through capable routers are being discounted. Since all of our capable routers are leaf nodes in the network, none of them are on the shortest paths between the source and the receivers. Therefore, no capable routers returned by a directory node will achieve a lower cost than the direct shortest path, and the protocol will not discover any capable routers at all. However, as the sweetener is increased, the cost of the tree drops significantly. The total cost of the tree is reduced by approximately 42% at a sweetener of 1.3, despite the relatively small number of capable routers (only 10% of all leaf nodes) available. This illustrates how valuable even the presence of a small number of branching routers is, converting the system from simultaneous unicast to a branching multicast tree.

In the second case, we have added more capable routers to the network, making 10% of its non-leaf nodes into capable routers in addition to the 10% of leaf nodes that were capable in the first scenario. Here, the chart clearly shows the beneficial effects of having capable routers in the middle of the network, rather than limiting their availability to edge nodes alone. The average tree cost with a sweetener of 1.0 (where the join algorithm will not select new capable routers from directory nodes) is almost 40% lower than that of the first scenario. This is a measure of the efficiency improvements attributable to the placement of capable routers in the middle of the network, rather than at the edge. The use of directory nodes with sweeteners above 1.0 only provides a small (about 14% at best) improvement in tree cost.

This trend is further demonstrated by our third scenario, in which half of all non-leaf routers and half of all leaf nodes are made capable. Here, we can see that the unsweetened case is lower again, at 61% of the unsweetened tree cost of our first set of results. Increasing the sweetener (and therefore using directory nodes) has very little additional effect, providing at most a 4% reduction in tree cost.

Our first scenario demonstrates that capable routers at the edge of the network can significantly reduce Lorikeet's tree cost, providing that a mechanism like the use of directory nodes is available to permit their discovery. However, as shown by our second and third examples, the addition of capable routers to the middle of the network (even only in limited numbers, such as 10% of non-leaf nodes in the network) provides a similar performance improvement without the additional complexity. This is intuitive if the cost of using a capable leaf node is considered. A capable router in the middle of the network which is either on the shortest path or already in the tree is likely to be considerably cheaper to use than a leaf node that is further away from the direct path, where data transmitted through it must traverse its single access link twice.

6.5 Conclusions

The last section showed that the use of directory nodes is warranted in scenarios when capable routers are not present on the shortest paths between the source and receivers. Without the use of an alternative discovery mechanism (like the use of directory nodes), the behaviour of Lorikeet degrades to simple simultaneous unicast, with every receiver maintaining a unicast connection to the source. If even a small number of capable routers at the edge of the network can be used to provide branching, this is sufficient to reduce the total cost of the tree significantly.

However, the performance gains achieved through the use of capable routers at leaf nodes in conjunction with directory nodes can also be achieved by placing capable routers in the middle of the network, *without* the directory node mechanism. Furthermore, adding directory nodes to this scenario does not significantly improve performance further: branching in the middle of the network is much more efficient than branching at the edge, where the network costs to end receivers are higher. These results were obtained with an exhaustive search of the capable routers at leaf nodes; in a practical implementation, performance improvements would necessarily be reduced further. The simple case described in the introduction, with a capable router placed in the network adjacent to the path on which multicast traffic is flowing, can be addressed much more simply by having a conventional router on the path filter and forward probe packets using standard IP filtering. This approach is described in more detail in Section 7.2.2.

The increase in message complexity required by the addition of directory nodes to the protocol is not negligible – it potentially adds several extra messages to each stage of the recursive join algorithm and an extra traversal of the tree path. This could be minimised by intelligent selection of the adopted children returned by the directory nodes, caching of query results in tree routers and other improvements, but it still remains significant.

Lorikeet is envisaged for deployment in networks where there will be at least some capable routers present on paths traversed by multicast joins, whether they are in the middle of the network or on border routers, hosted by service providers nearer the edge of the network. In these situations, the use of directory nodes to enable discovery of other routers adds extra complexity to the protocol while delivering very limited performance improvements, as shown in the previous section. For these reasons, we feel that the use of directory nodes as a part of the core Lorikeet protocol for general use is not warranted.

Chapter 7

Implementation Concerns

Previous discussion in this dissertation has largely focussed on the topological aspects of multicast and Lorikeet's design from a tree construction and maintenance perspective. In the requirements given in Section 4.2.3, however, we stressed the necessity for Lorikeet to be a practical, as well as efficient, multicast protocol. In this chapter, we discuss some issues of implementation that arise from those requirements and from the hierarchical, unicast-based design we have proposed.

7.1 Accessing a Lorikeet stream

Lorikeet is designed to be a multicast transmission protocol for live streaming multimedia. It has no mechanism for locating available content on the Internet: we feel that this functionality is better implemented in out-of-band mechanisms, as it is for other transmission protocols like BitTorrent [10] and streaming protocols like those used in Microsoft's Windows Media Player [78] or Apple's Quicktime [62].

Since Lorikeet is a single-source multicast application, the address of the source and an identifier to distinguish different streams originating at that source provides enough information to identify and connect to a multicast group. To that end, we propose the use of a Uniform Resource Identifier (URI) [73] mechanism to identify Lorikeet streams, with the following syntax:

```
lkt://host:port/path
```

where *host* is the address of the source that is sending the Lorikeet stream, *port* is the port number being used by the server on that host to accept receiver connections, and *path* is a path that identifies the stream, distinguishing it from others being distributed by the same source.

Using this scheme for describing a Lorikeet multicast stream allows easy description and referencing of Lorikeet resources on the World Wide Web as is currently done with many other protocols, including email, telnet, FTP, Windows Media streams ("mms" streams), Quicktime streams, etc. Existing Web browsers already have mechanisms for defining *handler applications* and *plugins* to handle protocols that are not supported natively by the browser. This functionality could be used to display streams inline in a Web page or hand over Lorikeet URIs to a video playing application that supports the Lorikeet protocol.

The Lorikeet protocol itself can be decomposed into two parts: *control* and *transmission*. Control messages are transmitted over TCP sessions established between communicating nodes in the multicast tree: for example, between the source and a child router, a parent router and a child router, or between a router and a receiver. A TCP connection is also used for the initial part of a join operation, when a new receiver contacts the source. These sessions are used for messages related to construction and maintenance of the tree, such as the searches for a parent that occur when a new receiver joins and the notification of a parent router during a leave operation.

Separately, Lorikeet uses UDP for transmission of the data stream itself through the tree. Each stream being transmitted by a source (since a source can manage several groups) can be identified by the source's IP address and a unique UDP port number used by the source for that stream. This (IP, port) tuple can be used to uniquely identify a stream within a capable router for forwarding purposes, as described in the next section. The source sends UDP packets containing the data to its direct children only. Those child nodes that are routers then rewrite the headers of these packets and forward copies to their children and so on, until the packets have been transmitted to every receiver connected to the tree.

7.2 Implementing Lorikeet

In this section, we present a discussion of the necessary requirements for a physical implementation of the Lorikeet protocol on all three types of participant node: receivers, capable routers and sources.

7.2.1 Lorikeet Receivers

In many ways, the client software used on end-users' computers to receive information from a Lorikeet multicast tree is the simplest of the three components described in this section. It does not need to do a great deal of control, since it supports no child nodes, and only needs to connect to the tree and begin receiving the data stream.

The client receiver software's operation is as follows:

1. Using a Lorikeet URI (as described earlier), identify and contact the source. Request the stream described in the URI. The source will return the (source, port) tuple identifying the group.

2. Wait for the source to provide a parent router R_P to connect to.

3. Connect to R_P, identifying the desired multicast group with the (source, port) tuple, and begin receiving data.

4. When the user decides to leave the tree, notify R_P that the receiver is leaving and disconnect.

The codecs (compression/decompression algorithms) necessary to play the stream back to the user will be application-dependent, and can be negotiated with the source in step 1 above. Most modern operating systems provide access to a variety of different codecs, and there are many third-party libraries available that could be used to provide this functionality.

A Lorikeet client could be built as a stand-alone application or Web browser plugin, as is done with many existing multimedia playback software packages [78, 62]. Playback software could also be built for set-top box devices, for streaming video to traditional television screens.

7.2.2 Lorikeet Capable Routers

The capable routers in Lorikeet provide branching to the multicast tree. Each router receives a single copy of the data stream from its parent and retransmits a copy of that data to each of its children. In addition to this, capable routers must handle control operations: adding new children, removing leaving children, participating in new receiver joins and performing rearrangements.

We envisage two different types of capable router implementation: a software implementation on a network-connected server and a router implementation on a core or border router. Which of these implementations is used would depend on the requirements of the organisation deploying the capable router and the properties of the network in which it is to be used. We describe both approaches in the following sections.

Software Implementation

A software implementation of a Lorikeet capable router would be hosted on a network-connected server running a general purpose operating system such as Linux. All functions would be implemented by a user-space server application, in much the same way that a World Wide Web server is usually implemented. It would listen on a TCP port for control messages and use UDP for transmission of the stream itself.

