Department of Computing Science

University of Glasgow

MSci Project Report

Peer-to-Peer Audio Conferencing

Stephen Strowes

Class: CS5M
Session: 2004/2005
Department of Computing Science,
University of Glasgow,
Lilybank Gardens,
Glasgow, G12 8QQ.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Stephen Strowes

Abstract

The intention of IP Multicast as a service provided by network infrastructure was to allow groups of hosts to share similar data, leaving the network to deal with the complexities of group membership and routing issues. One natural use for IP Multicast was group conferencing.

Adoption of IP Multicast has not been swift, however, leaving conferencing applications designed for use with the service unusable over significant parts of the Internet.

This dissertation presents Orta, a new peer-to-peer network overlay which is designed to allow group conferencing. The implementation is presented as a reusable software library, and is not tied to any existing application; one application, the Robust Audio Tool, is modified to use this library rather than IP Multicast as a proof-of-concept implementation. Presented are implementation details and evaluation results detailing the characteristics of the overlay, with some focus on its usefulness for real-time applications.

Acknowledgements

Thanks must go to Ladan Gharai and Tom Lehman at the University of Southern California's Information Sciences Institute (ISI) in Los Angeles for the use of one machine, kame.isi.edu, for the purposes of testing. To have one machine on the other side of the Atlantic was entirely useful.

Likewise, the Support staff here at Glasgow; Douglas for the additional machines and extra bits, Stewart for putting up with my persistent emails about sound support on the MSci lab machines, and Naveed for making the printers work so many times through the year.

Finally, Colin, for dreaming up the project in the first place, and for the nudge I needed to finally try out, install, and keep FreeBSD running at home, and all those on the MSci team who were willing to extend my project deadline after the few weeks of downtime I experienced during this term.

Cheers!

Contents

List of Figures

Chapter 1

Introduction

Peer-to-Peer systems have been used for many years in many application areas. Many Internet protocols such as FTP were designed to be peer-to-peer, though now they are primarily used for client-server style systems. Applications of peer-to-peer (P2P) have been thrust back into the limelight in recent years with the widespread use of Napster, a file sharing system intended for the sole purpose of sharing MP3 audio files. Napster, however, was only a P2P application with respect to file transfers, given that the acts of joining the network and performing a search for a file were coordinated with a central server.

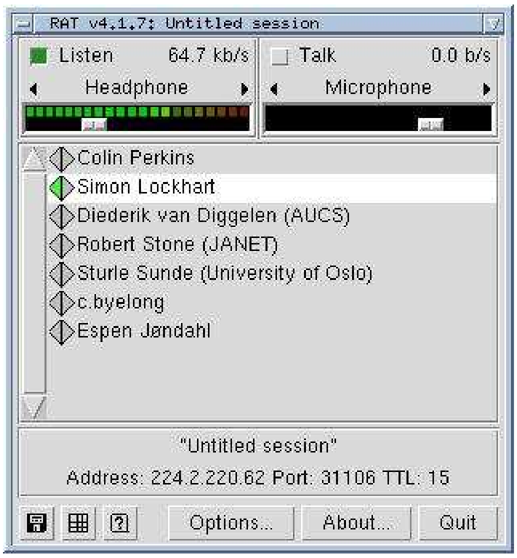
Fully distributed P2P systems can also be created, allowing for system structures to be built which avoid the need for centralised centres of control, thus removing one potential bottleneck and point of failure. Peers which communicate directly with other peers should theoretically be able to improve performance in many cases by not having to coordinate with some central point of control responsible for governing all hosts in a group communication. Applications of peer-to-peer systems are quite diverse, from file sharing [1], to media streaming [10, 30], to game playing [31].

This aim of this project was to build a P2P system on top of which could run an existing real-time group communication tool, the Robust-Audio Tool shown in Figure 1.1 (RAT, [4]); where RAT currently uses IP Multicast for communication between group members, this P2P system would be substituted. This dissertation covers previous work in this area, the approach taken by this project to tackle the problem, and the design, implementation, testing, and evaluation techniques used during the course of the project.

1.1 Group Conferencing and IP Multicast

Given the many-hosts attached to many-hosts nature of the Internet, the possibility of group conferencing in real time using audio and video streams would seem to be a natural conclusion. However, there are many problems inherent in designing a system to disseminate data between many recipients given the structure of unicast links between hosts on the Internet, and of how naïve implementations of a group communication application might be implemented.

The original design for the IP layer, responsible primarily for routing packets from source to destination, saw it as a stateless layer; it kept no knowledge of the packets passing through it, merely processing and forwarding on a packet-by-packet basis. The concept of connections, keeping orderings on packets, etc, is a



RAT v4.1.7: Untitled session

Listen 64.7 kb/s

Talk 0.0 b/s

Colin Perkins
Simon Lockhart
Diederik van Diggelen (AUCS)
Robert Stone (JANET)
Sturle Sunde (University of Oslo)
c.byelong
Espen Jøndahl

"Untitled session"
Address: 224.2.220.62 Port: 31106 TTL: 15

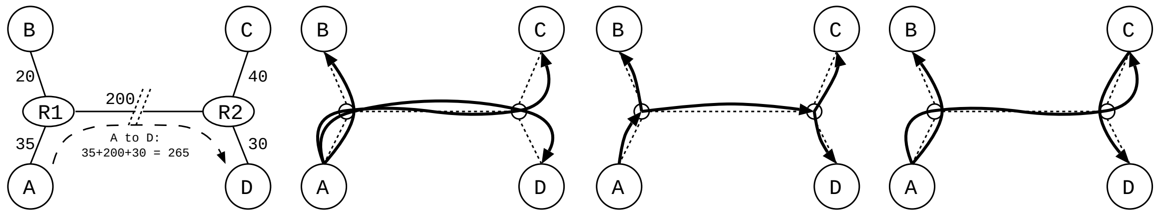
Figure 1.1: Screenshot of the RAT application.

higher level concern generally dealt with by the TCP layer, [16]. To naïvely forward data to multiple clients only using standard unicast IP links requires that multiple packets be sent, one per recipient for every packet to be sent, introducing an obvious bandwidth limitation on the sizes of groups.

IP Multicast required additions to the original IP specification, and new routing protocols, some of which are covered in [18]. The introduction of multicasting modified the initial specification by requiring that routers maintain per-group state in addition to the standard unicast routing table. In essence, members of a multicast group all share one multicast address (more information on multicast addressing can be found in [6]). Data sent to that multicast address is routed through the Internet, with packets replicated where required, to be delivered to all members of that group. The source need only transmit one packet for numerous recipients. This allows for larger groups to receive the same streaming content simultaneously.

Multicasting, despite showing early promise – in [35], for example – is unfortunately not as widespread as it was anticipated it would become. The main factor holding back IP Multicast is generally considered to be the cost of replacing existing infrastructure, the routers on the Internet responsible for dealing with the multicast traffic. There are also concerns about the scalability of the IP Multicast system (with the routers storing state for potentially many small groups, issues of time complexity arise when performing routing operations), and of higher level concerns with IP Multicast such as error checking, encryption, etc, [15]. Subscribing to a multicast group is usually not authenticated, so any host could subscribe to any group (to then receive information it is not authorised to receive, or send random data to other members of that group); indeed, any host could subscribe to all dynamically allocated IP multicast addresses, flooding the network with the influx of data to that one host [33].

One of the application types initially conceived as being viable over multicast was conferencing applications, and indeed any applications which require a level of collaboration between participants. To this end, a variety of audio conferencing applications, video conferencing applications, electronic whiteboards, etc, have been built which use IP Multicast for transmission of group data, by multicasting to all participants.



(a) (b) (c) (d)

Figure 1.2: (a) shows underlying network structure, the link R1 to R2 being a link with substantial latency. (b) demonstrates naïve unicast from A to all other nodes. (c) demonstrates IP Multicast. (d) demonstrates a possible overlay multicast configuration.

One of these conferencing applications is the Robust-Audio Tool (RAT), developed by the UCL Network and Multimedia Research Group, [3]. RAT is an audio conferencing application which uses the RTP protocol, [44], over IP Multicast, for transmission of audio data in a robust manner. Further discussion of difficulties in providing reliable audio over the Internet, particularly over the multicast-capable part of the Internet (the “MBone”), is provided in [24]. As it stands, the RAT software deals with two distinct situations: direct communication between two hosts, or group communication between members of a common multicast group. Given the slow uptake of multicast functionality mentioned above, the group conferencing features of RAT are usable primarily on research networks such as JANET[1], but generally not on commercially owned networks, rendering scope for usage significantly smaller than it might be. To allow for use of RAT over more parts of the Internet, it makes sense to replace true IP Multicast functionality by pushing multicast up to an application level overlay, the end-to-end argument [43] winning the day.

This is not a new idea. There are numerous different systems published which push multicast functionality to the endpoints of the group session ([7, 9, 10, 11, 12, 15, 20, 25, 26, 29, 30, 33, 34, 38, 41, 47, 52]), each of which tackles a slightly different intended application from a slightly different angle; these alternatives are discussed in-depth in Section 2. These systems all build some form of overlay multicast distribution network, which is almost always much more efficient than using IP unicast alone, but less efficient than IP multicast; overlay multicast incurs duplication of packets at hosts where IP multicast would duplicate at the routers between the source and the recipients, but requires that far less than the one packet per recipient be sent per cycle when using IP unicast. Figure 1.1 demonstrates the concept: Figure 1.1(a) shows the example network topology; Figure 1.1(b) shows the situation where node A is sending the same data to nodes B, C and D over IP unicast, and is forced to transmit 3 separate packets; Figure 1.1(c) demonstrates IP Multicast being used to achieve the same effect, with network routers duplicating packets where necessary; Figure 1.1(d) is a potential overlay multicast system, where node A transmits two packets, one bound for B, one bound for C, with C duplicating and retransmitting the packet bound for D.

While some of the works ([15, 9, 38, 33]) attempt to address multi-source multicast using an overlay structure, none of the papers covering these systems provide any actual data related to the performance of the systems when running in the multi-sender case, as they would be when running a conferencing application such as

[1] `http://www.ja.net/`

RAT. Indeed, all of [13] is discussion of the experience of deploying a single-source overlay multicast system, borne out of work from [14] and [15], which aimed to design for multi-source overlay multicast.

Given the lack of numerical results in the area of group communication applications over overlay multicast, it is difficult to discuss the performance of overlay multicast in one setting which IP multicast was originally designed for. We know, however, that such systems are possible to build – Skype, [2], is a group communication tool which builds some form of overlay multicast structure with which to send/receive data. This software is closed-source commercial software encased within an End User License Agreement (EULA), which offers no technical discussion on what techniques were used to build the underlying data structures, and is not standards compliant.

It seems then that there is no single solution with code available for constructing a many-to-many overlay multicast system which could be used with RAT, though some implementations of such things have been done in Java (notably [38], [20]). The aim of this project is therefore to build a system capable of application level, peer-to-peer multicast of data in a many-to-many manner, using RAT as a proof of concept. Clearly, the ideal outcome would be to produce a generic overlay multicast system which similar applications could use to multicast data when IP Multicast functionality is not available.

1.2 Dissertation Outline

The remainder of this dissertation is organised as follows: Section 2 is an in-depth literature survey and goes into detail on the background reading surrounding this project, covering such topics as conferencing and similar projects designed for conferencing-style applications; Section 3 presents the research approach taken through the course of the project; Section 4 describes the Narada protocol on which this project is based, with Section 5 covering concerns related to audio traffic carried by this protocol. Section 6 covers how the Orta protocol presented here varies from the original Narada protocol, with implementation details presented separately in Section 7; Section 8 then 9 cover the evaluation approach and metrics to be considered, then the actual evaluation results respectively. Section 10 concludes the dissertation, and covers potential future work from this project.

Chapter 2

Background and Related Work

The literature on peer-to-peer systems, particularly those focussing on multicasting, is varied. There have been many attempts at tackling peer-to-peer overlay multicasting, for various different applications in different environments (for example, the difference between a streaming video application with some infrastructure behind it, compared to a fully distributed conferencing application with no pre-existing network infrastructure to speak of).

This chapter subdivides the wealth of information available into some broad categories. Initially, in Section 2.1, some background discussion is provided around the two distinct areas of IP Multicast and of RTP and conferencing applications. In Section 2.2, peer-to-peer lookup systems are considered, as these research networks are generally reasonably widely known by researchers if not the public at large. Attempts at overlay multicast systems built on top of these lookup and routing substrates have been constructed, and so are relevant to the project. Further, these substrates provide interesting background into the organisation of peers within an overlay. There are a number of multicast systems available which are concerned with the streaming of data, but do not make use of these lookup substrates, nor do they impose real-time constraints on delivery of data as tight as those expected of a conferencing application. The ideas presented are still interesting and potentially useful, and are discussed in 2.3. Finally, fully distributed multi-source multicast systems which aim to tackle conferencing applications are discussed in Section 2.4.

The systems presented in this section have been implemented and tested to some extent, but not all have been tested by experimentation in a real network environment. Likewise, code is freely available for some projects, and not for others.

2.1 Conferencing and Real-Time Audio

On the surface, the Internet, as a carrier of data between points would appear to be ideal for the possibility of providing group conferencing abilities between hosts.

2.1.1 IP Multicast

IP Multicast was intended to extend IP beyond just offering point to point communications as was the case with standard IP unicast, to allow for group communication offered by the network, the obvious application for this functionality being group conferencing. Protocols for multicasting include PIM-DM, protocol independent multicast, dense mode [5], which works by flooding data, with routers sending 'prune' messages back up to the source to signal routers to stop forwarding if there are no listeners attached that part of the network, with PIM-SM sparse mode designed for the scaling over the Internet [19]. Other protocols are DVMRP [48], and MOSPF [36].

2.1.2 RTP and Conferencing

The Real-time Transport Protocol, RTP [44], is a protocol designed for the end-to-end carrying of real-time data. RTP can be used over IP Multicast, or over normal unicast connections. what it is, what it does. While RTP was initially designed with IP Multicast in mind, it does not require that it be run over IP Multicast.

Audio conferencing applications, such as RAT, are available which use the RTP protocol for carrying real-time audio. This application is naturally suited for an IP Multicast environment, but can be used to initiate a conference with just one other person over a single point-to-point link. Other tools available allow for video conferencing[1], whiteboard applications[2], and shared text editors[3], are all concepts for conferencing applications. The natural network environment for these applications is that of a network with IP Multicast functionality present.

RAT allows for a certain level of redundancy in the packets carried via RTP, such that the application can tolerate certain levels of packet loss [24].

2.2 Peer-to-peer systems for object lookup and routing

There are numerous research P2P systems which have been built, such as CAN [39], Tapestry [51], Chord [46], and Pastry [42], which propose naming schemes and methods of routing from one location in a namespace to other nodes in that namespace. A goal shared by all of these projects is that of scalability; being able to route a message from one host in the P2P system to any other within some reasonable number of overlay hops in relation to the number of nodes connected in the overlay network. This goal has been proven to be achievable.

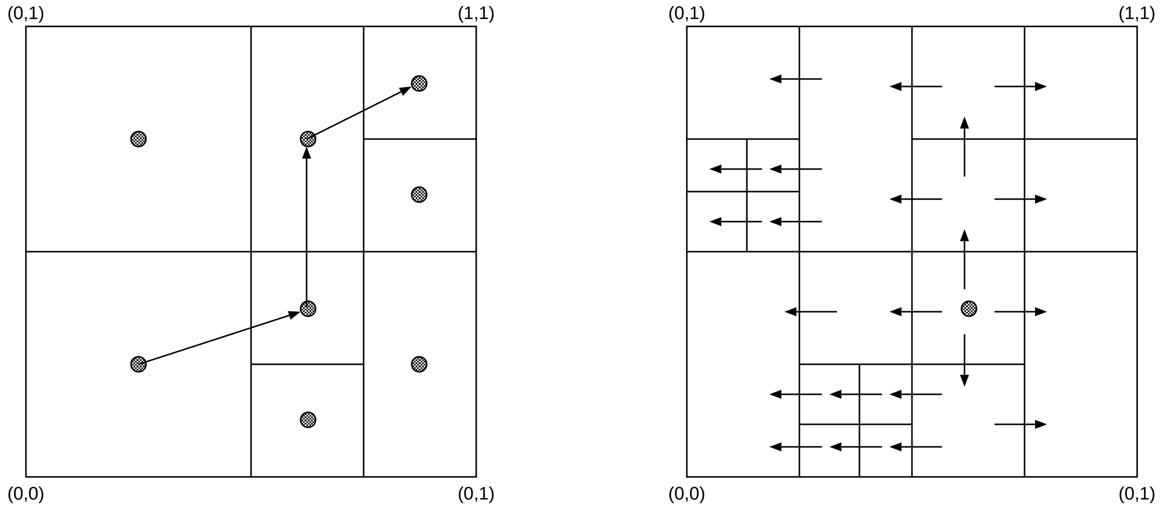
The basic idea that these schemes are exploiting is that of a distributed hash table (DHT) to efficiently arrive at the location of the required data. Nodes in a network form an overlay network, sharing some namespace. Nodes joining the overlay are inserted in the namespace according to the algorithm employed, and are able to leave the overlay. Nodes will also provide some routing mechanism for passing messages on to another node closer to the destination of that message.

In the following sections, several substantially different DHT schemes are reviewed. For each of these, attempts to incorporate some multicasting functionality into the group is also discussed.

2.2.1 CAN

The idea of a Content Addressable Network, a CAN, is presented in [39]. CAN presents a d-dimensional Cartesian coordinate space constructed on a d-torus. Objects are placed in the CAN space deterministically

[1] `http://www-mice.cs.ucl.ac.uk/multimedia/software/vic/`
[2] `http://www-mice.cs.ucl.ac.uk/multimedia/software/wbd/`
[3] `http://www-mice.cs.ucl.ac.uk/multimedia/software/nte/`



(a) Lookups in CAN space.

(b) Multicasting data over a mini-CAN.

Figure 2.1: Illustration of the hops taken to route from one node to another, and of multicasting over a mini-CAN. Rectangular regions are sections of the CAN-space allocated to a peer.

according to a uniform hash function over some key, K. An example CAN space is shown in Figure 2.1; nodes are allocated sections of the CAN space, with any keys falling into their space belonging to them. CAN nodes need then only know of their immediate neighbours in CAN space, i.e.: any node in CAN space adjacent to its own. This means that unlike IP routing protocols, only knowledge of neighbouring nodes is required, rather than some knowledge of the topology of the entire network. Routing a message from one location in the CAN to another involves attempting to take the shortest path toward the destination (in essence attempting to draw a straight line from source to destination, and similarly at each intermediate overlay hop), so the neighbouring node closest to the destination is chosen as the next hop for the message. The average path length between any two nodes in a CAN system is $O(d(N^{1/d}))$, where d is the number of dimensions in CAN space, and N is the number of nodes in the system.

Multicasting data over a CAN is presented in [41], where the stable, fault-tolerant platform of the CAN is used as a vehicle for multi-source application-level multicast to large groups of receivers (in the order of thousands, or more). Multicasting in CAN space is a simple matter of flooding the CAN, if all nodes in the CAN are members of the multicast group. If only some subset of nodes in the CAN are members of the multicast group, a mini-CAN is created over the existing CAN, and then the mini-CAN is flooded with multicast information. The coordinate space of the CAN allows for efficient flooding algorithms to be implemented which reduces the number of duplicated packets to (almost) nil (see Figure 2.1(b)). This feature of routing through the CAN space is cited as a reason why multi-source multicasting over a CAN is possible, as packets can naturally be duplicated only when required. However, larger groups will require that packets sent through the overlay are routed through more overlay nodes, forcing substantially increased latencies to nodes on the other side of the CAN space from a given source.