Control operations on the router are facilitated by maintaining TCP connections with the router's upstream parent in the tree (another capable router or the source itself) and each of its children, which could be either capable routers or receivers. Messages received on these connections are either passed on further up (or down) the tree, or result in changes to the parent or receiver lists maintained by the router so that packet forwarding can take place. When a request to participate in a join arrives from the upstream router, the measurement of the cost to the new receiver is done and passed upstream to be used in calculation. If the router is the cheapest option at this stage of the calculation, it performs its stage by passing the join message to its child routers, selecting the cheapest child from the results, and passing control of the join to that router. If a request to add a child node arrives, a TCP connection is established to that node and the router adds its address to the list of children to which it transmits copies of the stream. Leave operations are handled similarly, with the router receiving a leave message over the control connection from the departing child. In response, it removes that child from the list of children receiving the stream and disconnects its control connection. Rearrangement requests are received from downstream or upstream routers in the tree, depending on the direction of the router triggering the rearrangement. If the current router is a non-branching router, it passes the request on to the next router. A branching router must perform the rearrangement as described in Section 4.5.4.

In order to participate in multicast branching, this capable router needs to be discoverable on the paths traversed by messages between the source and new receivers, as described in Section 4.5.2: Capable routers on these paths sense a probe packet sent along the unicast path from a parent router to a receiver, announce their existence to the parent router and join the multicast tree. These probe packets are sent on a defined, well-known port number assigned to Lorikeet, and are therefore easily identifiable. An example of this behaviour is shown in Figure 21. In practice, however, it is very unlikely that a software router like the one proposed in this section could be placed on a critical path in the network, since it is unlikely to be able to forward packets as efficiently as a standard router can.

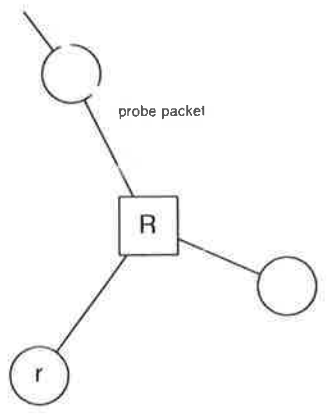
In this situation, a different approach is needed. We suggest a similar approach to that employed in "transparent proxying" schemes, where World Wide Web requests passing through a border router are redirected to a proxy server. That proxy server can then service the request using a cache if possible before contacting the actual server, in order to reduce latency and save bandwidth. Using this approach, the capable router need not be on the shortest path in a network: instead, a hardware router on the initial path identifies these probe packets (based on the use of a standard UDP port number used for Lorikeet path probes[1], for example) and forwards them on to a capable router. That router is then able to contact the parent router from which the probe originated, announce itself and join the tree. No further packet forwarding from the intercepting router is required, since the capable router is able to participate in the tree on its own after the initial contact is made. This behaviour is illustrated in Figure 22.

Such filtering based on port number is functionality that is already available and in heavy use for many other applications in currently deployed routing infrastructure, particularly for firewall security in border routers. This technique frees capable routers from the necessity of deployment on ingress and egress points in the network, allowing them to be placed at will as long as appropriate redirection filters are in place. Load-balancing between capable routers could also be achieved trivially with this technique, by redirecting packets to a pool of capable routers.

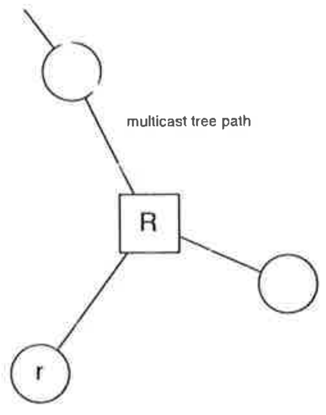
Router Implementation

Alternatively, the Lorikeet protocol could be implemented on a commercial router, designed to be deployed directly in the network with no additional support. For this to be possible, Lorikeet's copying and delivery of packets must be realistically capable of implementation on line cards, most likely in hardware as is done currently

[1]Note that this port would be a standard, defined UDP port used specifically for new receiver probes. It is a different port from the one allocated to the stream by the source, as described in Section 7.1, which could be randomly selected.

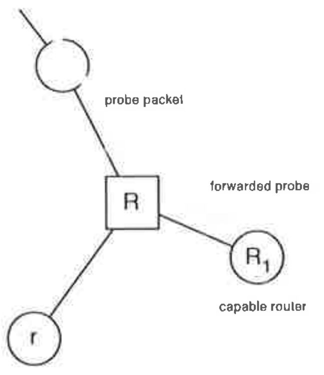


(a) A probe packet arrives at capable router R on the path to receiver r.

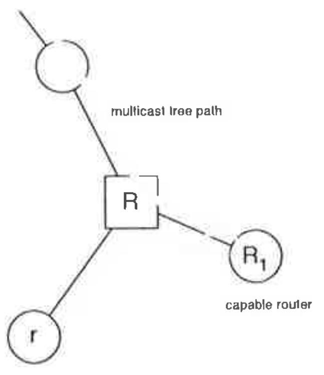


(b) Router R joins the tree and becomes the parent of receiver r.

Figure 21: Discovering capable routers that are on the shortest path to a new receiver.



(a) A probe packet arrives at (non-capable) router R. R forwards the probe to a nearby capable router, R_1.



(b) Capable router R_1 joins the tree and becomes the parent of receiver r.

Figure 22: Discovering a capable router that is not on the shortest path to a new receiver, but has probe packets forwarded to it by a filtering rule on a border router.

with TCP/IP forwarding.

We feel that such an implementation is possible. When a Lorikeet data packet arrives from the current router's parent (either the source or another capable router), it must be copied, and a version of the packet sent to all of the current router's children. For each copied packet, the source address and port are rewritten to match the current router's address and port, and the destination address and port are rewritten to match the child's address and port. This can be done without manipulating any part of the packet other than the IP header, which is fixed-length and can be operated on very quickly: indeed, it should require not many more operations than forwarding a traditional IP multicast packet or copying a packet to the monitor port on a switch.

In order for this branching operation to take place on a line card, it must be possible to match an incoming packet to the correct multicast group and the corresponding list of children from the packet's IP header alone. The forwarding tables necessary for this can be prepared in the control plane when the router joins the multicast group: it merely stores the transmitting address and port of its parent router alongside the (source, port) tuple that identifies the group. When a new child joins the group, the router looks up the (source, port) tuple provided during the join and identifies the appropriate parent router and port. This new child is then added to the list of children that packets from that parent router are forwarded to.

Control behaviour would occur in the router in much the same way as described for the software implementation in the previous section, except that no filtering rules would be necessary as the router could be deployed in a part of the network likely to carry Lorikeet traffic. Since modifications to the tree are not as frequent as the forwarding of data packets, they may be handled by the router's main CPU as are other routing and control protocols.

7.2.3 Lorikeet Sources

A Lorikeet source node is conceptually similar to a World Wide Web server: it handles requests for information and transmits that information to receivers when requested. The key difference is that, unlike the Web, a Lorikeet multicast group is not a client/server situation in which a single client requests information and is sent it directly by the server. Instead, the source handles join requests from receivers, organises their addition to the multicast tree, and transmits only to its direct children (which handle further dissemination of the data via the tree themselves).

We can think of the source as having two processes operating per stream (since a source can serve multiple different streams). The first process listens for control messages, such as leave requests from the source's direct children or join requests from new receivers in the network. When a join request from a new receiver is received by the source, it begins the join operation described in Chapter 4 and finds the new receiver a parent in the tree. The source's direct children are managed in the same way as a capable router manages its children, described in the previous section.

The second process running on the source handles the transmission of data to the multicast tree. This process acts as a pipe, receiving the data stream from an input device (such as a network connection, or a video capture mechanism), performing any data encoding and packetisation that is necessary, and transmitting a copy of the resultant stream of packets to each child node that is connected directly to the source. Those child nodes that are routers will then re-transmit the packets to their children, and so on.

7.3 Systems Issues

In this section, we describe issues that affect the multicast tree at a system level, resulting from the interactions of capable routers, receivers and the source. First, we consider the issue of finite resource limits in capable routers and the necessary behaviour to deal with situations in which those limits are reached. Second, we describe techniques for providing robustness against failure in the multicast tree. Third, we discuss Lorikeet's behaviour in situations where multiple operations that modify the tree (joins, leaves and rearrangement) take place simultaneously. Next, we examine the question of 'handover' in Lorikeet's proposed rearrangement algorithms, outlining how to minimise disruption of the stream while a router is being re-parented. Finally, we discuss the security ramifications of our design, identifying properties of the design that protect against typical weaknesses in multicast protocols, as well as discussing some possible attacks that are difficult to prevent.

7.3.1 Load and Capacity

It is unrealistic to assume that unlimited resources are available on Lorikeet routers: in practice, all routers have physical limits on their available CPU time, memory and interface bandwidth or link capacity. While our description of Lorikeet in Chapter 4 focuses on Lorikeet's topological behaviour, these physical limits have been considered and are easily addressed in implementation.

All three limits can be enforced by refusing to support additional child nodes (receivers or routers undergoing rearrangements) when they are reached. For example, a capable router that is approaching any of these resource limits may begin returning a "router full" response as part of a join operation, when it is asked for the cost of its path to a new receiver. This would result in that router's removal from consideration as a parent in the join, and the new receiver or rearranged router would consequently be parented elsewhere in the tree, where sufficient capacity is available. This approach does remove the branch of the tree downstream of the "full" router from consideration as well; this is no different, however, from what would have occurred if the router were available but simply more expensive than one of its siblings.

These limits could be enforced by either monitoring the status of the resources in question and entering “full” mode when they reach a threshold, or by setting administrative limits on the maximum number of groups and the maximum number of children that the router is to support. Note that interface bandwidth is not a global resource; it is quite possible to support a child connected via one network interface when another downstream interface is saturated.

Clearing a router of the groups it is participating in (for a reboot or scheduled downtime, for example) can be achieved gracefully through Lorikeet’s join mechanism, as described for rejoin rearrangement. Children (both receivers and other routers) are simply notified that they must rejoin the group. Since the router itself will not be accepting new children until the downtime is complete, these children will join the group at different parent routers and continue receiving the stream.

7.3.2 Robustness and Failure Recovery

The data in a Lorikeet tree is delivered hop-by-hop from the source, down a tree of routers, to the set of receivers. The failure of any router in the tree will therefore necessarily partition the tree and prevent delivery to the subtree supported by that router. In these circumstances, the tree must be able to detect and recover from such a failure and provide an alternate delivery path to that subtree if possible.

Failure in a Lorikeet tree can occur at three different points;

1. Failure of the source.
2. Failure of a receiver.
3. Failure of an intermediate router in the tree.

Each of these scenarios is treated in detail in the following sections.

Failure of the source

Lorikeet is designed to support single-source live video transmissions, and its tree structure is built upon the assumption that all data is transmitted from a single source. Building in support for multiple sources, therefore, is not possible without major modifications to the protocol's design. Other mechanisms may be used to increase the robustness of the source to failure, much as is done for web servers: for example, the multicast source could be made a "repeater", streaming the data sent to it by multiple redundant "real" sources that are protected from the outside network.

Failure of a receiver

In a Lorikeet tree, all receivers are leaf nodes. This property means that receivers may fail without affecting any other node in the tree, unlike in other protocols where a receiver can be required to support a subtree of other receivers. Hence, the failure of a receiver in a Lorikeet tree does not require any explicit recovery to take place beyond the cessation of data transmission from its parent. Unfortunately, because data delivery in the tree only occurs in the downstream direction, extra message passing must be added to the protocol to allow the parent to detect the receiver's failure.

Our suggested implementation is to require routers to query their direct children periodically with a status request, to which they must reply within a defined timeout period. If a child does not acknowledge three successive requests, the parent router must treat them as having left the tree and remove them according to the procedure described in Chapter 4. This allows a child node that has failed, or whose leave request has been lost, to be pruned from the tree.

Without this sort of recovery procedure, a failed receiver could prevent a router from being rearranged (since we trigger rearrangement when a router becomes non-branching) or being removed from the tree if appropriate. It is important to note, however, that these exchanges need not occur very frequently; routers could check their children for failures once every minute, for example. The presence of an undetected failed receiver does not affect the distribution tree or the underlying network apart from the bandwidth required for a single copy of the data stream on the final tree hop, and the resources that it consumes on its parent router.

Failure of a Lorikeet router

The failure of a Lorikeet router is the most complex of the three cases described here. Unlike a receiver, a router has a subtree to which it distributes data received from its parent; therefore, its failure results in this subtree being disconnected from the data source.

We propose the detection of router failure using the same mechanism as is used for the detection of failed receivers described above. Each node participating in the tree (both routers and receivers) must periodically exchange status request messages with its parent router, to ensure that each end of the "hop" in the tree is still active. If a node does not receive a response to a status request within a defined timeout period, the request must be repeated. If three successive requests time out without a response, the node must assume that the other node has failed and must attempt recovery.

If it is the *parent* node that has failed, the child node must rejoin the tree using the standard join procedure as described in Section 4.5.2; the only difference being that the node could be either a receiver or a router with a downstream subtree already in place.

If it is the *child* node that has failed to respond, the parent router has no option but to remove it from the tree. In the case of a genuine failure, the failed node's children (if any) will also detect the failure and rejoin the tree using the procedure above.

7.3.3 Multiple Simultaneous Operations

Lorikeet is designed to operate as a distributed system, with no centralised management (beyond the source's initial involvement) of joins and other operations that modify the multicast tree. In such a system, it is probable that these operations will occasionally overlap or occur at the same time, since they are not being scheduled by a central entity in the network. Lorikeet's hierarchical design ensures that changes to the tree are localised, but care should still be taken in design and implementation to ensure that simultaneous operations on the tree do not result in loops or partitions in the tree, or other anomalies. In this section, we examine each topological operation in the Lorikeet protocol and consider its behaviour in the event of multiple simultaneous operations.

Joins

Join operations in Lorikeet follow the following procedure:

1. A new receiver contacts the source and requests to join the tree;
2. The source finds the new receiver a parent router through a search of the tree;
3. The source connects to that parent router and begins receiving the data stream.

Since this operation only adds new leaf nodes to the tree and does not modify the connections between other nodes, multiple joins can proceed simultaneously without requiring any sort of synchronisation.

Leaves

Leave operations in Lorikeet are performed by having the leaving receiver contact its parent router in the tree and sending a 'leave' message. That router then disconnects the receiver from the tree and stops sending it data packets from the multicast. Since all receivers are leaf nodes, the disconnection of a receiver has no direct impact on other receivers in the tree (as it would in a system where receivers can support other receivers).

A leave operation in Lorikeet can, however, modify the topology of the tree to a greater extent than the simple removal of a leaf node. If the leaving receiver was the only child of its parent router, then that parent router will also leave the tree, and this will occur recursively up that branch of the tree until a router with other children is encountered. If the leaving receiver's parent had only one other child, then the leave operation may trigger a rearrangement as the parent router changes status from a branching router into a non-branching router.

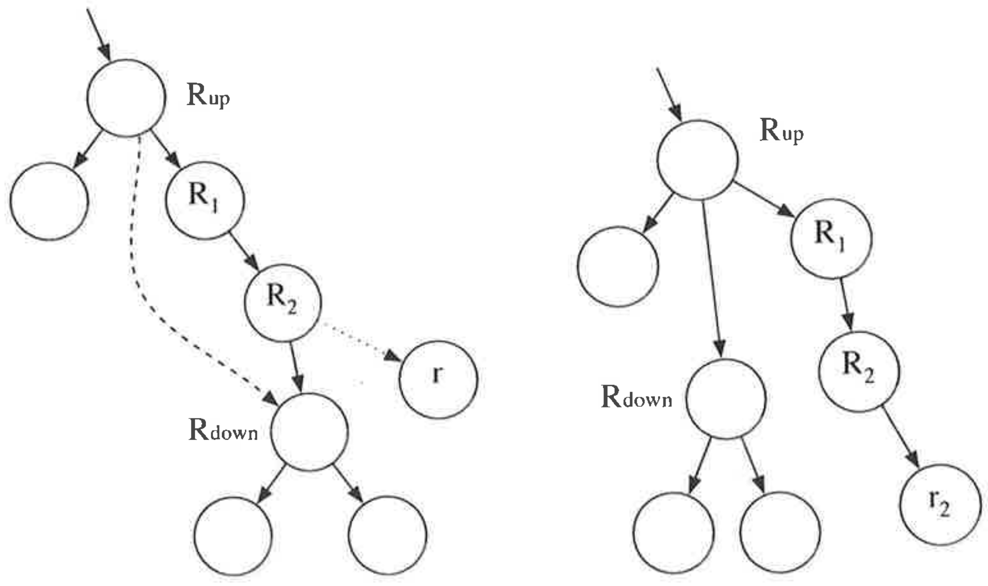
Although these events involve changes to the structure of the tree, they can be implemented without requiring complex locking procedures. In the case of the router with a single child which is leaving the tree, that router may also leave the tree as long as it is not currently negotiating a join with a new receiver. Similar logic applies in the case of rearrangement, described in the next section.

The issue of how to determine when a router is not currently negotiating a join with a new receiver requires some explanation. In the case of the Simple Join technique, the question is easily answered: when a router in the tree passes the query on to its cheapest child, it is no longer participating in the join. This is because decisions in the Simple Join algorithm are taken locally and immediately. In the Path-Greedy Join algorithm, however, a router may be called upon to become the parent of a new receiver much later, after the complete tree path has been traversed. In this case, the routers in the tree path (those that are being considered as potential parents) must be notified when a parent is selected. This can be achieved by sending a message upstream from the last node in the tree path (where the decision is made), announcing that the parent for the receiver has been selected and that those routers can consider themselves free to leave if appropriate. A router with no children that is involved in negotiating a join will leave after receiving this message.

Rearrangement

In Section 4.5.4 we described two rearrangement techniques for Lorikeet: the Path rearrangement algorithm, and the Rejoin rearrangement algorithm. Both rearrangement algorithms are triggered by a receiver leaving the tree, changing its parent node from a branching router to a non-branching router.

In the Path rearrangement algorithm, we attempt to remove a chain of non-branching routers from the tree. This is done by searching upstream and downstream for the nearest branching routers (or the source and a receiver) and comparing their current path through the tree to a direct connection. If connecting them directly would improve the cost of the tree, a direct connection is made and the chain of non-branching routers is disconnected. One approach to dealing with receiver joins on the chain during such a rearrangement would be to disallow those routers from accepting new receivers while the rearrangement is in progress. However, this is not necessary: instead, any new receivers may be accepted as normal, and the chain of routers will be left in place if it is supporting a receiver, as is done in the leave operation.



(a) Before rearrangement. Receiver r has left the tree.