The work presented in [40] provides a method of adding proximity awareness to the CAN system, termed "binning" nodes into areas, the process of binning boils down to pinging known servers and assuming that

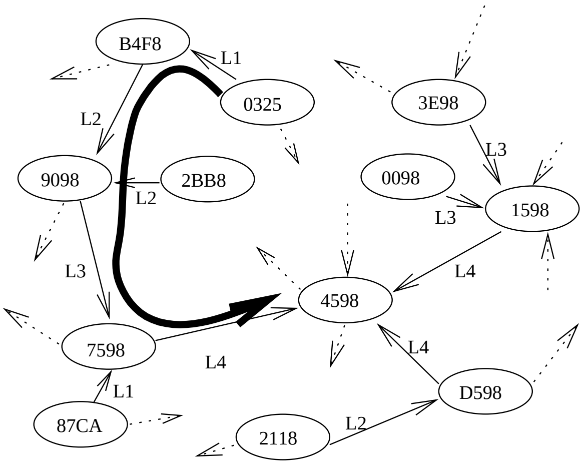


Figure 2.2: Illustration of a Tapestry network and routing across it.

two machines with similar outcomes from these pings are near to each other. Simulation of binning new nodes into areas of the CAN space close to each other based on the latency between those nodes appears to offer cross-overlay latency within the same bounds as other overlay multicast systems. This system of organising the overlay does imply that nodes near to each other in the CAN space will be reasonably near to each other in physical world, but it cannot take into account shifting network patterns. It is entirely possible for a node to be placed in the wrong bin due to unusually heavy network traffic at some known servers. Further, the formation of mini-CANs within CANs, [41], seems terribly excessive for the purpose of forwarding audio data. The lack of experimental results does not encourage confidence in a system which does not observe network conditions when considering a real-time application such as audio conferencing.

There does not appear to be any publically available code for the CAN system, so this would have to be built from scratch, following the outline of the system presented in [39] and [41].

2.2.2 Tapestry

Tapestry, [51], offers similar benefits to CAN, in that it is an overlay network capable of routing traffic efficiently through the overlay; it is completely decentralised, scalable, and fault-tolerant. Tapestry is self-organising in that it will respond to unicast links between components in the Tapestry infrastructure becoming busier, quieter, or being dropped completely or reinstated. Routing in a Tapestry system is a matter of matching as much of the destination ID to an neighbouring ID from a given node, similar to longest prefix matching used by IP routers. Neighbour maps are, however, of constant size, and each node in the path between source and destination only has to match one further digit of the Tapestry ID; an example is shown in Figure 2.2 with peer 0325 routing a message to 4598, with matching taking place arbitrarily from right to left. The distance a packet travels in Tapestry is proportional to the the distance travelled in the underlying network, suggesting that Tapestry is at least reasonably efficient.

Bayeux is a system implemented on top of Tapestry, to allow the possibility of single-source multicast over a distribution tree built using the Tapestry substrate for routing purposes, [52]. The trees can be built naturally enough – a node wishing to join a multicast group asks the root node for that group to join; the response from the root sends a special TREE message which sets up forwarding state in intermediate Tapestry routers, thus

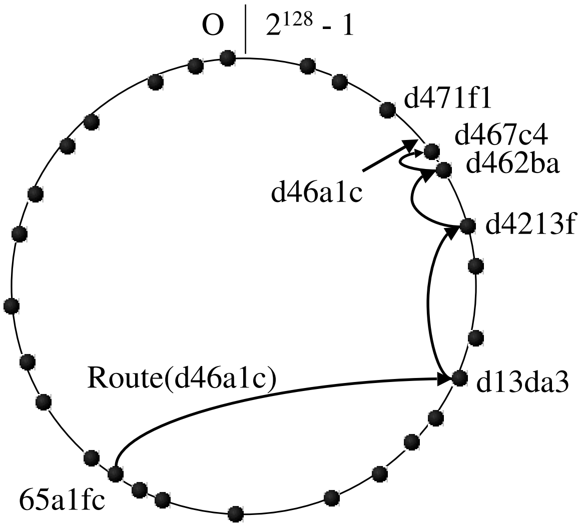


Figure 2.3: Example of routing across a Pastry substrate.

allowing for routers to duplicate packets only when necessary. Results borne from simulation have shown Bayeux capable of scaling to multicast groups in the order of thousands of members, with respect to overlay latency, and redundant packet duplication. However, there are issues with this scheme as proposed: the tree is fixed once created, it cannot be modified to take into account underlying traffic conditions and network traffic patterns, and it is designed for multicast operation with only one source. The simplest way to achieve multi-source multicast using Bayeux would be to build numerous distribution trees, each one rooted at a different multicast source. Significant modification of the design would be required to allow many-to-many multicasting in Bayeux, notably of how to inform numerous sources of the new receiver, and to allow efficient sending of TREE messages - in a large enough group with enough senders, the control overhead initiated when a member joins or leaves the group could easily flood and temporarily disrupt the overlay.

It is worth noting that while Java source code for Tapestry is available for download, the source code for Bayeux does not seem to be available, so an implementation of the Bayeux system would have to be built from scratch and modified to work from multiple sources. To integrate this into RAT then, difficult decisions would have to be taken in terms of which parts of the existing Java code to convert to C, and indeed if sections of the code could be left as they are, communicating between C and Java code sections using network sockets. This might result in a system too complex for a small conferencing application such as RAT, especially given the lack of a widespread Tapestry network to route packets.

2.2.3 Pastry and Related Systems

Pastry [42] is another peer-to-peer object location and routing substrate, where nodes each have a 128-bit node ID, node IDs define the node's position in a 1-dimensional circular node ID space. Pastry is capable of routing a message from one node in this ID space to another in $O(logN)$ overlay steps by keeping track of nodes at certain points around the namespace (e.g.: at one half, one quarter, one eighth of the way around the namespace, etc), demonstrated in Figure 2.3. Pastry is intended as a general substrate on top of which different applications can be built; an example of this generality is the fact that while Pastry nodes keep a track of nodes which are closer to them in a neighbourhood set, the locality metric used to determine just how close two nodes are is left as an implementation decision for the application itself. It is worth noting that Pastry nodes don't actually use the locality table when routing messages, though it is used to maintain the property that all entries in a node's routing table point to a node which is nearby, according to the locality metric.

Scribe is an example of multicasting over the Pastry substrate, [11]. Scribe aims to use the Pastry layer to allow for large numbers of groups, each with a potentially large number of members. Pastry is used to build trees rooted at multicast sources to disseminate multicast data. The distribution tree in Scribe is formed by using the `JOIN` messages routed from new receivers to the root of the multicast session to set up forwarding tables back down the tree to those receivers.

Pastry allows for application specific metrics to be defined to determine which nodes are close to each other on the physical network for building routing tables across the overlay network. Scribe can perform multicasting of data over Pastry, observing the "short-routes" property to determine the distance between two nodes in terms of overlay nodes, and the "route convergence" property, concerning the distance two packets sent to the same key will travel before their routes converge (this presumably makes a difference when routing data toward the rendezvous point to be multicast). While an argument could be made against still having an additional layer of abstraction in terms of node naming, a bigger hit in most application level routing systems in terms of time spent in transit by packets would be in the context switches required to pass data up from the network stack to the application and more importantly of latencies across network links; an additional, probably minor, layer of computation probably would not cause serious performance difficulties even for an application with real-time constraints. The problem with Scribe as a system for multicasting data from any node in the group to all others efficiently is that any node wishing to multicast to the group must transmit to the rendezvous node, where the rendezvous node can be chosen in a number of ways (to have the rendezvous node being the node in the namespace closest to the group ID is one method suggested in [11]). This allows for one distribution tree to be built out from that rendezvous point to all members of a group. Issues with this method are obvious: adding this extra step between sender and receivers increases latency; the rendezvous node could easily become a bottleneck especially if network resources are not taken into consideration; the rendezvous node also becomes a potential point of failure. The authors of Scribe provide results through simulation showing promise for their system; these results are difficult to accept without real-world testing, however.

Chord is a system similar to Pastry, in the sense that nodes are mapped onto a 1-dimensional coordinate space, [46]. Chord, like Pastry, achieves routing across the overlay network in $O(logN)$ hops.

2.2.4 GnuStream

While the above have been research networks which haven't necessarily seen widespread deployment outside of academia, one interesting system called GnuStream [30] was constructed using the widely deployed file-sharing network, Gnutella[4]. GnuStream uses Gnutella as its lookup substrate, and retrieves portions of a file for streaming from different sources in the Gnutella network. The GnuStream layer is responsible for collecting required data, and reordering all pieces into the correct order before passing that data up to the media player. This scheme requires buffering at the receiver, and also relies on the media source being duplicated across a number of hosts in the network to achieve appropriate transfer rates to stream the file reliably.

[4] `http://www.gnutella.com/`

2.2.5 Summary

All of these systems offer some form of multicasting functionality. However, there are issues with the systems as proposed. CAN and Chord don't take into account underlying network conditions, so cannot be trusted to provide a reliable medium with which to transfer data with real-time constraints, no matter the number of overlay hops exist between any two nodes. These systems are therefore not suitable for this application of group communication due to the fact that they do not make any correlation between overlay distance and the actual number of unicast hops between hosts, while Tapestry and Pastry do have some concept of the underlying network topology [51]. Results from experimentation in an environment as diverse as the Internet are not available, so real-world behaviour of these systems which make no correlation between overlay distance and network distance is not available. Furthermore, these systems aren't application specific to conferencing applications, and the naming schemes used are designed to allow for efficient lookups and routing across the overlay, and not the physical, network when dealing with many thousands of nodes.

2.3 Peer-to-peer systems for streaming content distribution

As well as the systems already mentioned which all generally deal with looking up data in large peer-to-peer systems and efficient routing across the overlay to this data, there are a number of projects which deal specifically with the issue of content distribution within an overlay multicast setting. Discussion of the feasibility of streaming data to large receiver groups using peer-to-peer overlays is presented in [45], and covers the distribution of large streaming media.

Sections 2.3.1 and 2.3.2 cover peer-to-peer systems designed for media streaming, with BitTorrent in 2.3.3 capable of doing the same with little modification. Sections 2.3.4, 2.3.5 and 2.3.6 then look at similar systems designed for streaming from one source, and as such perhaps better suited to a real-time conferencing system.

2.3.1 CollectCast

CollectCast, [26], is geared toward streaming content, and so is designed to select peers from a candidate set of peers holding the content to be distributed, based on available bandwidth and packet loss rate. CollectCast was built on Pastry but should be able to use any of the lookup substrates mentioned previously. The active sender set then, is the subset of those in the candidate set who can provide the highest quality playback for the receiver. Topology aware selection is presented as a means of selecting peers from the candidate set having inferred the underlying network topology. Network tomography techniques which involve passively monitoring unicast links to determine the underlying network structure, such as in [17], are used to build a model of the network structure, after which CollectCast determines how useful each link is to that receiver, generating a "goodness topology". Collectcast utilises the lookup substrate beneath it to fetch locations of required content; the implementation of Pastry used by CollectCast was modified to return multiple references to a given object, rather than just one.

PROMISE, presented in [25], is the name of the actual implementation of the CollectCast system, and demonstrates high levels of performance, and reliability.

2.3.2 SplitStream

SplitStream, [10], is an overlay multicast system also designed for content distribution, and attempts to maximise throughput by utilising the upload/download bandwidths of all participating nodes evenly (while taking into account the differing capacities of different nodes). SplitStream works by splitting the content to be distributed into a number of stripes, with each stripe having a separate distribution tree. In the ideal situation, SplitStream organises those distribution trees such that each node is only an interior node in one, and a leaf node in all others. The reasoning for this approach was the observation that tree-based systems will often generate trees which contain a core of nodes which carry the bulk of any data transfer, with a number of leaf nodes who contribute no outgoing bandwidth back into the overlay. SplitStream uses Scribe to build its distribution trees. While this system provides a very interesting way of sharing ingoing and outgoing bandwidth usage fairly at all nodes (where 'fair' is proportional to the connection type used by that host), it is not directly useful for conferencing applications. Splitting the streaming content from one source would not be feasible, this system requires that content is pre-striped across nodes in the overlay. With the natural situation in a conferencing application being that most nodes will contribute something during the conference, nodes are naturally contributing upload bandwidth anyway. The unique way in which SplitStream deploys its overlay multicast trees could be utilised by a conferencing application however, to overcome the potential issue of some nodes being interior nodes through which all traffic is routed.

Work presented in [50] tackles the problem of media streaming in a similar fashion to that of SplitStream, offering algorithms for optimal data assignments from serving peers, and admission control mechanisms based on the available incoming and outgoing bandwidth of connecting peers.

2.3.3 BitTorrent

Like Gnustream mentioned in Section 2.2, Bittorrent, [1], is an example of a widely used peer-to-peer system for distribution of data. Bittorrent is not intended as a real-time streaming platform, but the ideas presented are interesting in their own right. Bittorrent provides a means of minimising upload bandwidth used at some file's host, by utilising the upload bandwidth of those who have already downloaded (part of) that file. Groups of users downloading the same file at the same time (and subsequently sharing different parts of the same file with each other) form what is called a 'swarm' in Bittorrent terminology. Utilising the bandwidth of many hosts for uploading maximises download rates achieved; a host which is unable to upload (for whatever reason) should achieve particularly low download rates, so sharing is heavily encouraged in the Bittorrent networks. Swarms of Bittorrent clients downloading the same file generally won't last very long, so the benefits of Bittorrent can be felt by the file provider when the file has only recently been advertised as a 'torrent', and relative interest is still high. Information does not appear to be available on the technicalities of how Bittorrent peers organise themselves whilst in the swarm, though the implementation, written in Python, is open and available.

2.3.4 ZIGZAG

ZIGZAG is a peer-to-peer system designed for single-source media streaming, [47]. ZIGZAG uses a technique called chaining to ensure that data keeps moving along a distribution tree. The idea is that ZIGZAG be used to cater for streaming live media; clients need only buffer a certain amount before playback, but can pass data further down the distribution tree once it has arrived. Chaining is merely the term used by ZIGZAG to describe data passing through the distribution tree. ZIGZAG organises peers into a hierarchy of bounded-size clusters. One member of each cluster becomes the head of that cluster, and a member of a cluster at the next highest level. The multicast tree is built using this hierarchy, following a simple set of rules: a peer can only have incoming or outgoing links in the highest level cluster which it is a member of; peers at their highest layer can only link to “foreign subordinates”, i.e.: peers in clusters in the layer directly below the current layer, where that peer is not head of that cluster; other peers receive information from foreign head nodes. Proofs that this organisation leads to a tree structure are included in [47].

2.3.5 NICE

NICE is a project with the same goal of single-source multicasting, [7]. NICE constructs its trees in a very similar manner to ZIGZAG, though multicast forwarding is done differently: in NICE, the head of a cluster is responsible for forwarding data to that cluster, unlike ZIGZAG. It seems that ZIGZAG outperforms NICE, according to the results presented in [47].

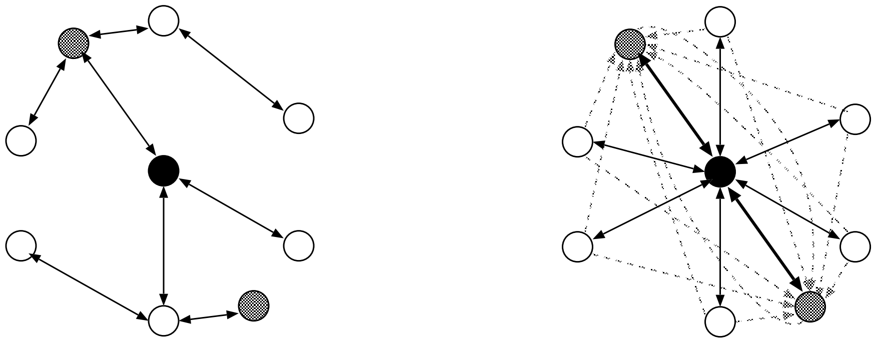
2.3.6 Overcast

Overcast, [29], is similar in style to BitTorrent, and also focuses on content distribution. It attempts to do so in a more transparent manner; nodes connect to the multicast group without knowledge of it’s existence – a normal URL is used to connect to the server, at which point the host becomes a part of the data distribution tree. The tree itself adapts to network conditions, and has been used successfully for distributing streaming video (though this is an example of streaming video with some considerable pre-buffering before playback in an attempt to tolerate unpredictable network conditions during the download, and not streaming of real-time video). This method of distributing data reduces the amount of outgoing bandwidth required at the source of data, provided there are numerous hosts downloading at the same time. Nodes which join the group evaluate their position in the tree according to bandwidth, then latency (when reported bandwidth difference between parents falls within 10%), always trying to move further down the tree away from the root, while not sacrificing too much in terms of bandwidth.

2.3.7 Summary

It seems that while all these systems are good examples of forms of peer-to-peer overlay multicast, they are not directly relevant to the task at hand. CollectCast (and therefore PROMISE), SplitStream and BitTorrent are concerned with achieving the highest throughput possible at the receiver by spreading the upload bandwidth requirements across a number of hosts in the peer-to-peer overlay. This clearly is not useful for a conferencing application. These protocols are entirely useful for streaming file transfers, perhaps even TV-on-demand scenarios, to help save bandwidth at the source of such content.

ZIGZAG and NICE form multicast trees from one source to many recipients, so the protocols would have to be modified to allow for the creation of multiple distribution trees. Overcast offers a single-source tree-based method of distributing data, but is concerned with reliable transmission of data and not transmission of real-time data. The protocol, however, is geared toward single-source multicast, and the rounds of bandwidth tests could become overbearing if multiple trees, one per source, were to be attempted for a conference group. These protocols, to contrast the others in this section, may be more suitable for ‘live’ data streams, perhaps for the streaming of live sporting events or concerts.



(a) ALMI data distribution structure.

(b) ALMI control structure

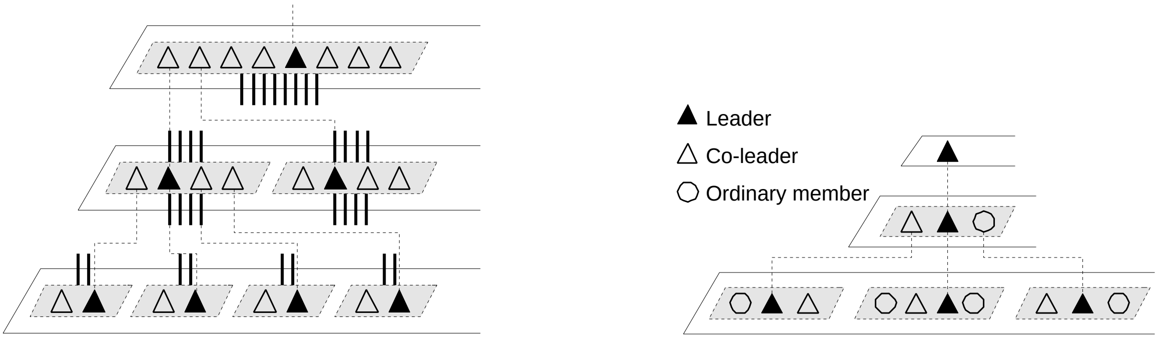
Figure 2.4: Example ALMI structures.

2.4 Peer-to-peer systems for the purposes of conferencing

The most interesting and closely related work to multi-source multicast using P2P overlays is generally split between two approaches for achieving this end-goal: tree-based approaches, and mesh-based approaches. Tree-based approaches involve directly constructing distribution trees from any source to all other group members, with members explicitly choosing their parents from the other hosts in the system of which they are aware. Mesh-based approaches instead have nodes build up a richer mesh structure consisting of many connections between different nodes, on top of which distribution trees are constructed. The use of multiple connections allows some level of redundancy when nodes fail or leave the group; further, redundant connections require that a routing protocol be run in the application to construct loop-free forwarding paths between group members [49].