(b) After rearrangement. Receiver r_2 joined the tree as the rearrangement of R_{down} was taking place.

Figure 23: Path Rearrangement with Simultaneous Joins

Figure 23 illustrates this situation. In the figure shown, receiver r has just left the tree, leaving its parent router R_2 a non-branching router. This triggers a rearrangement, and R_2 sends out probes to identify the branching routers R_{up} and R_{down}. It is determined that the direct path between R_{up} and R_{down} is cheaper than the tree path $(R_{up}, R_1, R_2, R_{down})$, so they are connected by the direct path. As this is occurring, a new receiver r_2 joins the tree at our original non-branching router R_2. Consequently, we disconnect R_{down} from R_2, but we do not prune R_2 as it now supports another receiver. The pruning of this "old" path should follow the same behaviour as the recursive pruning done as part of a receiver leave.

The Rejoin rearrangement algorithm identifies R_{down} in the same way as the Path algorithm, but offers R_{down} a chance to rejoin the tree, using the standard Lorikeet join operation. In this case, the link between R_{down} and its parent router is disconnected and a new parent is connected, changing the tree path for R_{down} and all of the nodes in the tree downstream of R_{down}. This should not affect joins that are in progress during a rearrangement, as the cost calculations that they perform do not involve the tree path (only the marginal cost of connecting the receiver) and no routers are being removed from the tree during the rejoin. If the rejoin is successful, the old upstream path to R_{down} will be pruned recursively according to the logic already described; if new receivers have joined those routers in the interim, they will be kept in place. If the rejoin is not successful, the algorithm falls back to attempting Path Rearrangement.

7.3.4 Handoff in Rearrangement

Rearrangement of the multicast tree in the Lorikeet protocol was described in detail in Section 4.5.4. It is triggered by a topological event (the change of a router from branching to non-branching) and involves the potential migration of a single router (the nearest downstream branching router) to a different location in the tree.

An even more efficient tree could potentially be created by making more complex changes to the tree structure, as is done by the ARIES algorithm [8] with its use of a Steiner heuristic to rebuild and optimise portions of the tree. This approach, however, is quite invasive: it deletes and re-connects all of the links in a portion of the tree, potentially disrupting the flow of packets being delivered through the tree to those nodes and any nodes further downstream. In comparison, we feel that the re-parenting of a single router is much simpler, requires less communication overhead, and is therefore more easily achieved with minimal disruption to the flow of data to the branch of the tree downstream of that router.

In implementation, the most effective way to achieve the relocation without interrupting the packet flow is to perform the steps necessary for rearrangement in the following sequence:

1. Signal R, the nearest downstream branching router, that it is to be rearranged. R continues receiving packets from $R_{current}$, its current parent router.
2. Using the chosen rearrangement strategy, locate R_{new}, the new parent router for R.
3. Connect R_{new} as the parent of R, and have R_{new} begin sending packets to R.
4. R remains connected to both $R_{current}$ and R_{new} until it is synchronised: that is, it is receiving the same packets from both parents.
5. R sends a leave message to $R_{current}$ and disconnects from it, completing the rearrangement.

This ensures that R is connected to at least one parent router at all times during its relocation in the tree and that the flow of packets through the tree to R's children is not interrupted. This description makes the assumption that the upstream capacity available to R is at least twice the capacity required for the stream, which we feel is a reasonable assumption for a router in the core of the network or even one on the border of a network operated by a small ISP. Note that receivers are never rearranged, as such an operation would not benefit the rest of the tree and because their upstream capacity is more likely to be limited. Instead, as described in Section 4.5.4, the last capable router on the path is rearranged in the event that there is no downstream branching router when a rearrangement is triggered.

In the case where two copies of the stream is sufficient to exceed the upstream capacity available to R, the rearrangement would have to be performed by interrupting the flow of data. However, the effects of this interruption downstream could be mitigated by having R cache enough packets, before disconnecting from its old parent, to last the time taken to connect to its new parent.

7.3.5 Security

The major focus in this work is the topological behaviour of the Lorikeet protocol. However, security is a concern in the design of any practical network protocol. This is particularly true of multicast protocols, since (unlike client/server protocols) misuse of a multicast tree can potentially affect many users simultaneously. In this section, we briefly outline some of the potential security implications of our design. We do not provide detailed solutions, but merely raise and briefly discuss some of the relevant issues.

The requirement that all receivers be leaf nodes in the tree effectively insulates receivers from each other. Receivers can only receive data from routers in the tree, and are not directly affected by other receivers. Allowing receivers to become tree parents would allow malicious receivers to potentially modify the stream and deliver different data to downstream nodes, or simply not re-send the packets entirely. It is less likely that routers in the network operated by service providers would engage in such malicious activity.

Lorikeet's hierarchical structure for control protects the tree from many distributed attacks. No single node in the tree (not even the source) has complete information about the topology of the tree, or even its population. Instead, nodes store only the addresses of their children and the addresses of the nodes in their path back to the source. Modifications to the tree's topology are performed as directed by messages sent up and down the tree, with join messages always originated by the source. This allows some authentication of topological changes, since they must always be passed to a node from a direct neighbour in the tree.

Likewise, Lorikeet routers only forward data packets arriving from their upstream parent router to their downstream children, so there is no possibility of a new source appearing in the tree and swamping the multicast group with false data packets.

Since Lorikeet's receivers have limited access to the tree and cannot affect each others' access to the stream, malicious capable routers are the most likely source of insecurity in the system. By their very nature, they are trusted to forward data packets to child nodes correctly, and to handle joins and leaves as directed. If a router were to attack or otherwise destabilise the tree, such behaviour could include:

- Dropping (not forwarding) packets, thereby denying all downstream nodes the data stream;
- Feeding downstream routers a different data stream than the one being transmitted by the source;
- Reporting an artificially low cost during a parent search for a new receiver, thus forcing that new receiver to join directly to that router, where it could be denied the stream or fed different data;
- Reporting an artificially high cost during a parent search for a new receiver, reducing the efficiency of the tree by forcing the receiver to join elsewhere.

Many of these attacks could be prevented or at least detected through the use of digital signing of the data by the source. However, this would require that complete data be delivered in order to each receiver, which means that additional complexity in the protocol is necessary to provide reliability, as in the many reliable multicast protocols (for example, [47, 48, 58]) in the literature. Such "hijacking" or "man-in-the-middle" attacks are possible with virtually any networked protocol that does not use end-to-end encryption. We feel that the overhead of providing complete reliability (with the associated re-sending of packets) and end-to-end signing or encryption in order to protect Lorikeet trees against malicious capable routers is unjustified: it is in the interest of service providers (who would deploy such routers) to ensure that they are operating correctly, much as it is in their interest to make sure that they correctly forward IP packets and correctly announce BGP routes.

7.4 Deployment

As shown in Chapter 6, when the concept of a directory node service was introduced, Lorikeet's efficiency is very dependent on its ability to discover capable routers in the network. Capable routers are necessary for branching: if no capable routers can be found between the source and receivers joining the tree, those receivers will join directly to the source, resulting in a simultaneous unicast scenario.

A similar issue has plagued traditional IP multicast for many years. IP multicast protocols require complete multicast protocol deployment on all of the networks spanned by a group in order for that group to operate. Deployment of native IP multicast across the wider Internet so that it can be accessed by ordinary home users, however, has not happened. There are a variety of reasons for this lack of deployment, including the CPU and storage load such protocols impose on ISPs' routers, the difficulty in providing an appropriate charging model for multicast traffic, and the lack of applications. The latter is a chicken-and-egg scenario: applications are difficult to deploy when the infrastructure they require is not widely available, and ISPs were not willing to provide the infrastructure without user demand, which is generally fuelled by applications.

Lorikeet does not require complete deployment across the network to operate, thus solving this issue. However, for this sort of protocol to become used, there must still be commercial and other drivers for its implementation and deployment by content providers, ISPs and end users. We discuss each of these entities in turn in the following sections.

7.4.1 Content Providers

At present, current providers of streaming multimedia content on the Internet primarily rely upon simultaneous unicast transmission alone for their content. Simultaneous unicast operates everywhere, across the complete Internet, and can be deployed with the reasonable expectation that all users interested in the content will be able to access it.

However, the use of simultaneous unicast puts considerable pressure on content providers to have enough server resources to support a large number of simultaneous users and very high-capacity links to the outside world. Since every user receiving the stream must have their own unicast connection to the source, the source must be well-connected enough to support the number of receivers that are connected. As the content provider's connectivity to the outside world approaches saturation, newly arriving receivers will either degrade the quality of the transmission *to all receivers*, or (if some form of access control is used) be denied access to the stream altogether. Very popular content on the Internet often exhibits this phenomenon when the content provider runs short of capacity or even CPU or memory on the server hosting the content, resulting in long delays or server failure.

The use of a multicast protocol would alleviate this situation by allowing the content provider to support the same number of receivers with a much smaller capacity requirement, since only a small number of connections directly to the source of the transmission would be required. Branching within the network would take care of delivering the stream to the complete set of receivers. This scenario potentially scales much better than simultaneous unicast, as well: assuming that sufficient branching routers are present in the network, a much larger number of receivers could be supported with no additional impact on the capacity required by the content provider.

Hence, the deployment of a multicast protocol that will operate over the existing Internet network, such as Lorikeet, provides content providers with a way to support a large number of users without a very high-capacity link to the Internet. It also increases the likelihood that they will be able to cope with spikes in demand without disruption of service.

7.4.2 Internet Service Providers

In order for Lorikeet to offer improved performance over simultaneous unicast transmission, capable routers need to be present in the network. The presence of these routers is largely dependent on Internet Service Providers having compelling reasons for deploying them and supporting the Lorikeet protocol.

Obviously, the strongest reason for the deployment of a new service by an ISP (and Lorikeet can be considered a new service, due to its requirement for changes to routers) is demand, in this case from content providers and users. If a service like Lorikeet were to be used by a significant fraction of an ISP's customers, that ISP would be very likely to deploy capable routers in order to improve the quality of that service experienced by its customers. A second driver for ISP deployment of capable routers in their networks is the cost savings such a deployment could provide. Once a capable router is present in the ISP's network, between its users and the Lorikeet sources elsewhere on the Internet, that capable router will become a local branching router for all users on that network. If the ISP has several users subscribing to the same multicast stream, it can supply all of those users from the single stream entering the branching router, rather than carrying a separate stream from elsewhere on the Internet for *each* receiver.

This approach is similar in spirit to the many ISPs and other organisations around the world carrying local mirrors of software repositories for their customers. A more direct comparison can be made with the use of Internet radio relays by some ISPs[2] to provide better performance to customers and save bandwidth on popular

[2]For an Australian example, see Internode's Online Streaming Radio page at `http://www.`

radio streams.

7.4.3 End Users

For end users, Lorikeet's advantages over simultaneous unicast are the same as those provided by other multicast systems, including traditional IP multicast. Multicast streams offer better performance, as the stream is most likely to be coming from a capable router that is closer in network terms than the source. They also offer improved availability, as the source is less likely to be swamped by requests for popular content (as described in Section 7.4.1). Unlike IP multicast systems, however, Lorikeet is deployable immediately, without requiring any protocol support from the users' ISPs.

In addition, Lorikeet is a less complex system to deploy than other protocols that require traditional IP multicast support or other new services at the network layer. For end-users, Lorikeet requires only unicast connectivity to the outside world, with no changes to users' routers or operating systems beyond the installation of an application (such as a browser plugin or standalone player) that supports the protocol.

7.4.4 Placing Capable Routers

The issue of the placement of capable routers in the network is highly dependent on how they are implemented, as described in Section 7.2.2. If a hardware router can be developed that can handle Lorikeet forwarding with comparable speed to native IP forwarding in current hardware, then these routers can be deployed near the core of the network, where they can provide easily-accessible branching points for multicast streams traversing the core. If, on the other hand, Lorikeet proves difficult to implement with sufficient performance, then Lorikeet branching must take place further out in the edge of the network, where traffic volumes are lower. In that

`internode.on.net/radio/`.

scenario, routers can be placed in ISPs' networks as described earlier, providing support at least to those ISPs' customers and other nearby users. It is important to note that once a capable router is discovered by a multicast source or other tree router, it can be used for branching on subsequent joins for any receivers that are appropriately close. Hence, even if it is not feasible to place capable routers on direct paths in the core of the network, their discovery can be achieved through the use of targeted filtering rules in standard routers or a directory service like the one proposed in Chapter 6.

This chapter has endeavoured to discuss some of the issues involved in writing a complete, practical implementation of the Lorikeet protocol. It is not a complete protocol definition, but was rather intended to illustrate the reasoning behind some of our design decisions, as well as to discuss the ramifications of wider deployment of Lorikeet-based multicast. In the following chapter, we investigate several different possible extensions to the Lorikeet protocol for future research, building on its hierarchical structure and hop-by-hop delivery to provide additional functionality.

Chapter 8

Conclusions and Future Work

In this dissertation we have presented our work on the multicast transmission of streaming multimedia over the Internet, from an analysis of existing work and the underlying Steiner Tree Problem in Networks, through to the development of a new protocol called Lorikeet. This chapter is a summary of our findings, followed by a discussion of future work and further possible research in the area.

8.1 Summary

In Chapter 1 we introduced our topic of research and presented some background material on multicast transmission, introducing a number of elements that are common to most currently available protocols and research in the field. In particular, we noted that the standardised IP multicast protocols are not in widespread use on today's Internet, despite multicast's obvious advantages for the dissemination of data to large groups. We also drew attention to the properties required by today's multimedia applications, such as the need to cope with very large, dynamic receiver populations.

This background material was extended in Chapter 2, where we examined the current state of research in multicast as well as the protocols that have been standardised by the Internet Engineering Task Force (IETF). We began by describing the current suite of IP multicast standards from RFC 966 [20], which introduced the concept of a "host group" identified by a single IP address, through to current protocols like Protocol Independent Multicast - Sparse Mode (PIM-SM) [26], the most popular current standard for IP multicast delivery. Current work on Source Specific Multicast (SSM) [9] was also described.

We then presented a brief overview of multicast research from the literature. Our discussion was divided into three broad sections: Small-Group multicast protocols, Application-Level and Overlay multicast protocols and Topology-Aware multicast protocols. These groups do not represent a strict taxonomy of multicast protocols; there is considerable overlap even between these broad groupings. They do, however, present different solutions to the problems associated with traditional IP multicast: Small-group multicast removes the need for separate group addresses and multicast routing protocols; application-level and overlay protocols remove the dependency on protocol support in the network, relying on unicast transmission between end-hosts; and the topology-aware protocols attempt to construct efficient trees, rather than relying on the simple reverse-path techniques used by IP multicast.

In Chapter 3 we investigated the graph-theoretic problem that underlies the construction of multicast trees, the Steiner Tree Problem in Networks. We began by defining the Steiner Tree Problem and explaining its relationship to multicast tree construction, in both static situations (with a fixed set of receivers) and dynamic ones (with a changing set of receivers). We investigated two algorithms for finding optimal Steiner Minimal Trees (SMTs) and four heuristics for finding approximate solutions to the problem. Since finding the SMT for an arbitrary network is NP-complete, our primary focus (particularly with a view to applications in multicast) was on the heuristics, which are able to operate in polynomial time. These heuristics were compared in simulation, and their suitability as a basis for an online multicast tree construction algorithm discussed.

Chapter 4 presented the core of our work on multicast, a description of the Lorikeet protocol. We commenced with a description of a target application: the streaming of multimedia content from a single source. This is currently a very popular application on the Internet that predominantly uses simultaneous unicast streams for delivery. In simultaneous unicast, each receiver connects directly to the source of the content and receives their own unicast stream of packets. The development of a more efficient multicast delivery system could result in large reductions in bandwidth use and increased availability for live streaming content.

From this application and the properties of the network environment in which it is to operate (the current Internet), we developed a set of requirements for a new multicast protocol. These include a number which address the limitations of other multicast protocols when used for single-source multicast, such as the requirements that the multicast tree be constructed in the direction of data delivery from the source, that the protocol cope efficiently with dynamic membership of the group, and that the protocol should operate even if not all routers in the network support it. The latter property enables Lorikeet to surmount the chicken-and-egg problem that has slowed the deployment of IP multicast, described in Section 7.4, while still allowing routers in the network to provide branching to the tree (unlike application-level multicast).

We developed two different algorithms for handling receiver joins in Lorikeet, the Simple Join and Path-Greedy join algorithms. Both of these perform a limited recursive search of the multicast tree, with the Path-Greedy join extending the search further in order to improve the result. Both algorithms have significantly lower complexity than exhaustively searching the tree. This explicit use of the tree for joining is very different from what most other multicast protocols do, since it builds the tree in the same direction as data delivery occurs (from the source to the receivers) and attempts to maximise the chance of reusing existing paths through the network. In contrast, traditional IP multicast and many other protocols build trees based on the use of the paths from receivers to the source. Since many network paths are asymmetric in today's networks, this leads to a tree that is suboptimal for data delivery in the forward-path direction.

Chapter 4 also described two rearrangement algorithms, called Path and Rejoin rearrangement. These algorithms were designed to maintain the efficiency of the tree in the face of changes to the receiver set. Both algorithms improve the quality of the tree by re-connecting a capable router to the tree via a different parent, thereby taking advantage of changes made to the tree since the router was first connected. To trigger a rearrangement, Lorikeet uses a *topological event* – a router changing status from branching to non-branching – which to the best of our knowledge is a novel technique. In other algorithms, rearrangements are triggered by periodic timers or by counters, rather than directly by a change in the topology of the tree.

The Lorikeet protocol, with its two join operation variants and two rearrangement algorithms, was implemented in simulation, along with several other protocols: the Source-Join and Greedy algorithms, which are simple approaches to provide baselines for comparison, and the ARIES [8, 7], DSG [32] and REUNITE/HBH [71, 18] protocols (described in detail in Section 5.2).

In Chapter 5 we performed a complexity and performance analysis of Lorikeet and these other protocols. We examined all of the algorithms listed above (including the different variants of Lorikeet) in terms of both analytical worst-case and empirical average-case message complexity. These results showed clearly that Lorikeet was much more efficient than those competitors which use exhaustive searches of the tree as part of their join operations. We then investigated the relative performance of the different variants of the Lorikeet algorithm, in order to determine which join and rearrangement algorithms were most effective. As a result, we selected the Path-Greedy join with Rejoin rearrangement as the representative "version" of Lorikeet for all further tests.