2.4.1 Tree-based approaches

ALMI (Application Level Multicast Infrastructure) is an interesting project which also aims to directly solve the problem of multi-source multicast, [38]. ALMI builds one distribution tree between group members and aims for group sizes in the order of 10s of members. An example ALMI group can be seen in Figure 2.4. ALMI builds a minimum spanning tree (MST) between all members of the multicast group, which is then used to carry multicast data (Figure 2.4(a)). As well as these connections, each node has a connection to a session controller node; this controller can be run alongside an ALMI client when initiating the group. It is the controller that is responsible for organising tree structure based on information returned by group members, and deals with members joining and leaving. This controller is obviously then a single point of failure, and after such a failure it prevents nodes from joining and, more importantly, from leaving the group safely. Back-up controllers would be required to build in a level of redundancy, and all group members would have to be aware of the location(s) of these back-up controllers (Figure 2.4(b)). In the event of failed communication with the session controller, a back-up controller can be elected as new session controller. The implication is that group members must be kept informed not only of parents and children within the tree, but also of where back-up controllers are located. Further, the session controller must ensure that back-up controllers are kept up to date so that if it should fail, state held within the back-up controller is consistent with the system itself.



(a) Fatnemo tree, demonstrating wider bandwidths nearer to the root of the tree.

(b) The logical organisation of a FatNemo group, showing the the role of a leader in the group.

Figure 2.5: Examples of Fatnemo trees; diagram taken from [9].

Another tree-based multi-source multicast system is known as FatNemo, which aims to build fat-trees rooted at sources in the group communication, [9]. A fat-tree is one which links with higher bandwidths are nearer the root of the tree, with lower bandwidth links appearing closer to the leaf nodes. This is an interesting idea, and is specifically geared toward allowing multi-source overlay multicast over the tree. Nodes in a FatNemo system are organised into clusters, with clusters closer to the root of the tree being larger (Figure 2.5(a)); one member of each cluster is elected as leader, and becomes a member of a higher cluster, closer to the root (Figure 2.5(b)). Co-leaders are elected to allow some redundancy in the tree mechanism. FatNemo presents particularly good performance for multi-source overlay multicast, though no source code appears to be available for the project. The FatNemo literature unfortunately does not provide much concrete information which would be useful enough to implement this protocol.

Yoid, [20], is another tree-first approach to offering overlay multicast, but attempts to offer overlay multicast for a multitude of applications. Yoid aims to bridge the gap between IP Multicast and the abundance of overlay multicast systems developed to date. It offers a simple API similar to that which IP Multicast offers, and will use IP Multicast if it is available and appropriate; otherwise, it will organise hosts into a distribution tree, creating an overlay multicast system as and when required. This functionality is hidden behind the aforementioned API. Yoid is designed to be usable within the infrastructure used by content distributors, to smaller scenarios where the only members of a multicast group at the end hosts themselves. Yoid source code has been released and is written in C, though there are different versions available which are not compatible with each other. Yoid also requires the usage of specialised URLs to join groups, for example: "`yoid://rendevous.host.name:port/group.name`", which could solve the problem of locating groups. One criticism which could be made of Yoid is actually one of the reasons for the birth of the project: it attempts to offer one single, large solution to a vast range of problems. While this idea might seem appealing, there is little in terms of level of uptake of the software, or numerical data available to suggest that Yoid is useful in the multitude of settings it aims to be suitable for.

2.4.2 Mesh-based approaches

Work at Carnegie Mellon University has produced the End System Multicast (ESM) architecture – [13], [14], [15] – based on the Narada protocol. ESM is interesting in that it initially aimed to target multi-source multicast group situations, such as those that would be expected to use RAT. Narada employs a two stage process to building distribution trees: the first stage constructs a mesh between members of the group (more fully connected than a tree, less so than a fully connected graph); the second stage constructs spanning trees from the mesh, each tree rooted at a source within the group. Motivation for this approach was the observation that one single tree with no backup connections is more susceptible to failure, as any node failing partitions the tree, disconnecting potentially high numbers of senders/receivers from each other.

The Narada protocol calls for better links being kept in the mesh, and poorer links being rejected over time. This allows for incremental optimisation of the mesh, and therefore links available for spanning tree links to distribute data. What links are chosen depends on the metric chosen; latency is a reasonable metric for this sort of application as for conversational audio to sound natural, a round trip time (RTT) of less than approximately 400ms is required [27]. Construction of spanning trees is managed by a distance vector protocol. Trees are constructed in an identical manner to DVMRP [48]. To avoid the situation that packets are dropped when a node has left the group but routing tables have not yet been updated, a “transient forward” routing cost is advertised by the leaving node to any nodes which it a valid route to. Nodes then avoid routes involving the leaving node, but the leaving node continues to forward packets for a fixed amount of time.

Narada requires that a considerable amount of control traffic is sent regularly, to ensure the mesh continues to reorganise itself to improve performance, and to also ensure that partitions in the mesh are detected and resolved quickly. The amount of control traffic is the limiting factor on deploying Narada for use with larger groups beyond around 100 members. The results by experiment demonstrate group sizes of around 128 experiencing 2.2 - 2.8 times the average receiver delay as would be experienced over IP Multicast, using around twice the network bandwidth as IP Multicast would; these figures are significantly lower for smaller group sizes. Experimental results in [14] show that Narada places a 10-15% network traffic overhead for groups consisting of 20 members.

The source code for implementations of Narada has not been released. However, [15] provides pseudocode for some algorithms used by the system, and much description of Narada’s functionality. This information, it seems, is enough to implement the Narada protocol, as it has been used for comparison in experiments in other projects, such as the FatNemo project.

It is interesting to note that FatNemo states that it outperforms Narada, because the mesh approach in Narada will neglect how many trees are making use of that one fat link. There are a number of possible ways of resolving this. It might be possible to make the algorithm aware of how many overlay links it is placing on each branch in the mesh (perhaps by using something like Topology Aware Grouping (TAG), seen in [32]). Audio applications also open up the possibility of mixing streams together while in transit, if destined for the same host in the overlay. Beyond this, it would be fair to say that in the average case, only one group member will be transmitting, and so the number of tree branches placed on one link in the mesh does not necessarily matter; obviously the worst-case scenario where all group members are transmitting will perform poorly as [9] suggests.

There are numerous other approaches other than Narada which form a mesh structure over which to route data. Of note is the protocol described in [33], in that it creates the basic mesh structure, but does not go on to generate distribution trees within that mesh. This presents the possibility of a more naïve, brute-force implementation of connecting multiple participants in a conference, and is limited to group sizes of around 10 members.

Scattercast, [12], uses a protocol called Gossamer, based on the ESM work, which attempts to cut down control overhead, to expand to larger multicast groups. The purpose of Scattercast is to provide a large multicast infrastructure which uses SCXs (ScatterCast proXies) at known locations; these SCXs are application-level software components, often replicated allow load balancing and redundancy, which form an overlay network over IP Unicast. Clients can then connect to SCXs using IP Multicast if it's locally available, or by using normal IP Unicast links if not. Since Scattercast uses a similar protocol to ESM for organising unicast links between SCXs, control overhead is cut down by only having nodes which are sources of information actively advertising routing information to the system – if a node wishes to transmit, it advertises a distance of zero to itself, and does so for the rest of the session; neighbouring SCXs will pick up on this information and propagate it outward.

Scattercast could be implemented, upon which RAT sessions could be run. While this is a possibility, the description of the SCXs sees them as semi-permanent entities within the network; even if this is not the case and the SCXs can adapt to a quickly changing network, there is still some disparity between the SCXs and the actual endpoints, and the implication is that a RAT session would rely on pre-existing SCXs. This is not necessarily the case – RAT sessions could create SCXs for their own use, but then each SCX is an endpoint using the ESM algorithms, since all members are possible sources and so the control overhead gain of Scattercast is lost. The Scattercast solution can only ever be equal to or greater than ESM, in terms of the amount of effort to be put in to obtain a working system.

Work presented in [28] offers another method of deploying a mesh-based overlay multicast solution, by generating a hierarchy of meshes. The aim of the work was to allow for self-organising overlay networks to scale to the order of tens of thousands of hosts, and does not look at the possibility of massive conferencing applications where all of those thousands are entitled to speak. The hierarchy formed generates a mesh of lead nodes, those lead nodes being elected within a cluster, itself organised into a mesh. This form of organisation cuts down the amount of control overhead substantially from the case of one single mesh encompassing all nodes, as would happen in the Narada case.

Some form of the work on End System Multicast at CMU has been trialled on the Internet; indeed, conferences and lectures have been broadcast over the Internet using the systems they have built, though obviously this is single-source multicast model rather than the full multi-source multicast they initially set out to achieve. Skype on the other hand, is an example of a peer-to-peer conferencing and messaging system currently in use on the Internet, which also offers lookup capabilities on users within the system. An analysis of the protocols Skype uses are presented in [8]. One interesting point to note on the analysis of the Skype system is that hosts forwarding separate data streams onto other hosts further down the overlay structure will mix together data streams, thus reducing packets which have to be sent to nodes further from sources of a distribution tree. Much of the previous overlay multicast work discussed already works on the idea of replicating a packet as and when necessary for it to reach all endpoints on the overlay, but audio applications do offer this opportunity to combine data streams together, thus combining two or more packets of data into one.

HyperCast, [34], is one other addition to the multitude of systems tackling multicast groups with many-to-many semantics. HyperCast forms a hypercube from group members, group members becoming vertices in the hypercube. Spanning trees are embedded into the hypercube easily, while control traffic is transmitted along the edges of the hypercube.

Of these systems designed for multi-source overlay multicast then, ALMI and FatNemo appear to both provide the functionality required, and despite attacking the problem from very different directions, are both tree-based methods of constructing the overlays required. Yoid, while being a nice idea, unfortunately appears too large and all-encompassing. The Narada protocol seems to offer what is required of a multi-source overlay multicast system for conferencing applications, though there is no source code available. The mesh-based approach to building the overlay is, however, one which should prove incredibly robust when considering groups with high join/leave rates. Progression beyond Narada in the form of ScatterCast the hierarchical method of managing meshes should allow for the mesh to scale to far larger groups, but these are approaches to attempt after building the basic Narada protocol. The mesh-only approach seen in [33] is, unfortunately, too simplistic, only allowing for very small group sizes.

2.5 Summary

Of the related work discussed in this chapter, little is directly applicable to the problem of audio conferencing applications being tackled by this project.

The systems which implement some form of distributed hash table for the purposes of object lookup and routing are not directly relevant, but could be used as a way of locating peers in an existing overlay multicast group in order to join that group. This is outwith the scope of this project, and it will be assumed that nodes connecting to the overlay system already have some way of discovering existing group members. The method of looking up group names could be similar to the methods used by CollectCast to locate data, being able to use any appropriate lookup substrate.

The systems which deal with bulk data transfer or streaming media offer some very interesting ways of providing sustained high-bandwidth data transfers over a peer-to-peer overlay, but these are simply not suitable for the distribution of real-time data from many sources.

The projects which seem to be directly relevant are ESM (and the Narada protocol), ALMI, and FatNemo. Skype is obviously directly relevant, but its closed-source nature, and lack of technical specifications detailing the protocols, renders it of little use.

Narada, ALMI and FatNemo each have their own advantages and disadvantages, and each could potentially form the basis for this project. In brief then:

- Narada has been implemented, and has been used on the Internet to broadcast lectures, etc. The authors of Narada & ESM are, however, attempting to push ESM as a product, and have not released any actual code for the project, the only details of implementation being those in the papers released (in particular, [15]). The Narada protocol is fully distributed, and no host in a running system is more privileged than any other, though group size can be limited by the amount of control traffic required to maintain the mesh.

- ALMI, on the other hand, has released code under the GPL, in the form of a Java package, but its homepage[5] no longer exists. ALMI essentially contains one node in the group communication which is also a controller host, through which all control traffic flows. Data traffic remains peer-to-peer. Redundancy has to be built into the overlay network to allow for the possibility of the controller host leaving the group, as is the nature of peer-to-peer applications. To implement this level of redundancy seems heavyweight in terms of implementation when compared to the truly fully-distributed Narada protocol which ESM uses.

- FatNemo has been implemented, but the literature does not inform us of what language it was implemented in, or indeed where source code might be available (indeed, the authors do not appear to offer it publically as a download). FatNemo elects one node per cluster to become a leader, and also elects co-leaders for the sake of redundancy.

While the all the protocols mentioned here are relevant work, they are generally not designed for multicasting from more than one source. Of the systems described in this chapter that are, it seems that Narada, FatNemo, or ALMI are best suited to form a basis for developing a protocol designed with the specific goal of being able to carry real-time audio in mind.

[5] `http://alminet.org/`

Chapter 3

Research Approach

Based on the selection of Narada, FatNemo, and ALMI from the review of previous and related work in Chapter 2, three possible approaches to providing an overlay multicast system for use with RAT were immediately available:

1. Model the overlay multicast system on the ALMI system, rewriting the existing Java code in C
2. Model the overlay multicast system on the Narada protocol presented by ESM, generating code from scratch with reference to the outline pseudocode provided in the papers released and standard routing algorithms available in any number of algorithmics textbooks
3. Model the overlay protocol on the FatNemo system from the information available in the papers.

Of these then, the FatNemo literature does not provide much detail regarding algorithms used for construction of trees, and any implementation of FatNemo would rely on the loose concept of calculating available bandwidth between nodes. While rewriting ALMI code in C rather than writing code from scratch should prove an easier task to complete, the fully distributed nature of the Narada protocol seemed more appropriate to the project, more challenging, and less dependent on particular hosts in the system surviving (and is therefore, potentially more robust) than either ALMI or FatNemo. For these reasons, a re-implementation of Narada was chosen as a starting point, with some consideration taken at a later stage as to how to improve the protocol.

With the Narada protocol in mind at start of development, the following steps were taken to fulfil the project aims:

- The overlay API was defined by analysing the existing code, and determining just what functions were used for the purposes of constructing a starting point from which to work. By defining this API, development took place in the space between this API (building downward from the functionality required by the application) and the standard network libraries available (building upward from the functionality provided by the network). This process produced a clean API for the functionality required.
- Work began on implementing the Narada protocol to build a mesh structure in order to test underlying data structures used by the overlay code. Performance of these data structures was not necessarily a primary goal; creating a system which worked first, from which may be subsequently optimised is more important. Once the data structures required and network code was apparently stable, work began on modifying the method in which control traffic is distributed throughout the overlay.

- Implementation of the modifications took place, taking care to alter all relevant Narada code. Once the overlay was stable, it was possible to construct a concrete evaluation plan, the outcome of which can be seen in Chapter 8. Evaluation of certain key metrics to evaluate both that the protocol works, and that it offers properties desirable for real-time applications.

- Integration of the implementation of the modified protocol into RAT using the overlay to carry data, rather than using the network directly, was carried out. This allowed RAT to treat the overlay as an intermediate layer between the network and the application, and served as a demonstration of the application of this overlay multicast software in a "real-world" application. To focus development work, only calls which RAT makes that involved the transfer of real-time data across the network were modified to use the overlay; the main components of RAT communicate using a message bus (mbus, [37]) system, which also makes multicast calls. These calls were left intact, for the purposes of testing. Patches are provided in Appendix C.2.

The end result then is an open implementation of a new protocol called Orta, which is a variant of the Narada protocol. This implementation is a demonstration of the algorithms used to create the mesh, and then the distribution trees on top of that mesh. The implementation offers functionality to allow hosts to join the overlay group, leave the group, and send/receive data to/from all other members of that group; this functionality is general enough that any host can attach itself to the group simply by identifying at least one existing group member and sending messages to it, asking to join. Once a known member of the group, the peers attempt to organise themselves to improve the quality of the mesh. This implementation is provided as a library, which RAT or any other application can use at will.

To help focus the coding for the project then, aside from leaving the previously mentioned mbus code used by the three main software components which constitute the RAT application, the coding will also focus only on IPv4 functionality, rather than offering IPv6 support. Further, issues raised by NATs and firewalls, [21], will not be considered for this implementation – the focus of the project is on the construction of the overlay multicast network, not the details of bypassing hosts sitting behind a NAT on a private network. The development will also assume that peers have some mechanism in place for locating an entry point into the multicast group, such as some of the lookup mechanisms described in Section 2.2. Development will be taking place on Linux workstations, and so code will primarily be tested on Linux; to attempt to implement this system in C for all manner of platforms within the time constraints available would most certainly be foolhardy.

Chapter 4

The Narada Protocol

This chapter briefly covers the Narada protocol on which the Orta protocol, presented in Chapter 6, is based. The Narada work forms the basis of the ideas used in this project, the major difference being how the protocols share enough information to be able to hold enough knowledge about the overlay such that reliable routing tables can be generated. Further and more detailed information on the Narada protocol is available in [15]. This chapter covers the construction of, and gradual improvements made to, the mesh, and the data delivery methods.

4.1 Properties of Narada

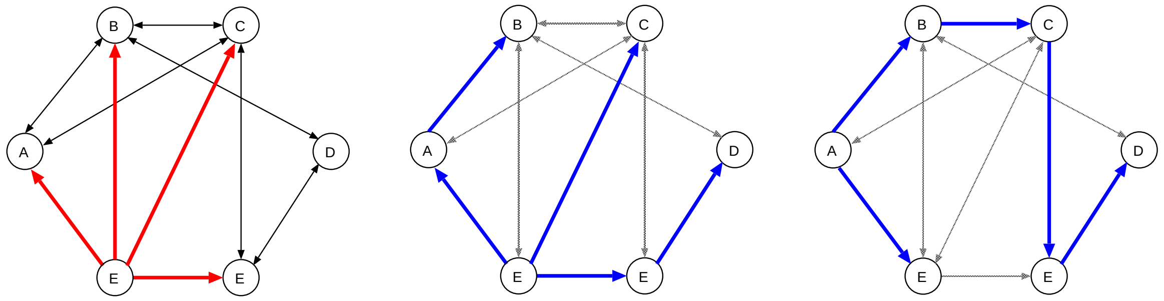
The Narada protocol is designed to be self-organising and self-improving. That is, the protocol is fully distributed, and the peers in the network must be capable of observing certain network performance metrics in a bid to enable peers to gradually improve the state of the overlay. Which metrics are observed are described as "application specific" though, for real-time communications, latency is the major metric to measure between peers due to timing constraints mentioned in Section 2.4.2. Available bandwidth is also an issue, once transmission quality, the potential audience (and their potentially varied connection types), and the size of the group are also considered.

Each peer tries to improve the quality of the overlay by judging the quality of the links it currently holds and the quality of potential links which could be added to the overlay. The actual adding and removing of links is defined by the Narada literature, and reiterated here in Sections 4.5.1 and 4.5.2.

4.2 Overlay Construction

Narada uses a two-step process to construct the spanning trees required to carry multicast data efficiently across the overlay. The first step involves the construction of a richer graph between nodes than that of a spanning tree; this structure is termed a *mesh* in the Narada literature. The mesh allows many connections between hosts in the group, and attempts to improve itself to provide "desirable performance properties". The second step involves running a routing protocol over the mesh to construct spanning trees from each source, thus allowing for the possibility of group communication.

The motivation for the two-step process is to allow more reliable overlay structures for multi-source ap-



(a) A mesh structure, showing the members E considers to be it's neighbours.

(b) A potential spanning tree from group member E.

(c) A potential spanning tree from group member A.

Figure 4.1: Illustration of Narada terminology.

plications, such as audio conferencing. While it would certainly be possible to construct an overlay capable of carrying data from one host to many using either one shared distribution tree built directly from knowledge of group members, or many distribution trees built directly from knowledge of group members, there are drawbacks to this approach:

- The construction of one tree is prone to failure, as it only takes one node failure to disrupt the tree structure.
- One solitary tree is not optimised for all participants.
- The construction of many trees requires additional overhead in terms of group membership when adding nodes to trees, removing nodes from trees, etc, as hosts join or leave the multicast group. The additional overhead is a side-effect of the trees themselves being separate, and therefore each requiring individual management.

The two-step approach of constructing a mesh and then multiple distribution trees allows the mesh layer to handle group membership and other problems such as how to optimise the mesh or how to repair a partitioned mesh structure, with the routing protocol running independently on top of this layer. The additional links between nodes allows for each distribution tree to potentially be of a higher quality for the host to which is belongs than a single, shared spanning tree would be.

4.3 Terminology

For the sake of clarity, a number of key terms to be used for the remainder of this dissertation shall be defined here, with reference to Figure 4.1.

Figure 4.1(a) shows a potential mesh structure, with many connections between peer. These are logical connections, and not the underlying physical links in the network. A connection between two peers within the network can be considered to be two unidirectional links (owing to the fact that the route a packet takes to get from A to B might not be the return path from B to A. Thus, the term ‘link’ refers to one direction over a TCP connection; when the actual physical network structure is being discussed, links will be qualified as either physical or logical. The terms peer and member may be used interchangeably, though on occasion a distinction may be made between a source and recipients; a source is simply one member, and the recipients are all other members of the group.

Figure 4.1(a) also highlights which peers E considers to be it's neighbours. Spanning trees, such as those in Figures 4.1(b) and 4.1(c) are built from the links between members and their neighbours, and peers certainly do not share the same spanning tree for data distribution.

The term neighbour implies that if one node, a, has chosen another, b, as it's neighbour, then a is also b's neighbour. Indeed, this assumption allows for information required to build correct routing tables at each node and all others to be passed back and forth between a new member of the mesh.

4.4 Group Membership and Overlay Maintenance

The Narada protocol is fully distributed, and as such, no single peer is solely responsible for maintaining group membership data; this responsibility falls onto all peers in the group. While this allows for a higher level of redundancy of data, peer group sizes are surely bounded by the capacity of hosts to store some information on all hosts in the peer group. Sharing this burden limits group sizes by requiring that state be sent between and stored at end hosts, which brings with it concerns of storage space and the efficiency of data structures holding the information.

Each member regularly signals to its neighbours within the mesh that it is still alive by means of a refresh packet, which contains a sequence number. The sequence number is a simple method of having other peers in the network log the last time they received any information relating to the source of that sequence number, and can be used to resolve partitions in the mesh structure, as will be discussed in Section 4.4.3. These refresh packets are also used to carry the routing information required in Section 4.6 on data delivery.

Members continuously probe their neighbours, to monitor network conditions and advertise correct weights on links.

4.4.1 Members Joining

Narada does not provide a lookup service for the finding location of any group member a new peer might be able to connect to; the protocol instead assumes that peers wishing to join the group have some method of locating at least one group member with whom the exchange of control information can begin. As Narada is fully distributed, which member is chosen for joining is irrelevant, as any existing member can admit entry to any joining peer. This contrasts with IP Multicast, where one address is used for the entire group, with routers internal to the network admitting new members.

Information regarding the new member will be propagated to the rest of the group via the normal mechanism of regular refresh packets. Given that the peer group will learn about the new member slowly rather than immediately, it will take some time until the entire group is aware of a new member and, likewise, approximately the same amount of time until the new member is aware of the rest of the group. If we consider that refresh packets might be generated every 30 seconds, and the shortest path between the new peer and some others might be four hops away, in the worst case it might take almost 2 minutes for those members to learn of the new group member (though generally, the highly connected nature of the mesh suggests that this would not be the case).

Once a member is part of the group, it may commence procedures to improve the quality of the resulting mesh structure.

4.4.2 Members Leaving

On a member leaving, that member informs its immediate neighbours, who subsequently inform other group members by means of the regular refresh packets. Remaining group members restructure their routing tables on receipt of the new information. As before, this process could take some time.

In a bid to try and counter the time taken for the information to propagate throughout the group, old routes continue to be used until such a time as the routing tables throughout the group converge. Coupled with this, old routes are advertised by a “transient forward” value, which is guaranteed to be higher than any weight, or latency, achievable by a real link, yet also lower than the infinite cost which signifies no link is in place. The nature of the distance vector algorithm then allows for the group to naturally avoid the links advertised with a transient forward value.

4.4.3 Member Failure

The Narada protocol takes into account the possibility of members failing. The sequence numbers mentioned earlier form a part of this process, by allowing each peer to log the last time they received any control data from each other member of the group. If a member were to fail without exiting gracefully (i.e.: without dispatching the appropriate ‘member leave’ information to allow for transient forward links to be advertised, and to allow for the continued forwarding of data for some acceptable amount of time), these sequence numbers would fail to be generated.

Narada ensures that the mesh is capable of repairing itself, by checking the sequence number for each known group member on a regular basis, and running Algorithm 1, shown on Page 26. In essence, the algorithm guarantees that a host which has been silent for some upper bound on time elapsed will be removed if no connection can be made to it; a host which has been silent for at least some lower bound on time elapsed might be probed, and possibly removed thereafter if no link can be made.

4.5 Improving the Quality of the Mesh

Narada peers constantly monitor the quality of the links they have to all of their neighbours. How they do this is dependent on what metrics are chosen by which to differentiate a good link from a bad link, but sending regular ping packets to neighbours is a method of measuring latency between peers, for example. The following outlines exactly how Narada chooses which links become part of the mesh structure, and how links are removed from this structure.

4.5.1 Adding a Link

In order that the quality of the mesh improve over time, peers must attempt to seek out links between each other which offer some significant improvement for the data trees, or alternatively add links to help ensure that a partition in the mesh structure is less likely to occur. Thus, there must be some mechanism in place

Algorithm 1 Algorithm used by peer i to detect and repair partitions in the mesh structure.

```
Let $Q$ be a queue of members for which $i$ has stopped receiving sequence number updates for at least $T_{min}$ time.
Let $T_{max}$ be maximum time an entry may remain in $Q$.
while true do
  Update $Q$;
  while !$Empty(Q)$ and $Head(Q)$ is present in $Q$ for $\geq T_{max}$ time do
    $j = Dequeue(Q)$;
    Initiate probe cycle to determine if $j$ is dead, or to add a link to it.
  end while
  if !$Empty(Q)$ then
    $prob = \frac{Length(Q)}{GroupSize}$
    with probability $prob$ do
      $j = Dequeue(Q)$;
      Initiate probe cycle to determine if $j$ is dead, or to add a link to it.
    end
  end if
  sleep(P). // Sleep for time P seconds.
end while
```

which probes potential links, over and above the continuous probing already carried out on existing links to monitor network conditions.

Narada achieves this by randomly selecting peers within the mesh to which the local peer is not connected, and probes those members to determine the properties of the links between them. If latency is the metric the protocol is concerned with, this is achieved by sending them a ping packet; required as part of the return data for this ping is the routing table of the other peer, which is used to estimate the usefulness – or utility – of adding this link. Algorithm 2 shows the utility calculation. Each peer in the group can contribute between 0 and 1 to the final utility score.

Algorithm 2 Evaluate Utility of link L

```
utility = 0
for each member, $M$, such that $M \neq localhost$ do
  $L_n$ = new latency to $M$, with $L$ in place
  $L_c$ = current latency to $M$, without $L$ in place
  if $L_n < L_c$ then
    utility+= $\frac{(L_c - L_n)}{L_c}$
  end if
end for
if $utility > threshold$ then
  add link $L$
end if
```

A new link will be added to the mesh if the perceived gain is greater than that of some threshold; the details of this threshold value are not disclosed in the Narada literature, beyond stating that it is to be based on group size and number of neighbours each member at either end of the link has.

The threshold value calculated at a node A is simply defined as relating to the number of group members A knows about, and the number of links both A and the other member has. No more detail is provided for the threshold, but it makes sense that the threshold would be proportional to the group size (as each link can potentially provide up to 1.0 of the utility value), and also proportional to the number of neighbours each of the two nodes have (so that once a member is well-connected amongst the group it becomes harder for it to add low-quality links, while members with fewer links may add new links with greater ease).

4.5.2 Removing a Link

Peers using the Narada protocol do not have enough information directly available to them to derive a value for the usefulness, or utility, of an existing link like they do for the addition of a link. This means that Narada must instead provide an alternative mechanism for dropping links.

Each member attempts to evaluate how valuable its links are to the group by looking at the number of members, including itself, which make use of this link to distribute their data. Algorithm 3 outlines the algorithm used by Narada to derive what it calls a *concensus cost* for the link.

The requirement on such an algorithm is that the utility lost on dropping a link is the same as if the link were to be added, given all other network conditions remaining stable. Without enough information though, Narada cannot judge this perfectly, and simply drops the link with the lowest concensus cost beneath some threshold. This threshold is different to the threshold used to add links.

As discussed in Section 4.4.2, owing to the fact that on the removal of a link it will take some time for other peers to be made aware of the state change, Narada employs a transient forward state for a link, where a peer will continue to forward packets along that link for some reasonable time to follow.

Algorithm 3 Evaluate Concensus Cost from local peer i to remote peer j.

```
  $Cost_{ij}$ = Number of members for which $i$ uses $j$ as next hop for forwarding packets.
  $Cost_{ji}$ = Number of members for which $j$ uses $i$ as next hop for forwarding packets.
  return maximum of $Cost_{ij}$ and $Cost_{ji}$
```

The threshold for dropping a link is always less than the threshold would be to add the link. This is, in essence, a simple hysteresis technique to ensure stability in link addition and removal decisions, and helps avoid the situation that links are repeatedly added and dropped by the same member. By taking members on both sides of the link into account, the mesh also ensures that one peer attempting to add a link will not be repeatedly thwarted by the neighbour dropping it each time.

The dropping mechanism should not cause a partition provided the network is stable, and the threshold for dropping a link is less than half of the group size. The reasoning is that if the link being considered is the only link holding together two halves of a mesh, then either $Cost_{ij}$ or $Cost_{ji}$ will be greater than half the group size, or both will equal half the group size. The dropping links algorithm obviously cannot handle the situation where multiple links are dropped simultaneously, though the mechanisms for fixing a partition as described in Section 4.4.3 can be used to fix such a partition.

4.6 Data Delivery

Narada runs a distance vector algorithm over the mesh, and calculates the distribution tree for each source using reverse shortest path between each recipient for each source, as is done in DVMRP [48]. The distance metric advertised via the distance vector algorithm is that of the weight of the link derived by whatever application specifics are required (e.g.: latency), rather than simply the number of hops from source to destination. While conceptually the routing algorithm could be viewed and implemented as an entirely separate entity from the mesh structure upon which it runs, it makes sense for the distance vector information required to be passed between peers to be sent as part of the regular refresh packets Narada defines. Thus, the very process of propagating routing information as the distance vector algorithm requires is the process by which members are able to monitor the liveliness of the other members.

Narada's data links use TFRC [23], a rate-controlled UDP, to carry data. This allows for data rates which are friendlier toward TCP connections on the same link.

4.7 Summary

Narada is a protocol which attempts to gradually improve the quality of a mesh of links, upon which multiple distribution trees (one per source) can be built. Routing tables are built via a distance vector algorithm. The method in which it selects new links in an attempt to improve the quality of the mesh, and attempts to create multiple links from any host, offers the strong benefit for a real-time conferencing application that peers leaving a group are unlikely to split the mesh, thus reducing the chances of breaking a stream of data.

It appears, however, that by the choice of routing algorithm Narada makes, the group might not be very responsive to changing group state.

Chapter 5

Constraints

Since the purpose of the project was to deal with a real-time audio conferencing application, some thought into the constraints this places on the design of the overlay protocol is required.

As mentioned briefly earlier in Section 2.4.2, 400ms is considered to be an approximate upper bound on the round trip time of audio packets in a communication to allow for conversational audio to be feasible [27]. Narada is designed to attempt to find the shortest path spanning tree from each source to all recipients in the group, and so is should to be able to find paths which offer less than a 400ms RTT if one exists.

However, owing to the routing mechanism Narada uses, should network conditions change during the lifetime of the group, the group members may have to endure longer RTTs until the mesh is reconfigured.

The 400ms limit should not only consider raw RTT values, however. If we are to consider real-time audio as the data the overlay shall be carrying, there are additional constraints to meeting that 400ms upper bound. The sender waits for around 20ms to grab an audio frame, and may take a few milliseconds to encode that frame. The receiver may buffer this packet for a few milliseconds to take into account jitter on the input stream of packets, and again for the decoding. With all these taken into account, the additional delay may be anywhere between approximately 50ms and 70ms, and therefore the raw RTT to be met must be less than 350ms or 330ms respectively. To ensure that these targets can be met, the overlay must surely be able to react quickly to network conditions changing.

Swift reorganisation in the face of member departure: members who rely on the leaving group member for packet delivery require that the overlay restructure as quickly as possible, with as little disruption to the packet stream as possible. While for some applications such as file transfers, a temporarily broken data stream is not an issue (provided enough of the peer group remains intact). A break in the tree means that some participants may be disconnected to others leading to a break in conversation and loss of information.

Similar to that above, low packet loss rates would be desirable, so reorganising the overlay structure must be performed in a timely manner. No packet loss due to reconfiguration of the network is preferred, though occasional packet loss is acceptable when compared to longer bursts of packet loss.

The Narada protocol provides some good properties for the purposes of conferencing applications. The gradual improvement to the mesh by adding better links and removing poorer links, and the building of a distribution tree rooted at each source, suits the needs of a conferencing application very closely.

However, the distance vector algorithm used for routing means that information regarding members and links propagates slowly through the group. The choice of a distance vector routing algorithm which does not require link state to be held at each group member leads to a link dropping mechanism which is not working on total knowledge of the group, and indeed only works by dropping links conservatively. The nature of the distance vector algorithm, and the fact that the distance vector of another group member is returned in response to a random ping, means that the algorithms that Narada uses are working with potentially out of date information. Further, the threshold by which a new link is judged for adding, and the threshold by which an existing link is judged for dropping, are entirely different to one another, with very different mechanisms in place for the addition and removal of links.

To resolve these issues, the Narada protocol required modification. The Orta protocol, presented in Chapter 6, modifies the routing algorithm and the methods used to deal with control state, which attempt to solve some of the problems with Narada, and allows the addition and removal of links to be governed in the same manner.

Chapter 6

The Orta Protocol

This chapter introduces Orta, a new overlay routing protocol for real-time applications. Like Narada, Orta is a fully distributed, self-organising protocol which attempts to improve overlay quality during the lifetime of the overlay. Orta, however, distributes control information in a different manner, to inform the entire group of state changes as quickly as possible.

Where one of Narada's failings is the use of a distance vector algorithm to slowly propagate information, Orta instead uses link-state routing to maintain link state at each peer, over which a shortest path algorithm can be run. By changing to a link-state routing mechanism, state changes at any group member are flooded to the entire group, so all group members are informed as quickly as possible about the state change and can react accordingly.

This rather major modification to how peers interact also allows for the algorithm responsible for dropping links to be changed, to mirror that of the algorithm designed to add links. Measuring the utility of a link by the same mechanism on both adding and dropping a link allows for the same threshold calculation to be used, and should offer more reliable decisions made by the link dropping mechanism.

6.1 How Orta differs

The key feature of the Narada protocol is that over the lifetime of the multicast group, links can be added and removed such that performance of the mesh is improved. The choice of running a distance vector algorithm over the mesh, while simple to implement and run, does not allow for quick propagation of information relating to network state changes.

The new protocol presented here uses the pseudocode presented in the Narada literature for operations such as evaluating whether or not a new link is worth adding, but instead uses a link-state protocol to flood information on new links out to group members. Dijkstra's shortest path algorithm is then run over the link state information which should be the same at each node, to provide optimal data distribution trees given the current mesh structure. Any state changes which might cause distribution trees to change should be flooded, i.e.: a new member arriving; an existing member leaving; the addition of a new link; the removal of an existing link; or the updating of the weight of an existing link.

While this does mean more computation taking place at each peer compared to what is required in Narada, the rate of propagation of information provides much more information for each node which can then be used to provide a more robust network structure for the carrying of multicast data. The flooding nature of the protocol means that peers will be less likely to arrive at differing states capable of creating loops in the network.

The link-state protocol then operates as follows:

- New information is propagated on arrival of a new member, on a member leaving, on the addition of a new link, and on the removal of a link.
- In order to keep peers informed of current network conditions, each peer continuously probes its neighbours to determine the distance between itself and each neighbour, exactly as is done in Narada. On a regular basis (every 30 seconds, say), each peer floods information regarding significant link distance changes between it and its neighbours.

On receipt of any of these flood packets, a peer runs Dijkstra's shortest path algorithm using the link state it has to generate a shortest path spanning tree from each group member to generate the routing table at each peer. This can clearly lead to a lot of computation being performed at each peer, though ideally network conditions would not change all too frequently; if no state has changed, no computation has to be done.

6.2 Overlay Construction

In the same way as Narada does, Orta uses a two-stage process to build the overlay structures required to route data from any source in the group to all receivers. Once again, the motivation of this two-stage approach is to allow for distribution trees optimised for each source, and to allow for a greater level of robustness than a single shared distribution tree can offer.

Orta adds and removes links in a bid to improve the quality of the mesh, like Narada. The key difference between the two is of how control traffic is distributed, and what control data is distributed. The additional information available to each peer as part of the link-state routing protocol allows for Orta to calculate its spanning trees in a very different manner to Narada.

6.3 Group Membership and Overlay Maintenance

As in Narada, the burden of maintaining group membership falls on each member of the group. Orta requires that each member monitor its own links, and flood information to the rest of the group regarding those links. Thus, in Orta, members collect current information about each link in the mesh on receipt of floods from other members, and so there is the additional burden of each host having to store all this additional information.

Where Narada peers, by using a distance vector algorithm, send refresh data containing known information about all group members only to local neighbours, Orta instead sends only information relating to links to local neighbours to the entire group by flooding the information. Thus, the peer responsible for probing an existing link in the mesh is always able to provide the group with the current information regarding the state of that link. This method of propagating control information allows for:

- Dijkstra's shortest path algorithm to be calculated over the link state to derive routing tables at each member, covered in Section 6.5.

- Faster reactions to changes in network conditions, owing to the nature of the propagation mechanism.

Under normal operation, new information carried in a refresh packet is merely updated information on an existing link or member, and is incorporated into local state. Orta must also be able to include information relating to unknown members and links, due to the fact that during any periods in which the mesh is partitioned, state changes on one side of the partition will not be observed by members on the other. Thus, peers must be able to absorb new knowledge as it arrives, to ensure that the mesh will be able to gradually repair itself.

For each peer to send information on each link it owns at every refresh cycle seems wasteful, if the state of that link has not changed recently. If a link has not changed since the last cycle, a peer can opt to not send information regarding that link. Members must still, however, send a refresh packet containing a sequence number as normal. It might be the case that on many refresh cycles, little more than the sequence number is flooded for any given group member. Each member must still forward information on each link it owns, even if at a reduced rate, for the same reasons as mentioned above: to allow for a gradual repair to take place if the mesh becomes partitioned and link state changes at any point during that partition.

To aid the flooding algorithm, looping of flooded packets is avoided by having flooded packets carry the incremented sequence number from its source. This simplifies the handling of flooded packets somewhat, even though all operations over control state should be idempotent; for example, receiving two copies of a packet to add a link from member A to member B should not result in two link entries for A → B. By ensuring that a sequence number received is not forwarded if it has already been acted upon, no flood operation should erroneously continue to cycle around the network.

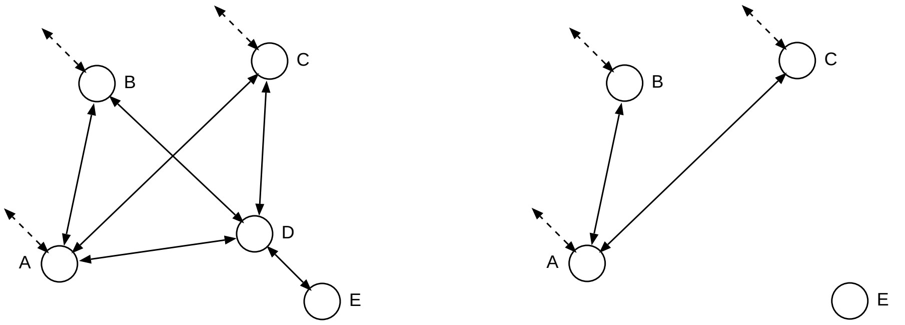
As with Narada, peers are required to constantly probe each other to ensure that the advertised link state is a good representation of the actual link state. If, however, Orta were to flood a state change on every link as soon as that state changed even marginally, the mesh would become flooded with constant state updates. (Ironically, the state updates would likely disrupt the current state of other connections, prompting those connections to re-advertise their status, and so forth.)

To this end, Orta declares that there be a distinction be made between the actual weight, and the advertised weight, of a link. The actual weight of a link is the current latency observed over a link, while the advertised latency is a recently observed latency on that link. The advertised latency need only change when the difference between the actual latency and the advertised latency is sufficiently large.