Next, we performed a series of simulations analysing Lorikeet's performance in terms of total tree cost in comparison with other protocols. The results of these simulations show that Lorikeet produces a tree with a total cost that falls between those of the low cost trees produced by ARIES, Greedy and DSG, algorithms that use an exhaustive search, and the high cost trees generated by REUNITE/HBH and

Source-Join, which both join new receivers directly to the source and rely on shared paths to provide branching. Lorikeet produced trees within 8% of the cost of the Greedy tree (for all intents and purposes a lower bound for tree cost, due to its use of an exhaustive search), while maintaining much lower complexity.

Finally, we investigated the performance of the two protocols in our simulation that support incremental deployment, Lorikeet and REUNITE/HBH. In our simulations, the total tree cost for both protocols was reduced as the proportion of capable routers in the network increased. At low capable router proportions (10% to 20%), Lorikeet outperformed REUNITE/HBH in terms of tree cost by more than a third. Even at higher proportions, Lorikeet constructed significantly cheaper trees until almost all routers were capable, where both protocols approach the limit of their performance. Lorikeet's better performance can be attributed to the fact that Lorikeet searches the tree explicitly during joins and is able to select from a wider range of potential parent routers. This enables Lorikeet to perform more branching, and hence produce lower-cost trees.

In Chapter 6 we explored the possibility of adding functionality to Lorikeet to aid the discovery of capable routers in networks where the majority of capable routers are *not* present in the core of the network. Our solution was to add *directory nodes* to the network and to the protocol. The Path-Greedy join algorithm was modified to query a directory node for additional "adopted routers" over the course of a join, providing a mechanism for other capable routers to be discovered.

Upon simulation, we found that this addition improved Lorikeet's performance in the case when all the capable routers were off the shortest paths. However, when capable routers in the core of the network were introduced into the simulation, they quickly began to provide the bulk of the branching in the tree, rendering the gains due to the use of directory nodes negligible. We concluded that while the use of directory nodes has significant benefits in networks with few capable routers, those benefits decrease as deployment becomes more widespread. If such deployment is to occur relatively rapidly, this negates the overall benefit of providing a directory node service.

8.2 Potential Implementation-Related Research

In this thesis, our primary focus has been on the analysis of the topological behaviour of multicast protocols and the development of Lorikeet's tree construction and maintenance algorithms. All of our results and analysis, however, have been gathered using a topological simulation of the protocol running on a discrete event simulator. The development of a complete, working version of Lorikeet would allow parts of the protocol that we have not investigated in depth to be tested. Examples of these properties, which have been touched on briefly in Chapter 7, include the storage requirements on capable routers, the protocol's behaviour when multiple simultaneous changes to the tree are taking place and the logistics of performing a rearrangement without interrupting the data stream to downstream nodes.

Implementation of the system in software and on router line cards would allow performance testing of Lorikeet in a variety of different locations in the network, from small networks on the edge with low rates of traffic, through to core routers in research networks that handle much higher throughput. It would be valuable to ensure through real-world testing that the extra processing required to participate in a Lorikeet multicast tree is practically achievable on current router hardware.

Once practical implementations of the protocol are available, performance testing of Lorikeet through real use across the current Internet would provide information on its performance on a wide-area network under real use. In particular, it would be possible to show how appropriate our simulation topologies are for simulating real Internet topologies and how the distribution of capable routers through the network affects the efficiency of multicast trees built with them.

8.3 Potential Protocol Extension

There are several related areas of research that complement our work on Lorikeet and could provide further improvements to the protocol or application-specific extensions. In this section, we outline a number of these possibilities for future work. Layered Video Delivery and Local Recovery are both extensions that leverage Lorikeet's control of the multicast tree topology to provide different levels of service or provide additional robustness. Subsequently, we discuss several other topics of interest: access control and authentication, calculation of the cost metric and the collection of statistics from receivers.

8.3.1 Layered Video Delivery

In today's Internet, information is accessed by a huge variety of devices, ranging from small, bandwidth- and display-constrained mobile phones through to commercial users with large capacity network links and high-resolution displays. This heterogeneity has led to considerable interest in providing differentiated multimedia content, designed to serve each class of receiving device with its own optimised stream [49, 54, 43]. For example, most movie trailers from large commercial studios are now made available on the World Wide Web for viewing in a number of sizes, from low resolution appropriate for viewing on mobile devices through to full high-definition video with surround sound, to cater for users with different requirements. This approach generally requires the user to select the appropriate stream manually, or relies on a user-selected preference that is set in the client application.

A great deal of research in the literature addresses this issue of heterogeneity through the development of scalable video codecs (compression/decompression algorithms) that perform what is termed *layered* or *hierarchical* coding. These algorithms code a video signal into a number of separate "layers", the complete set of which can be decoded together to recover the complete video signal, or a subset can be used to obtain a lower-quality video signal. This principle is applied in high-definition television (HDTV) broadcasts using the MPEG-2 standard, where the signal comprises a standard-definition (SD) layer and a second "enhancement layer" that provides the extra resolution to build the high-definition (HD) image. In this case, layered encoding is employed primarily to provide reliability – if the channel is too noisy to recover the HD layer, the picture will drop back to SD [16], which is transmitted with more robust error-correction. Several projects have sought to use layered coding to provide similar benefits to multicast transmission over the Internet [55, 67, 77], using multiple layers to tailor the picture to receivers or to provide additional reliability through the use of error-correction layers. McCanne *et al.*'s RLM protocol [54], for example, distributes layered video via a set of multicast groups, one per layer. Receivers can then adjust their receiving rate (according to congestion and available capacity) by joining and leaving those groups.

Lorikeet's hierarchical, managed approach to tree construction can be combined with layered video coding to provide support for heterogeneous receivers without the overhead of maintaining separate multicast trees for each layer. Instead, we suggest that the source makes available the complete set of layers and that capable routers in the tree perform "layer stripping" when appropriate. Capable routers need only provide their children with the number of layers they can support, and the number of layers being received from upstream can be "upgraded" to cope with the addition of a new child that requires a higher rate than that which is currently being received.

Figure 24 shows a simple example of a 2MB/s stream, partitioned into four layers of 512kb/s each, being transmitted by source S to five receivers with varying requirements. As the figure shows, each of the routers in the tree needs only to receive the number of layers required by its highest-rate child, making more efficient use of the network if many receivers in the tree do not require the full complement of layers.

Implementation of the control mechanism for layered video could be done in a number of ways. One option is a *reactive* model, where capable routers react to the capacity requirements of their children and upgrade their streams as required. A

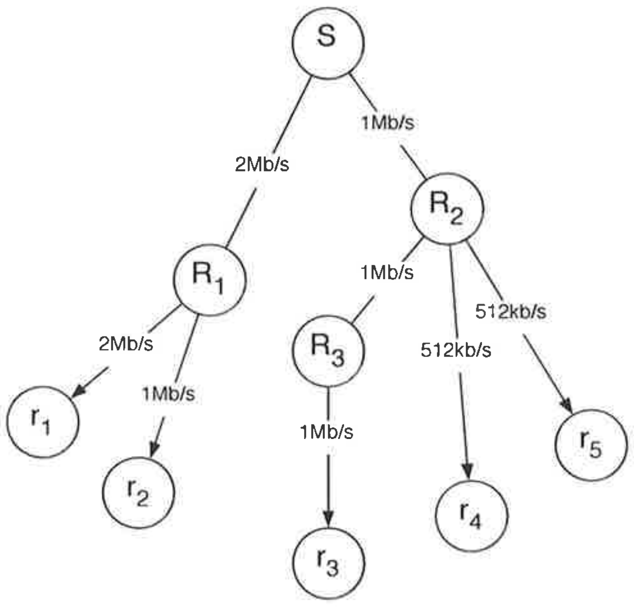


Figure 24: Layered Video Delivery using a Lorikeet Multicast Tree

more complex *metric-based* model could also be used, where the required capacity is considered as part of the join operation, and new receivers are joined to routers which are (a) already receiving a stream of sufficient rate, or (b) can be upgraded to the desired rate cheaply. Similarly, capable routers could downgrade the number of layers being received when a high-rate child leaves the tree, much like the recursive pruning operation that occurs when a router's last child leaves.

Adding the necessary control operations to Lorikeet's capable routers would not be difficult, although tuning the join operation to use a modified metric may result in the creation of very different trees. If this extension were to be implemented in hardware routers, however, delivery of these packets may present a challenge. The layer identifier presents another piece of information that needs to be decoded from incoming packets in order to forward them to the correct receivers: namely, those that are receiving that layer of a particular stream. In IPv6, this could be achieved using the "flow label" field, designed for identifying different flows, but this solution is not appropriate in today's largely IPv4-based Internet. In IPv4 we could use the IP TOS (type of service) bits in the header to identify a layer, as is done by the Differentiated Services (DiffServ) [57] architecture used for providing different levels of service for traffic. However, the TOS bits are used for many different purposes (including DiffServ) by different ISPs, and can not be relied upon to operate correctly across the wider Internet. An alternative approach would be for the source to transmit each layer from a different UDP port, allowing capable routers to match each port to a different layer of the stream. This approach will operate correctly across networks when the TOS bits cannot be relied upon, though it places more of a limit on the number of layers and streams that a source can support. Nevertheless, even two layers per stream (as for standard and high-definition television) could provide significant extra flexibility for users.