6.3.1 Members Joining

Like Narada, Orta does not concern itself with lookup mechanisms to locate an existing group member. Location of a group member is assumed to take place via some other mechanism (be it a centrally located lookup service, a fully distributed lookup service such as those discussed in Section 2.2, or using an existing members advertised IP on the Web). Being fully-distributed, any Orta peer is capable of admitting entry of a new peer to the group.

On entry to the group, information on the new member and new link is flooded to the entire group, thus allowing for distribution trees to immediately take into account the new member, even though the initial link chosen might not be the best possible for the group. Once a new member is admitted, mechanisms for improving the quality of the mesh take over.



(a) A functioning overlay structure. E has only one neighbour in the group.

(b) Member D has left the group, and peers have cleaned up their local state as required. E is detached from the group.

Figure 6.1: Illustrative example of packets lost due to a peer exiting.

In order to allow the new member as much information as possible to integrate it into the mesh quickly, on a successful join the existing member should send the new host both member information and link-state information; this information should allow for quick integration into the mesh network structure.

6.3.2 Members Leaving

On leaving, a member informs the group of their departure and also of any links which will be affected by this. Again, this allows for the peers to immediately be able to restructure their distribution trees such that the new mesh topology immediately comes into effect.

Unlike Narada, Orta does not specify a transient forward state for a link, nor is a peer required to continue to forward packets along a link for some reasonable length of time. If the mechanisms which deal with adding and removing links to the mesh have not created a mesh which is well-connected enough, the departure of a peer can then cause problems, as illustrated in Figure 6.1. We can see in Figure 6.1(a) that the mesh is well connected. At first glance it appears that to have only one link to E, creating a leaf node, is not a problem provided no member joins through E to create a chain of peers. As Figure 6.1(b) demonstrates, this is not necessarily the case. Member E is entirely dependent on member D, and as D leaves the group, it becomes detached. It is unable to route to the rest of the group, and vice-versa. The mesh then relies on mechanisms described in Sections 6.3.3 and 6.4.1 to reconstruct the mesh.

6.3.3 Member Failure

In the event of a member failure, state relating to that member will linger in each other member's local state, potentially leading to data loss. Orta employs the same mechanism as Narada does for detecting and repairing partitions in the mesh structure, as detailed in Section 4.4.3. Only when a member declares this failed member to be dead will all group members be able to remove the relevant lingering state. On detecting a failed member, Orta requires that the peer who discovered the failed member flood the same information as for a member leaving the group, as per Section 6.3.2, but instead using known group state to flood information about the failed member, and any links it held. This should happen on both sides of a partition, thus clearing any state relating to the failed member at all remaining peers.

By removing the affected peer and updating the local state at each peer via the normal mechanism of peers continuously flooding refresh messages, the group can reconfigure itself over time and generate correct routing tables from the new mesh structure for the distribution of data packets.

6.4 Improving the Quality of the Mesh

As in Narada, the key element of the Orta protocol is that links are added and removed in a bid to improve the quality of the mesh based on some metric. The mechanism employed in Orta for dropping links is entirely different to that of Narada, and is shown in Section 6.4.2.

6.4.1 Adding Links

Adding links takes place in the same way as it does in Narada, except the return payload of a ping packet to a randomly selected peer does not include distance vector information as Orta peers maintain all the link state the algorithm for adding links requires. Orta uses exactly the same algorithm as that defined in Section 4.5.1 to determine whether or not a link should be added to the mesh, but an Orta peer has the advantage of having all link state available to it, plus the reported latency to this other peer, to allow it to determine the latencies to all other members with or without that link in place using the same shortest path algorithm used to create the routing tables.

As before, the method of evaluating the worth of a new link is calculated from a local perspective; only the current node is taken into account in the hope that the addition of this new link will benefit the mesh as a whole. The utility of a link is essentially a measure of how much that link improves the quality of the mesh; once again, the utility of a link lies in the range $0..1$, with 1.0 being the highest attainable utility.

On adding a link after receiving a favourable turnaround time to a ping packet, the round trip time on that packet is used as the latency for the new link, being the only available estimate of the weight of the link. The assumption here is that the latency reported by the random ping will be close to the average latency observed over time. By advertising the link with this latency, we avoid the need to advertise as some arbitrarily large weight initially, only to have trees reconfigure a second time when the actual latency over time is observed and advertised at the next refresh cycle.

6.4.2 Removing Links

Given that Orta peers store complete link-state information for the peer group, there is enough information available locally at each peer to calculate the utility of a link by running a modified version of the algorithm used to add links. This is in stark contrast to Narada, which uses a separate measure of utility for links when calculating whether or not to drop a link.

Algorithm 4 (page 36) outlines the actions Orta takes to determine the usefulness of a link. Given perfect, unchanging network conditions, the utility of a dropped link would be exactly the same as if it were to be added again. For this to work, the threshold for dropping a link must be calculated as if that link were not in

Algorithm 4 Drop Links

```
$utility \Leftarrow 0$
$L \Leftarrow$ link L to randomly selected neighbour
**for** each member, $M$, such that $M \neq localhost$ **do**
  $L_n \Leftarrow$ new latency to $M$, without $L$ in place
  $L_c \Leftarrow$ current latency to $M$, with $L$ in place
  **if** $L_n = \infty$ **then**
    return
  **else if** $L_n > L_c$ **then**
    $utility \Leftarrow utility + \frac{(L_n - L_c)}{L_n}$
  **end if**
**end for**
**if** $utility < threshold$ **then**
  drop link $L$
**end if**
```

place (as the utility if the link were being added would be compared to the threshold before the addition of the link).

Little has to be changed from the Algorithm 2 for adding links, presented in Section 4.5.1. It is simply a reverse of the link addition. New overlays distances are those without the link being evaluated while current overlays distances are those with the link in place; if the utility of a link is below a given threshold – rather than above – the link will be dropped.

In normal circumstances, the dropping of a link will not create a partition in the mesh, provided that when checking the latencies, the observation of any infinite length links is enough to signal that the link should not be dropped.

By introducing this altered mechanism for dropping links, an Orta peer can judge the adding or removal of links against the same threshold calculation. This removes the necessity for the additional threshold required for dropping links as per Narada in Section 4.5.2.

6.4.3 Calculation of the Threshold

The threshold on which the adding and removing links in both Narada and Orta is difficult to specify for all network conditions. The threshold must be dependent on the size of the group, the number of neighbours a peer has, and also the number of neighbours the peer at the other end of the link has. Multiplying by these numbers alone would give a threshold value far too high to be able to add any links, though the whole lot can be multiplied by some small constant to deliver a useful threshold value.

It makes sense for the threshold to increase sharply once a peer has achieved a handful of links; the idea behind the threshold is that a peer should be able to achieve some links relatively easily, and after attaining those it shouldn't be able to add further poor quality links, only higher quality links.

Given the factors that the threshold must rely on, and a peer A with its neighbours B, it might be easily calculated as:

$$\begin{aligned} n &= \text{number\_of\_neighbours(A)}*\text{number\_of\_neighbours(B)} \\ m &= \text{number\_of\_members} \\ threshold &= const * n * m \end{aligned}$$

The calculation of this threshold value is covered further in Section 9.10.

6.5 Data Delivery

Routing tables are re-calculated on any link-state change using Dijkstra's shortest path algorithm. The routing tables stored reflect the nature of the peer group being one-to-many: rather than having a lookup table of *destination* against *next hop* as might be seen in a conventional IP router, here it makes more sense to store *source* against *next hop(s)*.

To calculate the routing table then, Dijkstra's algorithm is run for each source in the peer group. The local peer can then simply trace its own location in the spanning tree created, and store the outgoing links on this tree, if any, in the routing table against the source. If the local peer is a leaf-node on the tree, no entry need be added to the routing table for this source.

Using link-state, however, more computation is required to arrive at the same result. However, the outcome of this computation should be more up-to-date, and therefore precise, owing to the altered mechanism for distributing control traffic. Dijkstra's shortest path algorithm is reasonably efficient, given efficient data structures [22]. The worst case complexity of Dijkstra's in $O(m\log n)$, where m is the number of links in the network, and n is the number of nodes. To achieve this worst case complexity, efficient data structures such as adjacency lists for describing links between nodes, and a heap data structure for the ordered queue the algorithm uses for picking off the next best node when calculating shortest paths are required. Consider that at each peer, the algorithm will run once for each member in the group; with this in mind, the computational complexity of recalculating routing tables using this scheme is actually $O(mn\log n)$.

Once the routing tables have been calculated, they can simply be used for lookup purposes on the receipt of any data packet. The routing code must then both send data up toward the application layer, while also duplicating the packet for any outgoing links dictated by the routing table.

The routing table can only be affected by control traffic when link-state changes, at which point the routing table must be recomputed. Given that transmission times on control information packets are bound primarily by latency between hosts in the system, there will be short periods of time whereby the system is yet to converge on the same solution. Due to this fact, and that the fact that the system will constantly attempt to improve the quality of the links it holds onto, it is entirely likely that some packets may be lost or duplicated during transition periods. Further, routing tables at peers might allow looping of data packets during a transition period. These transition periods should not last long, however, due to the nature of the flooding mechanism used to distributed the control traffic. Data packets carry a time-to-live field, which would prevent looping packets from flooding the overlay until its destruction.

The increased frequency of state changes with a larger group precludes this protocol from being used for larger groups beyond the order of a few dozen members, due to the amount of computation taking place. The combination of increased computation, and more frequent state changes suggests that as group size increases, the time taken to reconfigure all routing tables in the group will take longer.

While Narada uses TFRC as the transport protocol for data connections, Narada uses plain UDP connections, leaving congestion control to the protocol being carried through the Orta links. This simplifies the design of Orta, and allows for a great deal of flexibility; RTP profiles for different types of data provide their own method of offering congestion control, so to force an assumption at a lower level would no doubt affect the performance of the RTP congestion control methods. In effect, an application is unaware of whether it is using IP Multicast or Orta to carry its data, so it seems wise to offer a basic, unreliable packet forwarding service, and allow the RTP protocol to run unaffected.

6.6 Summary

As in Narada, Orta uses a two-step process to construct the spanning trees it uses for multicasting of data, the first step involving the construction of a richer graph between nodes called a mesh, and the second being to create spanning trees rooted at each source in the group via some routing protocol.

Orta utilises link state routing rather than distance vector routing, which offers some very important benefits:

- Members are brought up to date with all state changes much faster.
- Owing to the storage of link state, link removal is more accurate.
- Also owing to the storage of link state, Dijkstra's shortest path algorithm can be used at each peer to calculate spanning trees from each source, and can do so to have the group arrive at a set of distribution trees which all 'agree', due to the flooding process.

These changes alone should suit Orta to real-time applications such as audio conferencing. The performance of the protocol is evaluated in Chapter 9.

Chapter 7

Implementation Details

The Orta protocol was implemented in C as a library which could be statically compiled into other applications. The interface to this library was based on that which the RTP code in RAT uses to initiate, destroy, and use a multicast session. This meant that the interface into the library was well-defined to start with, which served to guide development in the early stages of the project. The original API and the Orta API can be viewed in Appendices A and B respectively.

Many of the intricacies of the C language were learnt as the project advanced, such as using structs to carefully map data onto memory locations safely when dealing with packets, rather than as a convenient method for creating new data types and accessing members of that type.

Section 7.1 covers the architecture of the software. The Orta protocol relies on control traffic to maintain state at each peer, and handles data traffic as a separate entity from data traffic. Handling of control state and control traffic is dealt with here in Section 7.2 and Sections 7.3 and 7.4 respectively. Section 7.5 handles the generation of routing tables, with the data forwarding covered in Section 7.6. Finally, Section 7.7discusses known issues and improvements which could be made to the software, and Section 7.8 covers in brief the adaptation of the RAT application to use the Orta overlay.

7.1 Overview of Software Structure

Figure 7.1 gives an outline view of the conceptual structure of the software, showing the division between functionality of the control plane (which the application is never directly aware of), and the data plane. The only communication between the two is the routing table, which the control plane recalculates on receiving new link state information.

Identified in the discussion in Chapter 6, Orta peers are required to maintain group member state and link state, as well as keep track of group neighbours. This naturally lends itself to splitting control plane state out into three separate data structures to handle neighbours, group members, and all link state. The information in the latter of these two structures is used to calculate the routing table.

These data structures all have helper functions to allow lookups over, additions to, and removals from any of them. The functions provide quasi-object oriented design, by having the relevant data structure passed as the first argument to the function, rather than the function being an operation offered by the data structure as

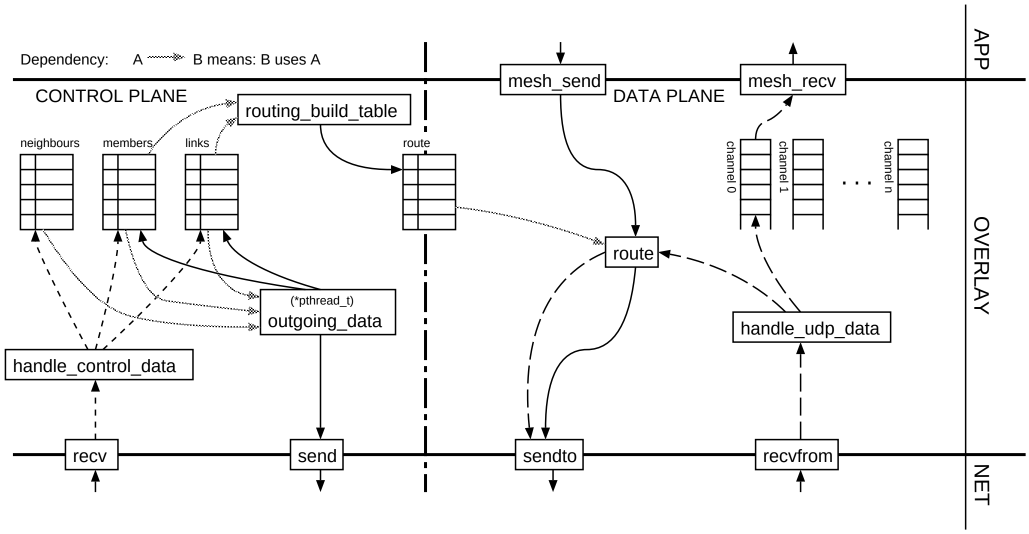


Figure 7.1: Conceptual view of the operation of the software architecture, showing the split between the control plane and the data plane; route() in the data plane handles the duplication and forwarding of packets to the relevant hosts.

in object oriented programming. It is fair to say that implementation and testing of these data structures whilst learning the various nuances of the C language took a considerable amount of development time.

The helper functions hide a certain amount of memory allocation and deallocation required to perform operations over the data structure. All data structures offer their own mutex lock, but helper functions do not lock over any of the structures themselves; this is left as a higher level task to be performed as and when necessary. To avoid deadlocks, locks are acquired in alphabetical order by the name of the variable they belong to (so “members→lock” must be locked before “neighbours→lock” if a function requires the use of both), and deallocated in the reverse order.

7.2 Control Plane

Control traffic, vital to the upkeep of the overlay, is sent primarily over TCP connections between neighbours. This makes the maintenance of state slightly trickier, in that on top of already monitoring which members of the group are also in the neighbours set, the implementation has to also handle TCP connections and socket descriptors. TCP traffic in this implementation always uses the arbitrary port number 5100, though a fixed port number certainly might not be ideal for all network configurations. An additional initialisation function which allows for an alternative port to be used for TCP is certainly viable, and would be trivial to implement. This implementation, however, assumes that all hosts are using the same port numbers.

The control plane handles group membership, link state, and neighbour state. Each of these is required to maintain correct state at each peer. It is responsible for sending data which affects other group members, and handling incoming data from other members which potentially affects local state.

While most control data is passed between members over TCP connections, the pinging mechanism which is part of the control plane is handled over the open UDP socket primarily intended for data transmission.

7.2.1 Neighbour State

The table handling all neighbour data is required to manage, for each neighbour, the socket descriptor for the TCP connection to that neighbour, the IP address for that neighbour, and the actual point-to-point latency to the neighbour (as mentioned in Section 6.3, a distinction should be made between the actual latency on a link, and the real latency on a link; the implementation allows for a 10% deviation from the advertised latency before the actual latency is advertised). The requirement to keep the actual latency of a link separate from the advertised link stems from the fact that for the generation of routing tables which create valid distribution trees, link state must be globally consistent.

To facilitate not sending information on this link at every refresh cycle, a 'last_sent' field is also defined for each neighbour, which is a counter that will increase at each refresh cycle until it hits the defined limit, arbitrarily defined in the implementation as 5 refresh cycles, after which some information must be sent about that link.

The neighbours form a linked list, with each neighbour easily indexed by socket descriptor. This allows for the flooding mechanism to run through the neighbours quickly, simply picking off the socket descriptor to use in the appropriate `send()` call.

7.2.2 Member State

Member state is marginally simpler than the neighbour state; all that is required for a member is that their IP address be held, the last received sequence number for that member, and a timestamp holding the local time that that last sequence number was received. The IP address is used to identify a member as an existing member when a refresh packet arrives, and therefore can be considered to index the entries; the same IP address should not appear twice. The IP addresses can also be used rather conveniently when picking a member to randomly probe, as per Section 6.4.1.

Members are, again, stored in a simple linked list, though the data structure can be thought of as a table indexed by IP address.

7.2.3 Link State

The link state to be stored is structured in a different manner from the member or neighbour state; link state is instead stored as a set of adjacency lists; implemented by having the head list identifying each member, each node of which is the head of a list identifying the neighbours to that member, and thus also describing which links are in place. Each link is identified by the member IP at the head of each adjacency list, the neighbour IP within the adjacency list, and also the weight of the link; an illustration of the data structure to help clarify can be seen in Figure 7.2.

Links added to the link state have a floor value, a minimal latency which no link will drop below. This is beneficial is local area network settings, since ping times are small enough that what are normally considered minor variations in ping times can alter the structure of the peer group quite considerably. Implementing a floor value can help prevent this reconfiguring across latencies between machines of a few hundred microseconds. The floor value in the implementation is 3ms, and was chosen as a cut-off point below which differences

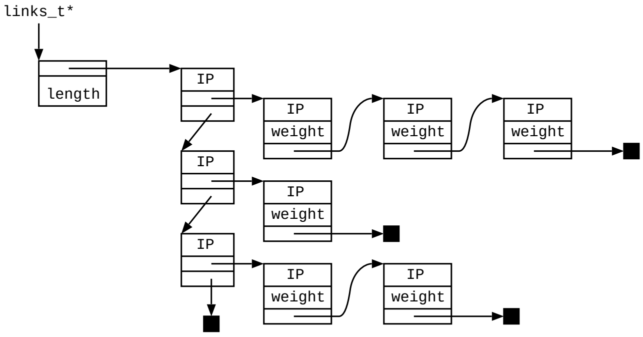


Figure 7.2: Illustration of the data structure used to maintain link state; the pointer at the upper left is the pointer maintained by the `mesh_t` data structure.

in latency do not necessarily matter (in the sense that there is little difference in the outcome if Orta were to route a packet through one host which is 1.5ms away, or another which is 2ms away); latencies across local area networks are generally negligible.[1]

7.3 Communicating Control State

TCP connections were chosen for the sharing of control data as TCP offers some guarantees that data won't be lost in transit (provided all routes to a host are not severed), and helps to ensure that state at each peer is kept up to date; the only additional cost for dealing with TCP connections is the handling of socket descriptors, as mentioned in Section 7.2.1. All control traffic which affects state is sent over TCP connections. We can consider three different types of control traffic:

- One-time floods
- Regular floods
- Probing of existing/potential links

Of these, the first two are sent over TCP, the third is sent over the same UDP socket which is used for sending application data. A fourth category would include more miscellaneous types of communication, such as join requests, acceptance packets, requests to add links, etc. Each of these types of traffic shall be considered in turn in the following subsections.