8.3.2 Local Recovery

Lorikeet is designed for multicasting one-way streaming multimedia, such as live broadcasts of events or news. For this application, perfectly reliable transmission is not necessary: multimedia data is able to tolerate small amounts of packet loss or damage to packets without becoming visibly or audibly distorted to the viewer. In addition, the playback of received data need not be as immediate as it is required to be in two-way voice communication, where a delay of more than a few hundred milliseconds is very obvious to the participants. A constant delay of a few seconds would not be noticed when watching a cricket match or a news broadcast. Such a delay could be introduced to allow receivers to cache the stream in order to cope with transient network effects like congestion, as well as potentially perform recovery of missed packets.

In [53], Maxemchuk *et al.* present a protocol for performing recovery of lost packets on the Internet Multicast Backbone (MBONE), aimed at improving the quality of video transmitted on multicast groups. Their design involves a subset of receivers in a group cooperating by detecting lost packets and querying a retransmit server for them, which would retransmit the packets in question. After a fixed time delay, the querying receiver then retransmits all of the packets it has received (including the retransmitted ones) on another multicast group, referred to as the repair channel. Receivers that wish to make use of this mechanism join the repair channel instead of the main multicast group and are thereby able to take advantage of retransmitted data. The advantage of this approach is that it can be deployed without changes to the source or to the receivers – these retransmit and repair servers merely need to be deployed in the group.

In Lorikeet, however, we could deploy a packet recovery system without requiring extra multicast groups. Packet recovery could take place on a local, link-by-link basis: for example, capable routers could cache a defined number of packets (for example, ten seconds' worth). When a lost packet is detected by a node, it could send a negative acknowledgement to its parent, requesting the retransmission of the missing packet. That packet would then be sent to that node *only*, rather than retransmitting it to all of the receivers on a separate multicast tree or requiring the creation of a new "repaired" tree.

This approach, like that of Maxemchuk *et al.*, does not seek to provide complete reliable delivery of all packets to all receivers. Instead, it seeks to improve the quality of the stream delivered to receivers by providing a mechanism for retransmission of packets that may not yet have been played back to the user. Such retransmission may have a considerable effect on the user experience, particularly if the network is suffering from congestion. If a receiver is caching several seconds of video in order to minimise network effects such as jitter, then it is quite possible for a lost packet to be recovered from that receiver's parent router before it is due for playback.

8.3.3 Other Further Work

Another large area of research applicable to Lorikeet is that of access control, authentication and charging. Because of the highly distributed nature of a multicast tree, the creation of appropriate methods for the implementation of access control and authentication and the development of a charging model for multicast traffic are not trivial. If a capable router somewhere in the network can simply make additional copies of the stream for receivers at will, how is control of the traffic possible in the case of a content provider that wishes to control and charge for access to its content? In Lorikeet's case, this is simplified somewhat when compared to traditional IP multicast, since Lorikeet requires that joining the tree be done through the source, which can also therefore authenticate and charge receivers. However, protecting the stream so that malicious users cannot receive it without authenticating is an interesting distributed security problem. Similarly, there are other charging models – such as charging by the minute – which would require changes to Lorikeet's join and leave operations, and the development of a mechanism for tracking the lengths of individual receiver sessions.

In Section 4.3, we wrote about the difficulty of finding a metric for use in Lorikeet's cost calculations used to select routers in the tree during a join operation. This metric should represent the cost of a path in terms of network bandwidth, must be increasing and must be calculable at individual routers without requiring significant communications or storage overhead. For use in Lorikeet initially, we have suggested the use of hop-count as this metric, which, although a crude measure of the cost of a path, is available already for use on the current Internet. There is a great deal of interest in the research community and in the commercial world in the provision of guaranteed quality of service over IP networks, however, and it is very possible that research in this area may furnish us with a more accurate metric for determining the cost of a given path. We feel that research on easily making available measurements of the *bottleneck bandwidth* (the bandwidth of the smallest capacity link on the path) or *available bandwidth* (the proportion of the bottleneck bandwidth that is available at a given instant) for a given path would be very beneficial to both Lorikeet and other applications with specific network resource requirements.

In the Lorikeet protocol there is very little feedback transmitted by receivers or routers in the tree to upstream routers or the source. It is possible to use Lorikeet's managed tree structure for collection and aggregation of information: for example, routers could periodically count the receivers that they support directly, counts that could be sent upstream and summed recursively until the source is provided with summary information on which of its downstream branches are "heaviest" with receivers. This aggregation technique could be used to collect many different kinds of information about the tree and its receiver population, which could be put to a variety of uses: examples include lightweight statistics gathering by the source and data collection for more advanced join and rearrangement strategies.

Our work on Lorikeet provides some of the groundwork for a new class of practical protocols for large-scale multimedia transmission, protocols that can be deployed without requiring universal changes to Internet infrastructure and make efficient use of the network's topology. Multicast offers an attractive way to significantly reduce the bandwidth requirements of multimedia applications, something that will become necessary as the use of streaming video over the Internet increases in prevalence.

Bibliography

[1] A. Adams, J. Nicholas, and W. Siadak. RFC 3973: Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol Specification (revised), January 2005.

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

[3] Mozafar Bag-Mohammadi, Siavesh Samadian-Barzoki, and Nasser Yazdani. Linkcast: Fast and scalable multicast routing protocol. In *NETWORKING*, pages 1282–1287. Springer-Verlag, 2004.

[4] Suman Banerjee and Bobby Bhattacharjee. Analysis of the NICE application layer multicast protocol. Technical Report UMIACS TR 2002-60 and CS-TR 4380, Department of Computer Science, University of Maryland, College Park, MD, USA, 2002.

[5] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *ACM SIGCOMM*, pages 205–217, Pittsburgh, Pennsylvania, USA, August 2002. ACM.

[6] T. Bates, R. Chandra, D. Katz, and Y. Rekhter. Multiprotocol extensions for BGP-4, February 1998.

[7] Fred Bauer. *Multicast Routing in Point-to-Point Networks Under Constraints.* PhD thesis, Computer Engineering, University of California, Santa Cruz, June 1996.

[8] Fred Bauer and Anujan Varma. ARIES: A rearrangeable inexpensive edge-based on-line Steiner algorithm. In *IEEE INFOCOM*, pages 361–368, 1996.

[9] S. Bhattacharyya. RFC 3569: An overview of source-specific multicast (SSM), July 2003.

[10] BitTorrent website. `http://www.bittorrent.com/`.

[11] R. Boivie, N. Feldman, Y. Imai, W. Livens, D. Ooms, and O. Paridaens. Explicit multicast (Xcast) basic specification, July 2005. Internet Draft, draft-ooms-xcast-basic-spec-08.txt.

[12] Ali Boudani and Bernard Cousin. SEM: A new small group multicast routing protocol. In *ICT 2003*, volume 1, pages 450–455. IEEE, February 2003.

[13] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan. RFC 3376: Internet Group Management Protocol, version 3, October 2002.

[14] Miguel Castro, Michael B. Jones, Anne-Marie Kermarrec, Antony Rowstron, Marvin Theimer, Helen Wang, and Alec Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *IEEE INFOCOM*. IEEE, 2003.

[15] Yatin Chawathe. Scattercast: an adaptable broadcast distribution framework. *Multimedia Systems Journal*, 9(1):104–118, 2003.

[16] Tihao Chiang and Dimitris Anastassiou. Hierarchical coding of digital television. *IEEE Communications Magazine*, 32(5):38–45, May 1994.

[17] Coral content distribution network website. `http://www.coralcdn.org/`.

[18] Luis Henrique M. K. Costa, Serge Fdida, and Otto Carlos M. B. Duarte. Hop by hop multicast routing protocol. In *ACM SIGCOMM*, pages 249–259, San Diego, CA, USA, August 2001. ACM.

[19] S. Deering. RFC 1112: Host extensions for IP Multicasting, August 1989.

[20] S. E. Deering and D. R. Cheriton. RFC 966: Host groups: A multicast extension to the Internet Protocol, December 1985.

[21] Christophe Diot, Brian Neil Levine, Bryan Lyles, Hassan Kassem, and Doug Balensiefen. Deployment issues for the IP multicast service and architecture. *IEEE Network Special Issue on Multicasting*, 14(1):78–88, January 2000.

[22] S. E. Dreyfus and R. A. Wagner. The Steiner problem in graphs. *Networks*, 1:195–207, 1972.

[23] D. Z. Du and X. Cheng, editors. *Steiner Tree Based Distributed Multicast Routing in Networks*, pages 327–351. Kluwer Academic Publishers, 2001.

[24] Ayman El-Sayed and Vincent Roca. Improving the scalability of an application-level group communication protocol. In *ICT*. IEEE, 2003.

[25] Ayman El-Sayed, Vincent Roca, and Laurent Mathy. A survey of proposals for an alternative group communication service. *IEEE Network Special Issue on Multicasting: An Enabling Technology*, 17(1):44–51, January 2003.

[26] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. RFC 2362: Protocol independent multicast-sparse mode (PIM-SM): Protocol specification, June 1998.

[27] B. Fenner and D. Meyer. Multicast Source Discovery Protocol (MSDP), October 2003.

[28] Paul Francis. Yoid: Extending the Internet Multicast Architecture, April 2000.