7.3.1 One-time Floods

These flood messages are sent as and when the event they describe occurs; they are not generated cyclically or otherwise predictably in any way. Each describes a distinct event, which requires that some part of the control state be updated, and also that routing tables are recalculated. Control data of this sort is flooded in the following circumstances:

- Addition of a new link.
- Removal of an existing link.

[1] Of interest is the latency between sibu.dcs.gla.ac.uk and www.gla.ac.uk, which is highly variable, but rarely greater than 3ms.

- Addition of a new group member.
- Removal of an existing group member.

Of these, the first two carry an updated sequence number from the source of the flood, to track the liveliness of group members. These message types correspond directly to the actions described in Sections 6.4.1, 6.4.2, 6.3.1, and 6.3.2 respectively.

7.3.2 Regular Floods

Control information sent regularly is that of refresh messages, which carry current link state (if either the weight of that link has changed significantly since the last refresh cycle, or if no information relating to that link has been sent recently), and carry the current sequence number for the peer which generates the message. The refresh messages are generated once every 30 seconds.

To help ensure that minimal bandwidth is used on each refresh cycle, exponential smoothing of latency values held within the neighbours state (Section 7.2.1) is employed, such that for each ping packet returned to a peer, the updated ping time can be described as:

$$p' = p * weight + p * (1 - weight)$$

This is used to counter the fact that variations between reported ping times can be relatively large, especially considering the pinging mechanism employed in Orta works on the application level, and is at the mercy of the Linux kernel at both ends of the communication. This smoothing process is buried within the functions which deal with neighbour table state, and need not be a concern for higher layers of the software, but help ensure that the network can stabilise given unchanging network conditions, rather than having the possibility of a data path fluctuating between two similarly weighted links. The weight used in the implementation is 0.95, weighting heavily toward old latency values.

If, when a refresh message is being generated, it is determined that the weight of a link has deviated far enough from the advertised weight, the current actual weight is used as the new advertised weight, with this value being entered into both the refresh message, and also the local link state, to be observed when the routing tables are recalculated.

7.3.3 Control Data over UDP

Ping packets are sent over UDP, the only reason being that throttling of connections at the TCP layer causes high levels of variation in the turnaround times of ping packets sent over TCP, causing spurious ping times to be returned. Sending ping packets by UDP generates more uniform ping times, though occasional spurious numbers are still sometimes observed (probably due to unfortunate scheduling decisions by the Linux kernel at end-hosts). The pinging mechanism employed is simple: a host places a timestamp into a packet and sends that packet to another host. The other host catches the packet and returns it, leaving the original timestamp intact. The source waits for this return packet and, on receipt, compares the time held in the packet to the current time.

Ping packets to randomly selected members, are sent using exactly the same packet types as for ping packets to existing neighbours, the only difference being that on a response, the peer who initiated the ping will notice that this peer is not currently a member, and will initiate code to evaluate the utility of that potential new link.

7.3.4 Miscellaneous Control Traffic

Other control data is sent during the lifetime of the mesh; some information exchange using TCP takes place between only two members. For instance, a new member requests with the existing member to join, and awaits an ‘ok’ or a ‘deny’ message before it will continue. The other data of this type is for the purpose of adding a link to the mesh, before flooding that the new link has been created, as per Section 7.3.1.

7.4 Control Packet Types

The following control data is sent over TCP:

- `join`: Sent to a peer by a new peer seeking to join the group.
- `join_ok`: Sent back to the peer seeking to join the group after receipt of a `join` packet. `join_ok` accepts this new member as a group member, and carries with it up-to-date information on the state of the system (current group members, current known links).

  On receipt of this message, the new member initiates a `flood_new_member`, which updates the rest of the group.
- `join_deny`: Sent back to the peer seeking to join the group after receipt of a `join` if the join cannot be accommodated. Currently unused, but code is in place to facilitate the use of a `join_deny`.

  On receipt of a `join_deny`, the attempt to join the group fails.
- `request_add_link`: Sent from one peer to another when a peer has determined that the creation of the link would benefit the mesh. This is similar to the `join_ok` operation, except that accepting this call does not require that all system state be sent back in response.
- `reply_add_link`: Sent back to the peer to notify that the new link has been accepted. On receipt of this message,
- `flood_new_link`: Sent on the creation of a new link; this packet is sent the peer who initiated the new connection, and carries the local sequence number.
  Initiator: Sends message to all neighbours.
  Other peers: Send message to all neighbours aside from the one from which it received this message, only if we’ve not previously received a control message from the source of the flood with the sequence number. Local state to be changed is the addition of link state for that link. The implication is that if the message was sent in the first place, then some state has changed somewhere else in the group; if we haven’t changed any state by receiving this message, we do not need to forward it, as we have already done so.
- `flood_drop_links`: Sent on the destruction of an existing link by the members at each end of that link.
  Initiators: Send this message to all neighbours.
  Other peers: Send message to all neighbours aside from the one from which we received this message, only if we’ve updated local state given this message. Local state to be changed is: the removal of link state for that link.

- `flood_new_member`: Sent on the addition of a new member by that new member itself.
  Initiator: Send this message to all neighbours.
  Other peers: Send message to all neighbours aside from the one from which we received this message, only if we've updated local state given this message. Local state to be changed is: the addition of member state and link state information to/from this new member.

- `flood_member_leave`: Sent on a member leaving the group, sent by that member itself, or sent by a member who has detected a failed peer within the group, as per Section 4.4.3.
  Initiator: Send this message to all neighbours.
  Other peers: Send message to all neighbours aside from the one from which we received this message, only if we've updated local state given this message. Local state to be changed is: the removal of member state and link state information to/from this member.

- `flood_refresh`: Sent periodically by each node, carrying a local sequence number (which other peers can use to help determine when they last heard from this peer), and local link state (ie: the current weights of outgoing links from this peer).
  Initiator: Sends this message out to all neighbours.
  Other peers: Send message to all neighbours aside from the one from which we received this message, only if we've updated local state given this packet. Local state to be changed is: link weights from the peer who created this packet.

Of the control traffic sent over UDP, the only calls for the definition of two packet types, the format of which is the same:

- `ping_request`: Carries a timestamp, and expects the receiver to simply return the packet, with the 'type' field modified.

- `ping_response`: The response to the request packet.

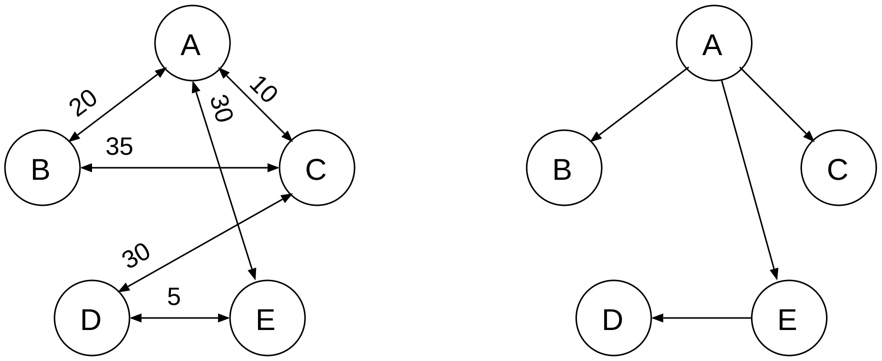
7.5 Calculating Routing Tables

On receipt of any control information which alters the state of the system, a peer must recalculate its routing table, to ensure that distribution trees are not disrupted.

If the peer were routing in the conventional manner, it would store the tree in some convenient form and use it for lookups by destination, choosing the appropriate output link from that information. Routing over the mesh is different, however, in that all peers are recipients of data, and the source of a packet is what is important, as the distribution tree for each may be different for each source.

Thus, each peer executes Dijkstra's shortest path algorithm for every group member. However, much of the data generated here is redundant, as each peer only concerns themselves with the point at which they occur in each of the distribution trees. By storing each source and a matching list of outgoing links in an adjacency matrix, each peer knows which peers it must forward data packets to as part of the distribution tree for that source. Lots of information is thrown away from each tree here, and for larger groups substantial amounts of computation will be performed.

Figure 7.3 demonstrates, on a small scale, how this works. Figure 7.3(a) shows the logical mesh topology, while Figure 7.3(b) shows the resulting spanning tree for node A. Consider node E; on calculating this



(a) Five peers connected by a virtual mesh structure, with link weights shown.

(b) The same five peers, showing the spanning tree from A.

Figure 7.3: Illustrative diagram for discussion of routing tables.

spanning tree, E will store A as one of the keys for lookup on the routing table, with D as the result for lookup on A. Further, on calculating its own spanning tree, E will store A and D as results for lookup on the tag E. This allows the forwarding of all data packets to be handled by the same routine, in that the application can generate packets and as part of the `mesh_send()` call will cause a lookup on the source (in this case, the localhost), and from that arrive at a list of outgoing links.

7.6 Data Plane

Data traffic is sent using user-defined ports over UDP. This reflects the use of UDP by the IP Multicast code on which the library was modelled.

Given that the data plane and the control plane should not interfere with each other, the data plane does not need access to the control structures, and therefore never has to wait to acquire a lock to any control state. Access to the routing table is managed safely by a mutex lock, though in general only reads are taking place over the routing table. The only time this lock comes in truly useful is on the calculation of a new routing table, when the safe destruction of the previous table must be guaranteed to ensure that all forwarding operations still using the old table are completed, and that bad pointers which might cause a segmentation fault are avoided. This data structure would be more efficient if it were accessed using a read/write lock, or perhaps a semaphore.

7.6.1 Data Channels

Data channels are used to to emulate multiple sockets used to communicate with the same multicast group. In terms of RAT, this translated to one channel for RTP data, and one for RTCP data. The channels only translate to actual data structures at the receiving end of any communication: on sending, a packet is tagged with its channel number in the packet header, and on receiving, it is placed into the appropriate queue to be later retrieved by the application.

The implementation always assumes a channel 0 (zero) exists, so for the case where only basic communication is needed with the multicast group, use of the overlay is marginally simpler.

7.7 Known Issues & Potential Improvements

The peers using the overlay must space connections apart by at least 100ms; due to the handling of the select() statement to monitor TCP connections, the socket that listens for new connections must be polled rather than simply blocking over the socket as would be normal. This does not normally cause a problem for the running of the overlay.

More intelligent pinging code could be implemented. Currently the code for pinging hosts does not log which hosts it has pinged; it merely responds on receiving the appropriate packet and assumes that the timestamp stored within that packet is one which it generated itself. A more intelligent way of handling ping packets might be to have a fixed allocation of pings which can be pending a return packet, holding the time sent for that ping, and storing an ID in the outgoing packet. This would allow for peers to determine when a host is not reachable (either due to that host being down, or due to firewall restrictions in place), by monitoring for lost ping packets.

Explicitly defined packet types for the addition of a new member or a new link are not necessarily required; the same functionality could be implemented via normal refresh packets. The advantage to this would simply be that there would be less duplication of code within the mesh, and therefore less chance of triggering an obscure bug.

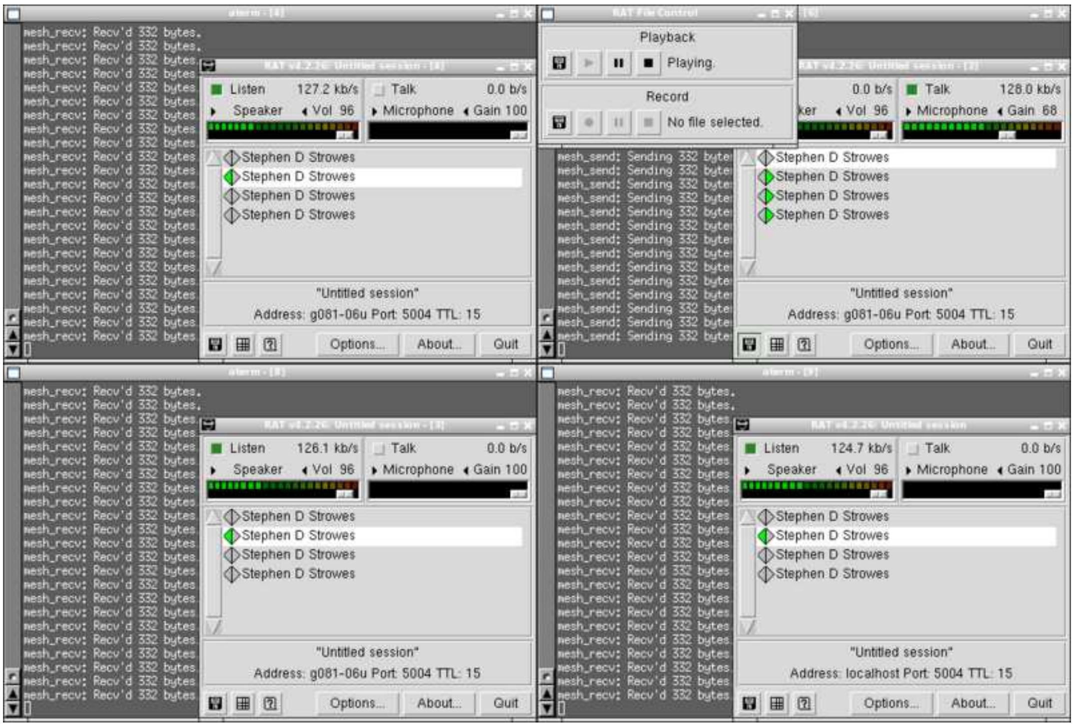
Also related to control packets, some bit packing could allow for more efficient control packets, to help save bandwidth usage. This is not a vital requirement, but would help reduce the volume of the control traffic during the lifetime of the overlay.

Improving the locking mechanism over the routing table, as noted in Section 7.5, would allow for more efficient data transfers than the current mutex can allow. This is has not been a problem during the testing or evaluation of the project, but it might become a limiting factor for data rates, depending on group size or type of data to be carried.

7.8 Integration into RAT

RAT was modified to send and receive data using the Orta overlay. The modifications required can be seen in the patches provided in Appendix C. All changes to actual source code were contained to one file; the rest of the changes are within Makefiles to specify the location of the library and to define the identifier to have the preprocessor enable the Orta as opposed to the IP Multicast code.

A screenshot of 4 RAT clients communicating via the overlay can be seen in Figure 7.4.



```
mesh_recv: Recv'd 332 bytes.
mesh_recv: Recv'd 332 bytes.
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
```

🟩 Listen 127.2 kb/s | □ Talk 0.0 b/s
Speaker Vol 96 | Microphone Gain 100

Stephen D Strowes
Stephen D Strowes
Stephen D Strowes
Stephen D Strowes

"Untitled session"
Address: g081-06u Port: 5004 TTL: 15

Playing.

No file selected.

```
mesh_send: Sending 332 byte
mesh_send: Sending 332 byte
mesh_send: Sending 332 byte
mesh_send: Sending 332 byte
mesh_send: Sending 332 byte
mesh_send: Sending 332 byte
mesh_send: Sending 332 byte
mesh_send: Sending 332 byte
mesh_send: Sending 332 byte
mesh_send: Sending 332 byte
mesh_send: Sending 332 byte
mesh_send: Sending 332 byte
mesh_send: Sending 332 byte
mesh_send: Sending 332 byte
```

ker Vol 96 0.0 b/s | 🟩 Talk 128.0 kb/s
Microphone Gain 68

Stephen D Strowes
Stephen D Strowes
Stephen D Strowes
Stephen D Strowes

"Untitled session"
Address: g081-06u Port: 5004 TTL: 15

```
mesh_recv: Recv'd 332 bytes.
mesh_recv: Recv'd 332 bytes.
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
```

🟩 Listen 126.1 kb/s | □ Talk 0.0 b/s
Speaker Vol 96 | Microphone Gain 100

Stephen D Strowes
Stephen D Strowes
Stephen D Strowes
Stephen D Strowes

"Untitled session"
Address: g081-06u Port: 5004 TTL: 15

```
mesh_recv: Recv'd 332 bytes.
mesh_recv: Recv'd 332 bytes.
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
mesh_recv: Recv'd 332 bytes
```

🟩 Listen 124.7 kb/s | □ Talk 0.0 b/s
Speaker Vol 96 | Microphone Gain 100

Stephen D Strowes
Stephen D Strowes
Stephen D Strowes
Stephen D Strowes

"Untitled session"
Address: localhost Port: 5004 TTL: 15

Figure 7.4: Screenshot of multiple RAT clients using the Orta overlay for communication.

Chapter 8

Evaluation Approach

To evaluate the system, there are a number of characteristics of the protocol to look at. The nature of the overlay is important, with respect to how the protocol suits real-time communications. The evaluation of the system then requires that not only are the characteristics of the overlay measured, but also that measurements are taken to validate that the overlay can be used for real-time conferencing applications.

One key aspect of the testing is that as this overlay is designed for multi-source multicasting of data between all group members, all experiments will gather data for each source, unless otherwise stated. This makes the protocol presented here stand out against some of the other multicast protocols, in that many of the others present a case only for single-source multicast to many group members.

Section 8.1 covers the different environments available for testing of the Orta protocol. Section 8.2 covers the various metrics by which the protocol will be tested, describing both why each metric is relevant, and how it will be tested. A summary of the metrics is presented in Section 8.3

8.1 Testing Environments

Given the time available to conduct the evaluation, the construction of a network simulator, while desirable due to its ability to evaluate the protocol over a variety of interesting network topologies at varying group sizes, could not feasibly be built and tested quickly enough to test the behaviour of the protocol. Other testing environments available were:

1. Large numbers of undergraduate lab machines.
2. One machine at the University of Southern California's Information Sciences Institute (ISI) in Los Angeles.
3. A small network of systems linked together by transparent bridges running FreeBSD with Dummynet[1] functionality enabled to impose latencies on links. This network consists 6 hosts with one network card to act as end-hosts on the network, and five machines with two network cards to offer up to five transparent bridges, or perhaps act as additional end-hosts.

The first setup is useful for the purposes of observing how the protocol behaves at different group sizes. The lab machines are all located on the same LAN, so exhibit similar latencies between any pair of machines.

[1] `http://www.dummynet.com/`

These latencies are measured in microseconds, rather than milliseconds, so experiments run over the lab machines cannot be used to derive much information about additional delays imposed by routing through the overlay, for example.

In terms of evaluation, firewall configurations prevented the machine at ISI from being a useful member of either peer group involving the other machines. The machine was, however, useful for testing of code before evaluation, and became the ping site for the California node in the dummynets used (Figure 8.1(b)).

The third setup mentioned above is useful for simulating a more realistic setting where peers are sited at different geographic locations, thus imposing real latencies on packet transmission. Dummynet offers functionality designed to be used for the testing of network protocols, and allows for enforcing network behaviour such as imposing delays on links, bounding queue sizes in the router, and increasing packet loss rates. With bridge and dummynet functionality enabled in their kernels, the FreeBSD machines can easily be set up to forward traffic between the other hosts, with the simulated delays able to emulate high-latency links between hosts in the same room; for example, this can be used to simulate peers at different ends of the UK. The actual dummynet networks constructed can be seen in Figures 8.1(a)[2] and 8.1(b)[3]. The Dummynet topologies shown here were designed to offer real-world latencies given group members at different locations around the world; place names are simply useful to get a rough idea of the physical geography. While real locations were used to derive approximate latencies between sites, they haven't necessarily been followed too closely, due to the limited number of machines capable of acting as transparent bridges in any test setup.

8.2 Evaluation Metrics

Many of the test metrics described in the following sections could have variants which run on the first network mentioned previously to monitor behaviour as group sizes increase, as well as the last network to test behaviour with realistic latencies in place. Each of the following will describe which network the metric will be measured on, how it will be measured, and what effect the metric can have on the behaviour or suitability of the overlay as a carrier of real-time audio. In all tests, all peers connect to the first member of the group spawned; this allows for an inefficient mesh to be formed, which Orta must improve upon.

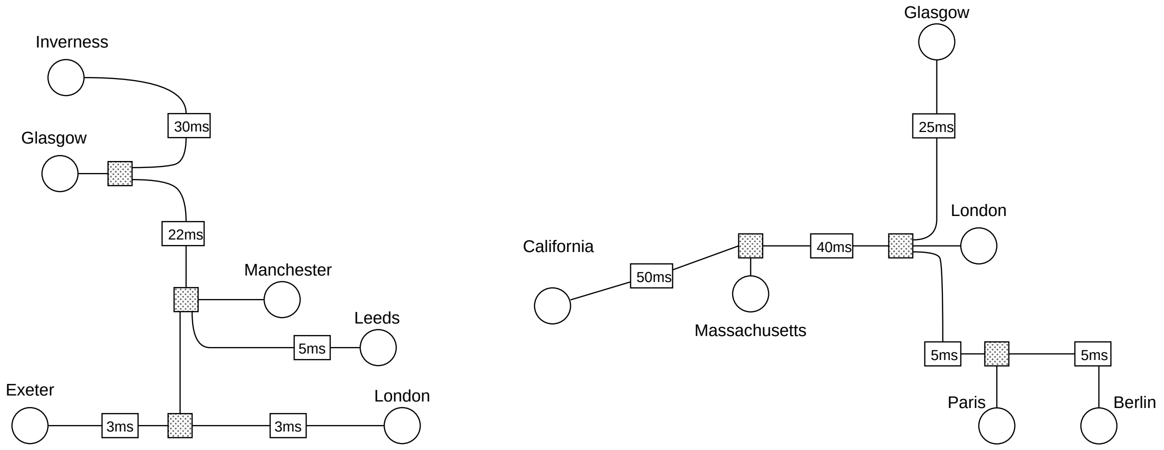
8.2.1 Worst Case Stress

The very nature of the overlay means that individual hosts are sending and receiving more data packets than the application layer is aware of. How many packets are duplicated is determined by how many links the protocol creates to or from each host for a given network setup. This duplication raises two concerns:

1. Additional bandwidth usage over links to end-hosts. While the discussion of the protocol has not considered bandwidth, it would be a serious issue to be considered if connection types varied considerably. Clearly, many connections to be carrying data over a modem link is not as desirable as the same number of connections through an ADSL link, or an Ethernet link.

[2] Measured from Glasgow; Inverness: uhi.ac.uk (60ms); Manchester: www.mbs.ac.uk (45ms); Leeds: www.leeds.ac.uk (45ms); London: scary.cs.ucl.ac.uk (50ms); Exeter: www.ex.ac.uk (50ms). Ping times are approximate to average ping time logged at start of evaluation.

[3] Measured from Glasgow; California: kame.isi.edu (175ms); Massachusetts: mit.edu (110ms); London: scary.cs.ucl.ac.uk (50ms); Paris: www.univ-paris3.fr (50ms); Berlin: ping www.tu-berlin.de (60ms). Ping times are approximate to average ping time logged at start of evaluation.



(a) UK Dummynet.

(b) Cross-Atlantic Dummynet.

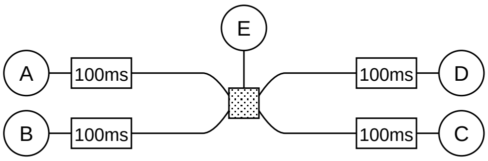
Figure 8.1: Graphs of the dummynet setups used for testing. Boxes containing numbers represent the delay in one direction in the line (so the round trip time from Inverness to Glasgow is 60ms, not 30ms) which is created by a transparent bridge with dummynet functionality enabled between hosts. Shaded boxes are merely switches, and empty circles are end-hosts eligible for running the test software.

2. The increase in packets to be processed at each host leads to increased processing costs in the overlay code executing at the application layer, and puts additional pressure on the networking subsystem of the host operating system.

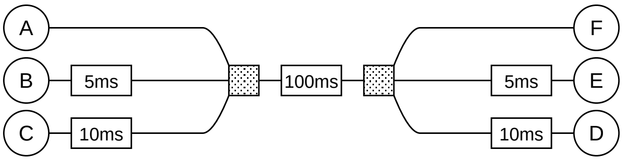
The stress of a link is simply defined as the number of identical copies of a packet carried by a physical link to deliver that packet to the rest of the peer group. Worst Case Stress is then the maximum stress value observed on any link in the group. In comparison, all links in a properly configured IP Multicast group have a stress of 1, while a naïve conferencing overlay which created a connection between every pair of members over which the members copy packets directly to all recipients would have a physical link stress of n on access links. Orta should attempt to minimise the stress of links throughout the group by spreading the duplication of packets throughout the distribution trees created from each source in the group.

Determining the Worst Case Stress of a network could be automated, but it is much more useful to design a network specifically for observing what Worst Case Stress value Orta will generate for that network. As such, the two networks shown in Figure 8.2 will be used specifically to discuss the Worst Case Stress behaviour of Orta. For each of these networks, simple test code will be run which will initiate connections between Orta peers; debug output from the Orta code can be used to easily trace how links have been placed across group members. For the sake of comparison, Worst Case Stress will also be considered for the UK and Cross-Atlantic Dummynets described already, and it may be possible to look at Worst Case Stress values on a group of Orta peers running over the undergraduate lab machines.

The goal of the protocol should be to keep link stress low across as many group members as possible, thus spreading the duplication of packets across group members.



(a) Worst case stress, test network 1. Designed to attract connections to the centre peer.



(b) Worst case stress, test network 2. Designed to monitor link stress in the presence of one high-latency link.

Figure 8.2: The physical topologies of the Dummynet networks to be used for testing Worst Case Stress.

8.2.2 Absolute Round Trip Time

As discussed in Chapter 5, the round trip time (RTT) between two hosts is important when we're considering conversational audio. By looking at the round trip times a group of Orta peers generate, it should become apparent as to whether or not Orta is making sensible decisions regarding where in the mesh to place links. This experiment will be run over the UK and Cross-Atlantic dummynets. It makes little sense to run the experiment over the lab machines, where latencies are so low that it does not necessarily matter how data packets are routed, so long as they arrive at all recipients.

Round trip time can easily be measured using only application code designed for the purpose of the experiment; no modification of library code is required. If the source of a packet sends, through the overlay, a packet containing the current timestamp, the source IP address, and something to identify it clearly as a 'SEND' packet, then each recipient can sends the same packet to the mesh with type simply changed to 'RESPONSE' packet, altering no other state. The source can identify its own 'RESPONSE' packets by the presence of its IP address in the packet, and can then calculate the difference between the current time and the time carried in the packet.

For each response received, the test code should output time deltas in the form:

- `RTT host_ip packet_no RTT_value`

The only limit on the frequency of these packets is that the recipient of an RTT packet which it did not generate sends its response via the overlay, and as such, each peer will receive many response packets which it must ignore. With all peers sending RTT packets, this method of calculating the RTT between pairs of hosts is costly, but works without modification to the library code.

After leaving the test code to run for 10 minutes, to ensure that over time the overlay not only arrives at a reasonable solution, but also stabilises and does not disrupt good links, RTT values can be collected and plotted over time for each pair of hosts, to give a view as to what the overlay is doing. RTT packets every 20 seconds would give a reasonable look at RTT between all pairs of peers during the lifetime of the group, while not flooding the network.

8.2.3 Adaptability of Mesh to Changing Network Conditions

Since the intention of the mesh approach to building distribution trees for carrying data is that the quality of the mesh gradually improves over time, it's worthwhile observing how Orta behaves when the network conditions change during the lifetime of a peer group. This experiment, by its very nature, requires the use of the dummynet networks defined earlier.

Peers would be started on the dummynet networks shown in Figure 8.1, and given time to settle on a mesh configuration. Link weights can then be modified by altering the rules at the transparent bridges that govern the delays on links, and the peers again given time to settle on a configuration.

The following alterations to the dummynet have been chosen to observe the behaviour of the protocol under different conditions; some 'long' links have been shorted, and some 'short' links have been lengthened.

- On the UK dummynet network, the latency on the bridge between Inverness and Glasgow shall be reduced from 30ms to 5ms, thus removing that long link to try and prompt more links to be formed up to Inverness. Further, the latency on the bridge separating London and the switch to which is is attached will be increased from 3ms to 100ms, providing a longer link which Orta should try to avoid.
- On the Cross-Atlantic dummynet network, the latency on the bridge between California and Massachusetts would be reduced from 50ms to 5ms; the latency on the bridge separating Paris and Berlin from the rest of the network shall be increased from 5ms to 100ms; finally, the latency on the final bridge before Berlin will be increased from 5ms to 50ms.

In order to monitor the variation in packet round trip times, it may be useful to graph round trip times in the same manner as suggested in Section 8.2.2. For further visualisation, it would be possible to display the altered logical 'shape' of the mesh in a graphical form from both before and after the modification of link weights.

8.2.4 Normalised Resource Usage

The Resource Usage (R.U.) of the overlay, as defined in [15], is calculated as:

$$RU = \sum_{i} d_i * s_i$$

where: link i is an active link
d_i is the delay of link i.
s_i is the stress of link i.

This can easily be calculated by summing the weight of the link each data packet is forwarded on at each peer; by considering each one in turn, duplicated packets are taken care of naturally.

Normalised Resource Usage (NRU) then is:

$$NRU = \frac{(R.U.\ through\ overlay)}{(R.U.\ over\ DVRMP)}$$

This can be calculated by hand over the dummynet network, using the stress values identified in Section 8.2.1. Without knowledge of precise latencies of links between routers and hosts in the lab network, it becomes difficult to determine the weights of each physical link used over a DVMRP tree in that environment.

8.2.5 Relative Delay Penalty

By routing packets through numerous hosts to deliver data to all recipients, packets will naturally be in transit for at least as as if they were to be duplicated and delivered by the network alone. The Relative Delay Penalty is describes the increase in delay that applications perceive while using the mesh when compared to the same unicast links being used to communicate directly between any two hosts in the peer group.

It is defined as:

$$RDP = \frac{mesh\ latency}{normal\ latency} >= 1.0$$

Note that RDP values closer to 1.0 are better, since the additional delay through the overlay would then be smaller.

The ideal would be to monitor how this metric changes as group size increases, but latencies between between lab machines are so small that scheduling decisions made by the Linux kernel on each host can have a dramatic impact on the reported latencies between machines at the application level. Thus, the time taken to route a packet across the lab machines is not a good indicator of the performance of the mesh in this scenario. For this reason, the Dummynet setups shall be used for a “real world” look at RDP.

The experiment would involve setting up the peers to connect to each other and have them regularly send packets to each other. The frequency of these packets does not matter, so long as they would run for 10 minutes, to allow time for the overlay to settle. All members can connect to the first peer to be spawned, leaving the mesh to improve itself and generate data transmission trees.

Peers would be transmitting numbered packets at a rate of one approximately every 20ms. With every other peer outputting the time taken for each packet to arrive using the link state information held for each peer’s neighbour (which should prove accurate enough). Determining actual ping times between the source node and each peer could also be scriptable.
Testing of this metric requires modification of library code. On receipt of each packet, peers should output:

- `RDP host_ip source_ip packet_no distance_to_here`

Analysis of the behaviour of the RDP value after the fact can then be performed.

8.2.6 Time Taken to Repair a Partitioned Mesh

While it is possible to have the mesh disrupted by a member leaving, as discussed in Section 6.3.2, there is the further possibility of a member failing and not exiting cleanly, in which case group members must determine that this member has died as per the algorithm presented in Section 4.4.3, and attempt to clean up state.

This experiment could either be lab-based or dummynet based; it’s not entirely important which one. The difficulty lies in partitioning the overlay.

In order that the overlay can be partitioned with ease, it makes sense in this case to disable the code responsible for attempting to add links during the normal operation of the overlay. We shall assume that the mesh is partitioned on a peer dying unexpectedly, leaving all physical links in place (obviously if the only physical link between the two halves of the peer group is severed, software has no hope of overcoming the problem).

To demonstrate the repairing of an overlay partition, a chain of peers will be constructed. One of the peers in the middle of that chain can then be stopped abruptly (‘kill -9’), and the other peers left to figure out what happened as part of their normal running cycle.

Observation of link-state on successful repartitioning of the overlay, using debug output to determine the sequence of events leading up to the reconstruction of the mesh. Time taken to repartition the overlay is essentially time when peers log the dead peer as having disconnected, until the first new link is added from one side of the partition to the other.

8.2.7 Volumes of Control Traffic Sent

This is an interesting metric due to the very different way that control traffic is handled in the Orta protocol compared to the Narada protocol. Control traffic is particularly easy to monitor; each node can be modified to output the size in bytes of all control traffic sent or forwarded. Once all peers have terminated, calculating the total control traffic sent from each node is trivial.

To observe the variations in control traffic sent, this experiment should produce graphs showing the average level of control traffic with error bars showing the minimum and maximum volumes of control traffic, plotted against group size.

Testing of this metric requires modification of library code. On sending or forwarding control data, each peer should log the number of bytes sent; if the peer logged the volume of data sent over TCP connections and the volume sent over UDP connections, a comparison can be drawn between the two types of control data.

This experiment can, again, be run with differing group sizes, with each experiment running for 5 minutes.

The nature of the latency between lab machines being so low does not demonstrate a ‘normal’ situation for the flooding of control data, however. The dummynet network can also be used to monitor control traffic volume for one or two of the smaller group sizes tested on the undergraduate lab machines, to compare volumes of flooded data.

8.2.8 Reliability of Overlay: Lost & Duplicated Packets

Reliability of the overlay can easily be measured using simple test harness code, simply making use of the overlay library.

Volume of lost packets can be calculated by:

```
  **if** $packet\_no > (max\_recvd\_packet\_no + 1)$ **then**
```

```
    $lost\_count+ = (packet\_no - max\_recvd\_packet\_no - 1)$
  end if
```

Volume of duplicated packets can be calculated by:

```
  if $packet\_no \leq max\_recvd\_packet\_no$ then
    dupe_count++
  end if
```

Running this test over larger groups on the lab machines should see marginally higher levels of lost/duplicated packets, as with the peers so close to equally distant from each other, the overlay is more prone to link state changes.

8.3 Summary

A number of test metrics have been proposed here by which the behaviour of Orta in different situations can be discussed. The results of the experiments to test these metrics are covered in Chapter 9.

Chapter 9

Evaluation

This chapter presents the results from the experiments discussed in Chapter 8. After discussion of the behaviour of the dummynet setups in Section 9.1, the following sections focus on each of the evaluation metrics covered in Section 8.2, in the same order.

For the purposes of the evaluation, two different experimental setups were used as detailed in Section 8.1. For experiments which made use of large numbers of machines, or comparisons between larger peers groups against smaller peer groups, undergraduate lab machines were used; these machines were all 1GHz Pentium IIIs running Scientific Linux[1], with its default Linux 2.4.21 kernel. Other experiments which require that nodes be some distance apart (in terms of latency) made use of a number of FreeBSD 4.11 systems setup to act as bridges, with dummynet functionality enabled for the purposes of introducing additional latency over links. End hosts on this network were 450MHz Pentium IIIs running Knoppix[2], installed directly onto disk.

In all experiments presented, the threshold for addition or removal of links was calculated as described in Section 6.4.3, with the constant value being set to 0.02. The constant was chosen based on testing during the implementation phase. Testing was primarily conducted on the lab machines, and over time this threshold appeared to give a reasonable balance between peers holding links to too many neighbours, and the opposite, peers not forming any more links than they were initially started with. The threshold is discussed further in Section 9.10.

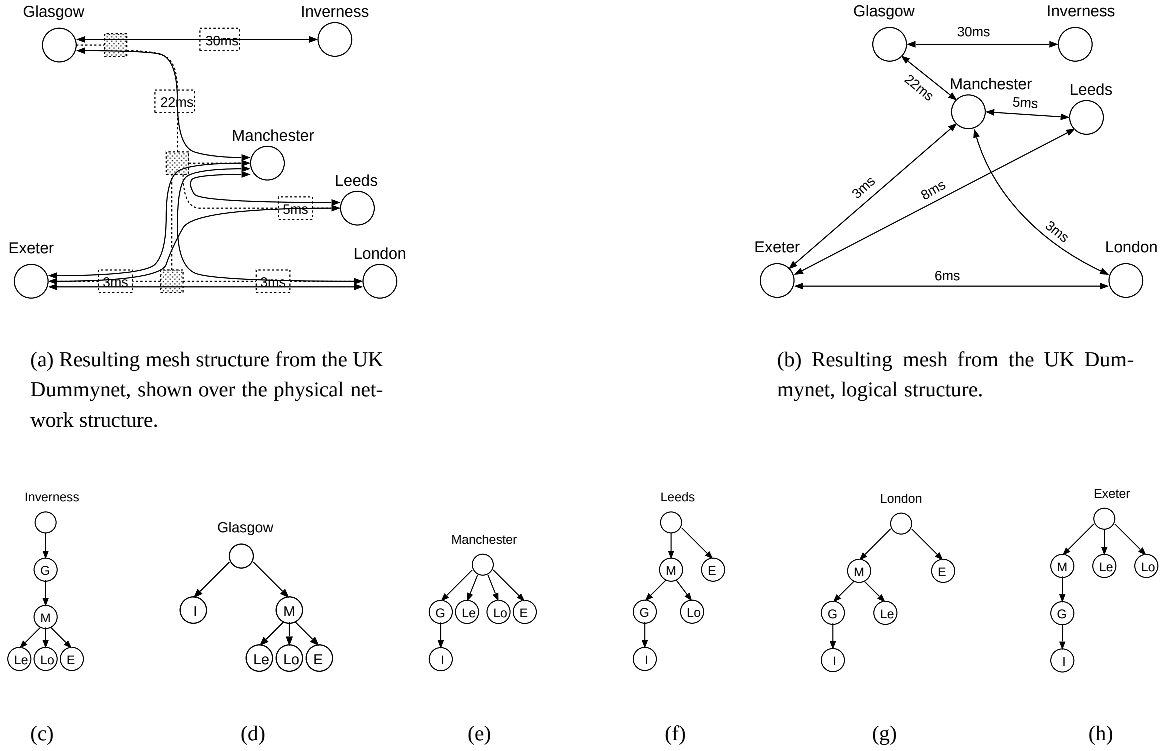
9.1 Analysis of dummynet networks

Before discussion of the results from the test metrics themselves, the behaviour of the Orta protocol over the dummynet networks presented in Figures 8.1(a) and 8.1(b) is should be considered. Figures 9.1 and 9.2 show the resulting overlay structure and distribution trees for both these dummynets, using the threshold defined earlier.

In each of Figure 9.1 and Figure 9.2, subfigure (a) shows the mesh superimposed over the physical network topology, while subfigure (b) shows the logical network structure of this mesh. Subfigures (c) through (h) show the distribution trees rooted at each source, with the recipients on each tree having their names condensed to one or two letters which uniquely identify the peer.

[1] `https://www.scientificlinux.org/`

[2] `https://www.knoppix.org/`



(a) Resulting mesh structure from the UK Dummynet, shown over the physical network structure.

(b) Resulting mesh from the UK Dummynet, logical structure.

(c)

(d)

(e)

(f)

(g)

(h)

Figure 9.1: The resulting mesh over the UK dummynet, and the distribution trees rooted at each source.