[29] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with Coral. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation*, San Francisco, CA, USA, March 2004.

[30] M. R. Garey, R. L. Graham, and D. S. Johnson. The complexity of computing Steiner Minimal Trees. *SIAM Journal of Applied Mathematics*, 32:835–859, 1977.

[31] Gnutella protocol development website. `http://rfc-gnutella.sourceforge.net/`.

[32] Ashish Goel and Kameshwar Munagala. Extending greedy multicast routing to delay sensitive applications. Technical Report STAN-CS-TN-99-89, Dept. of Computer Science, Stanford University, July 1999.

[33] M. Grötschel, A. Martin, and R. Weismantel. The Steiner tree packing problem in VLSI design. *Mathematical Programming*, 78(2):265–281, August 1997.

[34] GWebCache website. `http://www.gnucleus.com/gwebcache/`.

[35] S. L. Hakimi. Steiner's problem in graphs and its implications. *Networks*, 1:113–133, 1971.

[36] C. Hedrick. RFC 1058: Routing Information Protocol, June 1988.

[37] David A. Helder and Sugih Jamin. Banana tree protocol, an end-host multicast protocol. Technical Report TR-429-00, University of Michigan, July 2000.

[38] Hugh W. Holbrook and David R. Cheriton. IP multicast channels: EXPRESS support for large-scale single-source applications. In *ACM SIGCOMM*, pages 65–78, Cambridge, MA, USA, August 1999. ACM.

[39] Yang hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. Enabling conferencing applications on the internet using an overlay multicast architecture. In *ACM SIGCOMM*, pages 55–67, San Diego, CA, USA, August 2001. ACM.

[40] Yang hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. A case for end system multicast. *IEEE Journal on Selected Areas in Communications*, 20(8):1456–1471, October 2002.

[41] Frank K. Hwang, Dana S. Richards, and Pawel Winter. *The Steiner Tree Problem*, volume 53 of *Annals of discrete mathematics*. Elsevier Science, 1992.

[42] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and Jr. James W. O'Toole. Overcast: Reliable multicasting with an overlay network. In *OSDI 2000*, October 2000.

[43] J. Kangasharju, F. Hartanto, M. Reisslein, and K.W. Ross. Distributing layered encoded video through caches. In *IEEE INFOCOM*, Anchorage, Alaska, USA, April 2001. IEEE.

[44] Jussi Kangasharju, James Roberts, and Keith W. Ross. Object replication strategies in content distribution networks. In *Proceedings of WCW'01: Web Caching and Content Distribution Workshop*, Boston, MA, USA, June 2001.

[45] T. Koch and A. Martin. Solving Steiner tree problems in graphs to optimality. *Networks*, 32:207–232, 1998.

[46] Minseok Kwon and Sonia Fahmy. Path-aware overlay multicast. *Computer Networks*, 47(1):23–45, January 2005.

[47] Li-Wei H. Lehman, Stephen J. Garland, and David L. Tennenhouse. Active reliable multicast. In *IEEE INFOCOM*, pages 581–589, San Francisco, CA, USA, March 1998.

[48] B.N. Levine and J.J. Garcia-Luna-Aceves. A comparison of reliable multicast protocols. *ACM Multimedia Systems Journal*, 6(5):334–348, August 1998.

[49] Bo Li and Jiangchuan Liu. Multirate video multicast over the internet: An overview. *Network Magazine*, 17(1):24–29, January 2003.

[50] Le-Chin Eugene Liu and C. Sechen. Multi-layer chip-level global routing using an efficient graph-based Steiner tree heuristic. In *EDTC '97: Proceedings of the 1997 European conference on Design and Test*, pages 311–318, Washington, DC, USA, 1997. IEEE Computer Society.

[51] Laurent Mathy, Roberto Canonico, and David Hutchison. An overlay tree building control protocol. In J. Crowcroft and M. Hofmann, editors, *Networked Group Communication: third International COST264 Workshop, NGC 2001*, pages 76–87, London, UK, November 2001. Springer-Verlag.

[52] Laurent Mathy, Roberto Canonico, Steven Simpson, and David Hutchison. Scalable adaptive hierarchical clustering. In *NETWORKING '02: Proceedings of the Second International IFIP-TC6 Networking Conference on Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; and Mobile and Wireless Communications*, pages 1172–1177, London, UK, 2002. Springer-Verlag.

[53] N. F. Maxemchuk, K. Padmanabhan, and S. Lo. A cooperative packet recovery protocol for multicast video. In *Int. Conf. on Network Protocols*, pages 259–266, Atlanta, Georgia, USA, October 1997.

[54] Steven McCanne, Van Jacobson, and Martin Vetterli. Receiver-driven layered multicast. In *ACM SIGCOMM*, pages 117–130, Stanford, CA, USA, August 1996. ACM.

[55] Steven R McCanne. Scalable compression and transmission of internet multicast video. Technical Report UCB/CSD-96-928, EECS Department, University of California, Berkeley, 1996.

[56] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. BRITE: Universal topology generation from a user's perspective. Technical Report BUCS-TR-2001-003, Computer Science Department, Boston University, April 2001.

[57] K. Nichols, S. Blake, F. Baker, and D. Black. RFC 2474: Definition of the Differentiated Services field (DS field) in the IPv4 and IPv6 headers, December 1998.

[58] Sanjoy Paul, Krishan K. Sabnani, John C. Lin, and Supratik Bhattacharyya. Reliable multicast transport protocol (RMTP). *IEEE Journal on Selected Areas in Communications*, 15(3):407–421, April 1997.

[59] Vern Paxson. End-to-end routing behaviour in the Internet. *IEEE/ACM Transactions on Networking*, 5(5):601–615, October 1997.

[60] Dimitrios Pendarakis, Sherlia Shi, Dinesh Verma, and Marcel Waldvogel. ALMI: An application level multicast infrastructure. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 49–60, March 2001.

[61] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Tech. J.*, 36:1389–1401, 1957.

[62] Apple Quicktime website. `http://www.apple.com/quicktime/`.

[63] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *ACM SIGCOMM*, San Diego, CA, USA, August 2001. ACM.

[64] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level multicast using content-addressable networks. In *Proceedings of NGC 2001*, 2001.

[65] Vincent Roca and Ayman El-Sayed. A Host-Based Multicast (HBM) solution for group communications. In P. Lorenz, editor, *ICN*, pages 610–619. Springer-Verlag, 2001.

[66] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[67] Claudia Schremmer, Christoph Kuhmünch, and Wolfgang Effelsberg. Layered wavelet coding for video. In *11th International Packet Video Workshop (PV 2001)*, page 42 ff., Kyongju, Korea, 2001.

[68] Myung-Ki Shin, Yong-Jin Kim, Ki-Shik Park, and Sang-Ha Kim. Explicit multicast extension (Xcast+) for efficient multicast packet delivery. *ETRI Journal*, 23(4):202–204, December 2001.

[69] Kunwadee Sripanidkulchai, Bruce Maggs, and Hui Zhang. An analysis of live streaming workloads on the Internet. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 41–54, New York, NY, USA, 2004. ACM Press.

[70] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, pages 149–160, San Diego, CA, USA, August 2001. ACM.

[71] Ion Stoica, T. S. Eugene Ng, and Hui Zhang. REUNITE: A recursive unicast approach to multicast. In *IEEE INFOCOM*, Tel Aviv, Israel, March 2000. IEEE.

[72] Lakshminarayanan Subramanian, Venkata N. Padmanabhan, and Randy H. Katz. Geographic properties of internet routing. In *Proceedings of USENIX Annual Technical Conference*, pages 243–259, Monterey, CA, USA, June 2002.

[73] L. Masinter T. Berners-Lee, R. Fielding. RFC 3986: Uniform Resource Identifier (URI): Generic syntax, January 2005.

[74] Duc A. Tran, Kien A. Hua, and Tai Do. ZIGZAG: An efficient peer-to-peer scheme for media streaming. In *IEEE INFOCOM*, San Francisco, CA, USA, April 2003. IEEE.

[75] R. Vida and L. Costa. RFC 3810: Multicast Listener Discovery version 2 (MLDv2) for IPv6, June 2004.

[76] D. Waitzman, C. Partridge, and S. E. Deering. RFC 1075: Distance Vector Multicast Routing Protocol, November 1988.

[77] Bin Wang and Jennifer C. Hou. QoS-based multicast routing for distributing layered video to heterogeneous receivers in rate-based networks. In *IEEE INFOCOM*, pages 480–489, Tel Aviv, Israel, March 2000.

[78] Windows Media Player website. `http://www.microsoft.com/windows/windowsmedia/`.

[79] Pawel Winter. Steiner problem in networks: A survey. *Networks*, 17(2):129–167, 1987.

[80] Guoliang Xue, Theodore P. Lillys, and David E. Dougherty. Computing the minimum cost pipe network interconnecting one sink and many sources. *SIAM Journal on Optimization*, 10(1):22–42, 1999.

[81] Beichuan Zhang, Sugih Jamin, and Lixia Zhang. Host Multicast: A framework for delivering multicast to end users. In *IEEE INFOCOM*, 2002.

[82] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, University of California, Berkeley, April 2001.

[83] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant

wide-area data dissemination. In *NOSSDAV'01*, pages 11–20, Port Jefferson, NY, USA, June 2001. ACM.