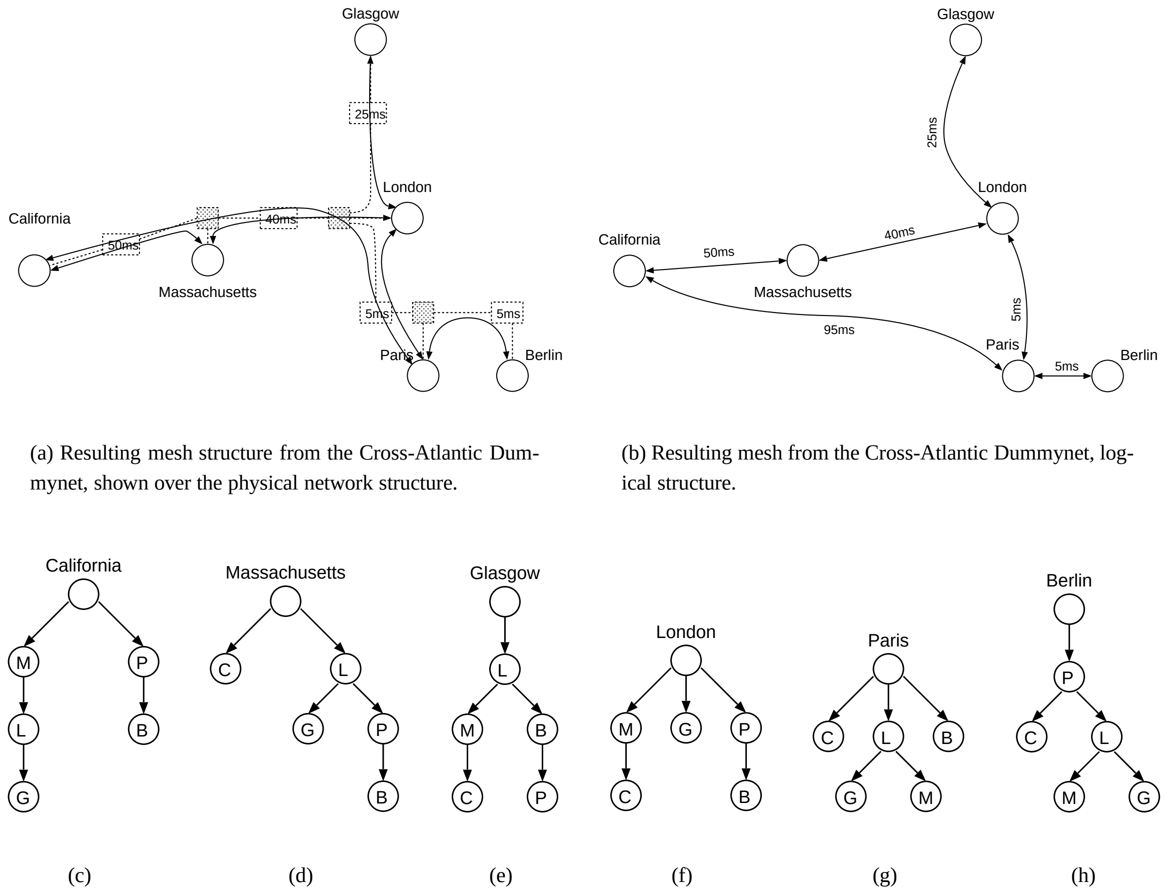
9.1.1 UK Dummynet

Figure 9.1 shows the mesh structure that Orta forms over the UK Dummynet. Orta has concentrated links between Exeter, London, Manchester and Leeds, providing only one link to Glasgow and Inverness. Further, the link to Inverness is through Glasgow, creating a chain of peers. The problem here is that if Manchester were to leave the group, Glasgow and Inverness would be separated from Leeds, Manchester and London, partitioning the mesh into two groups. Mechanisms described in Sections 6.3.3 and 6.4.1 would be able to repair the partition, but it would be desirable that a partition could not occur so easily. The solution might be to lower the threshold for adding links, but care would have to be taken to ensure that the threshold was not too low, allowing most, or all, possible links to be added.

The structure of the mesh beneath the Manchester node (i.e.: Leeds, London and Exeter), with many connections between members, would allow for any one of those peers to exit cleanly without affecting the others. Likewise, Inverness could exit cleanly, having no peers which depend on it. Glasgow and Manchester, however, become a potential points of failure.

An improved overlay structure would have at least one more link to Inverness. With that one link in place, either Glasgow or Inverness could leave the group without disturbing the data flow to or from the other.

Orta has created few connections over the longer physical links, and choosing instead to create many connections over the shorter links The protocol hasn't placed many connections over the longer link up to Inverness and Glasgow, instead choosing to concentrate more links among the hosts who are relatively close together.



(a) Resulting mesh structure from the Cross-Atlantic Dummynet, shown over the physical network structure.

(b) Resulting mesh from the Cross-Atlantic Dummynet, logical structure.

(c)

(d)

(e)

(f)

(g)

(h)

Figure 9.2: Distribution trees from each peer in the peer group, taken after the mesh had settled.

This makes sense, as Algorithm 2 in Section 4.5.1 will try to favour short links between any two peers. Since there are no links to Inverness considerably shorter than the links that it found, no more have been added. While the variation in latencies between some peers lower in the dummynet is equally small, the ratio between the overlay distance with or without the link is presumably significantly large for the link to be added, and for it to remain part of the mesh.

The fact that, although most of this network is well-connected, Orta has not created enough links suggests that the threshold value that determines whether or not a link should be added or not should be weighted to offer a higher chance of allowing a link when a peer has only one neighbour in the group.

9.1.2 Cross-Atlantic Dummynet

The mesh structure Orta forms over the Cross-Atlantic Dummynet, as seen in Figure 9.2, suffers from the same problem as the UK Dummynet. Glasgow and Berlin have both become leaf nodes in the mesh structure. Orta has not created a chain of peers like that of Manchester → Glasgow → Inverness seen in the UK dummynet.

The link between California and Paris is unexpected. It appears that to route a packet from California through

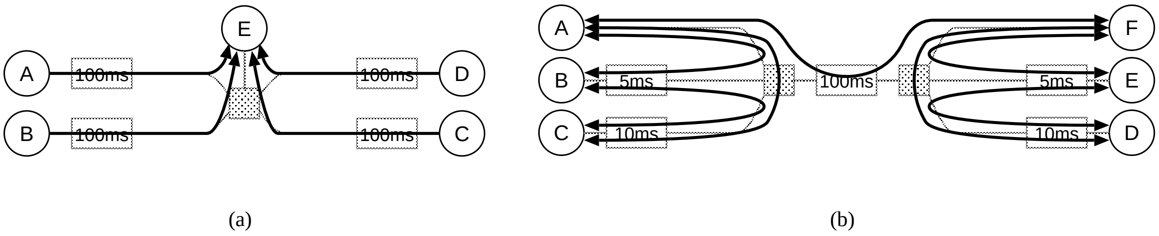


Figure 9.3: Logical topology of two of the test dummynets placed over the physical topology.

Massachusetts and London to Paris places enough additional latency on the round trip time that Orta has deemed the California → Paris link useful to the mesh structure. The addition of this link allows for a faster connection from California to not only Paris, but also Berlin, so the addition of the link is actually entirely reasonable, based on the link adding algorithm.

Again, this mesh could be improved by lowering the threshold required for adding links. The worry is that a departing group member can partition the mesh, which should be avoided if at all possible by the Orta peers.

9.2 Worst Case Stress

The physical topologies designed specifically for testing the worst case stress of a link were shown in Figure 8.2. These physical topologies were designed to observe Orta how behaves in smaller, perhaps more artificial, environments. Figure 8.2(a) was designed to attract links to the centre peer, while Figure 8.2(b) was designed to observe Orta's behaviour with two clusters separated by one high-latency link. The resulting mesh structures for this test can be seen in Figure 9.3.

Just as in the UK and Cross-Atlantic dummynets, the quality of the overlay in Figure 9.3(a) suffers by not creating enough connections between peers. This is, in part, due to the highly artificial nature of the overlay; if node E was behind an access link with an additional latency of a few milliseconds, the peers on the periphery of the network would be more likely to create some connections between each other, since routing through E would then become more expensive; E became the hub of connections on this test network purely because the time taken to route packets through E to any other host is essentially the same as the time taken to route directly from A to B, for example. A link from any peer to E is of lower latency and achieves the same round trip time as a link directly between A and D, for example.

The distribution trees for this network all route through E, and E duplicates packets to all other group members. Thus, the worst case stress for the network is 4. Consider Algorithm 4 in Section 6.4.2 for dropping links, and it becomes clear that the difference between the latency with and without a link between A and B will be so close to zero that it would be very difficult for any threshold value to retain these links. This, again, is a side-effect of the artificial design of the network, but it shows that it is still possible to achieve the worst possible stress value. All physical links other than the access link to E have stress 1, and it is only E which handles the duplication of packets.

In Figure 9.3(b), however, Orta has created many more links between peers. To have only one link between the two halves of the group is a point of failure, but only one link is formed in this scenario perhaps due to the highly artificial nature of the network. Peers A and F bridge the two halves of the group because they are the closest together of the two halves.

The worst case stress in this example is 3, and occurs on the links from A to its connecting switch, and from F to its connecting switch. The stress of the high latency link is 1. Orta, having partitioned this group into the two distant groups with a connection in the middle, has avoided the potential worst case stress of 5, in this case.

If we consider the UK dummynet in Figure 9.1 for slightly less artificial results, we can see that the worst case stress on any physical link is 4 at Manchester. Despite carrying three TCP connections, the physical link from Exeter is not used as much as it may seem. While the highest stress level it sees is 3, this stress level is only actually met by considering the distribution tree rooted at itself; by considering the other distribution trees, Exeter's stress level is significantly lower, and often 1.

Further, the and Cross-Atlantic dummynet in Figure 9.2 exhibit a worst case stress of 3 at both London and Paris, but in this scenario, those stress levels are met for numerous distribution trees. The worst case stress is clearly affected by the number of links that the protocol creates for the peer group.

Of further note is that the worst case stress across a group of size 36 was only 6.

9.3 Absolute Round Trip Time

RTT results for the two dummynet configurations can be seen in Figure 9.4. These graphs are showing a modified RTT between each pair of hosts, such that the lines represent raw RTT, plus another 60ms to compensate for buffering, encoding and decoding delays, etc.

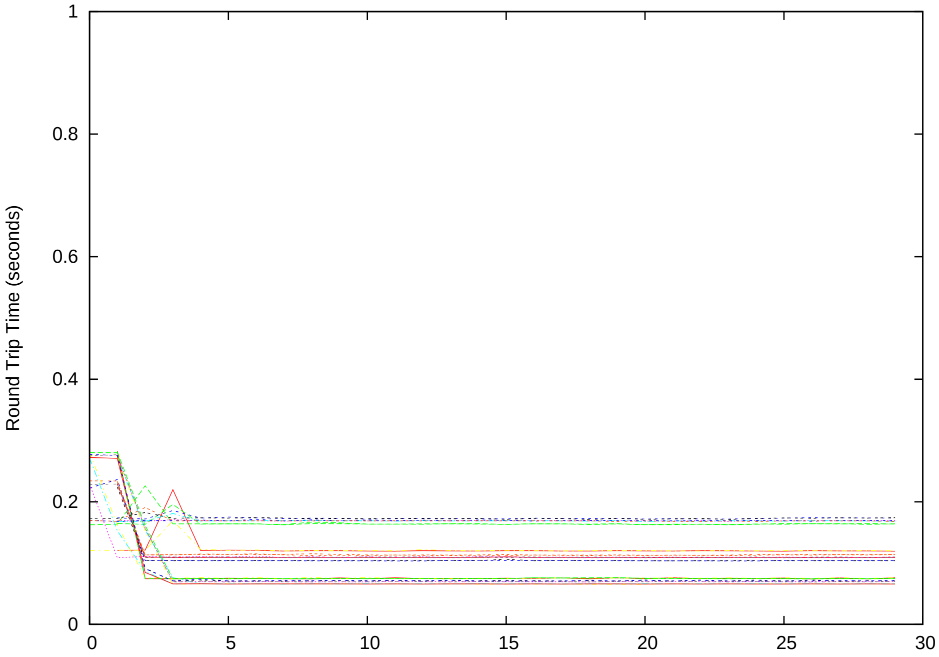
The variation we see in RTT near the start of the test is simply that of the mesh reconfiguring itself to find a good set of links for the overlay. All peers initially connect to the first peer to be initiated (which in the UK network is Inverness, and in the Cross Atlantic network is California); with this in mind, it is not unexpected that we see initially high RTTs for most hosts. The RTTs stabilise, however, and we can see that no two peers have a round trip time of over 400ms, aside from the initial poor configuration at the start of the simulation for the Cross Atlantic configuration.

We can see from these graphs that the overlay manages to add better links after the initial formation of the group, and keeps those good links (as the RTT does not increase once the good link is added).

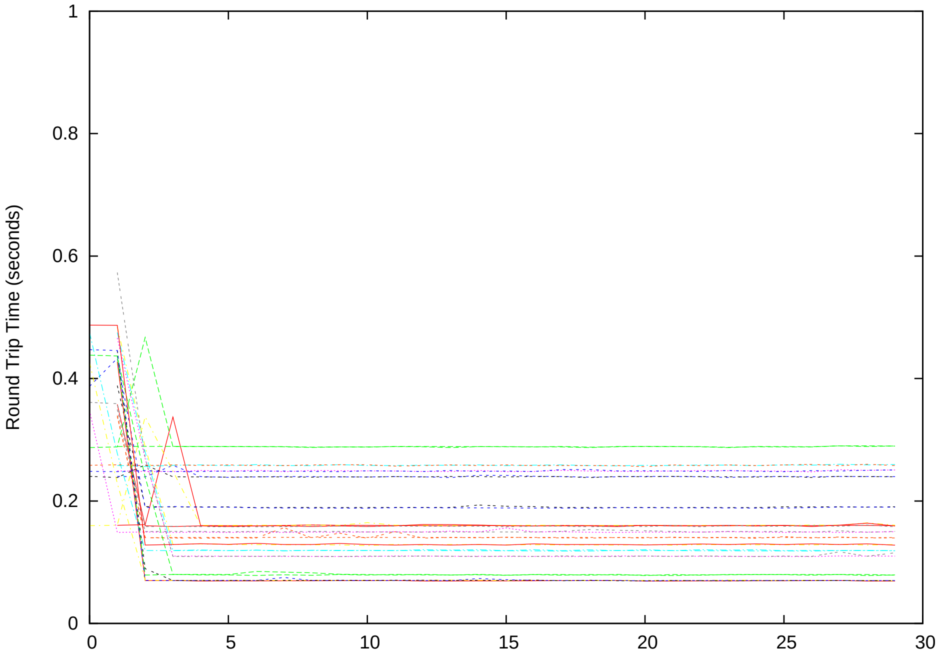
9.4 Adaptability of Mesh to Changing Network Conditions

As part of the Orta protocol, peers are required to constantly probe their neighbours, to monitor network conditions. The implication of this monitoring of network conditions is that the Orta protocol should be able to adapt should those conditions change.

For the purposes of this experiment, the two 'real-world' dummynets were used, and after allowing the peers to run for long enough to have the mesh stabilise, the latencies on some bridges were altered, as described in

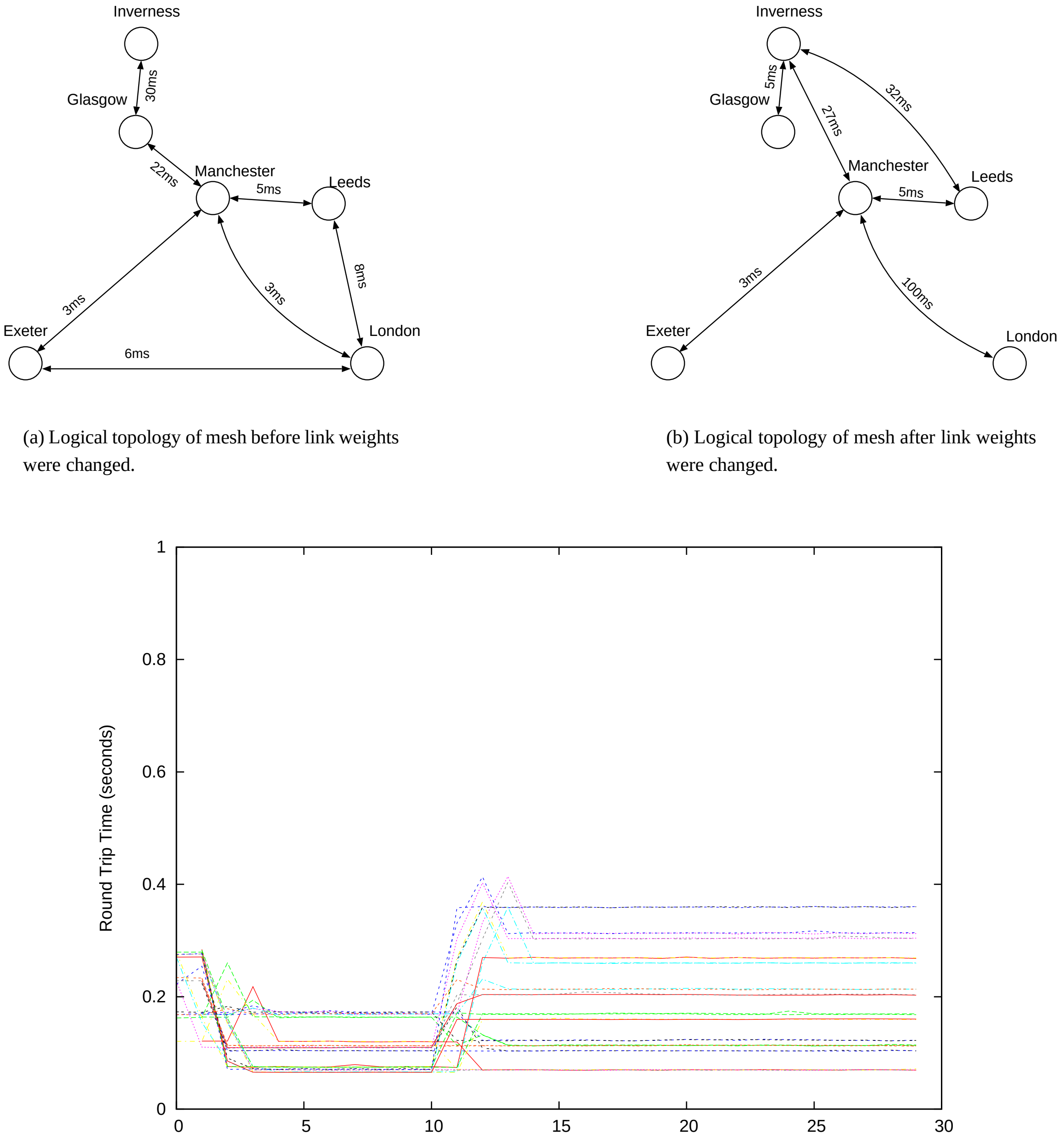


(a) RTTs from all hosts to all other hosts in the UK Dummynet. x-axis is represents time, over a 10 minute period.



(b) RTTs from all hosts to all other hosts in the EU-US Dummynet. x-axis is represents time, over a 10 minute period.

Figure 9.4: Graphs of Round Trip Times to all hosts over both Dummynets. Each line plots the variation in RTT from one host to one other in the group, over time. Numbers on x-axis merely indicate reading number; one reading every 10 seconds for the duration of the group.



(a) Logical topology of mesh before link weights were changed.

(b) Logical topology of mesh after link weights were changed.

(c) RTTs from all hosts to all other hosts in the UK Dummynet. x-axis is represents time, over a 10 minute period.

Figure 9.5: Variability of mesh under changing conditions on the UK Dummynet.

Section 8.2.3.

For both of these networks, the initial logical mesh structure is shown next to the resulting logical mesh structure. RTT values have also been provided, to monitor the behaviour of the mesh during the transition.

The behaviour of Orta over the UK Dummynet can be seen in Figure 9.5. The reduction of latency to Inverness has allowed more links to be added, though somewhat surprising is the removal of links to Glasgow. The assumption is, again, that without additional weights on access links, the difference in distance from Manchester to Inverness (via Glasgow or not), for example, is negligible. Orta has reduced the number of links to the London host, having had the latency to it increased considerably. On reconfiguration, Orta is still placing too few links to some nodes.

Figure 9.5(c) shows the variation in round trip time against the lifetime of the mesh; around the 10th sample point we see the rather dramatic reconfiguration of the network structure. Immediately after the link weights are reconfigured, round trip times peak, some at a little over 400ms. The mesh restructures itself and the routing tables react accordingly, however, to bring the round trip times between all pairs of members down to beneath the 400ms limit.

The variation observed in the Cross-Atlantic dummynet, see Figures 9.6(a) and 9.6(b), is minimal. Reducing the latency to California has allowed Orta to create one further link in the mesh, connecting California to Glasgow. No addition of links has occurred elsewhere. For the London/Paris/Berlin peers, this is understandable, due to the similar increase in the latencies observed on those links; by reducing the latency to California, it should be expected that a new link be formed to that peer.

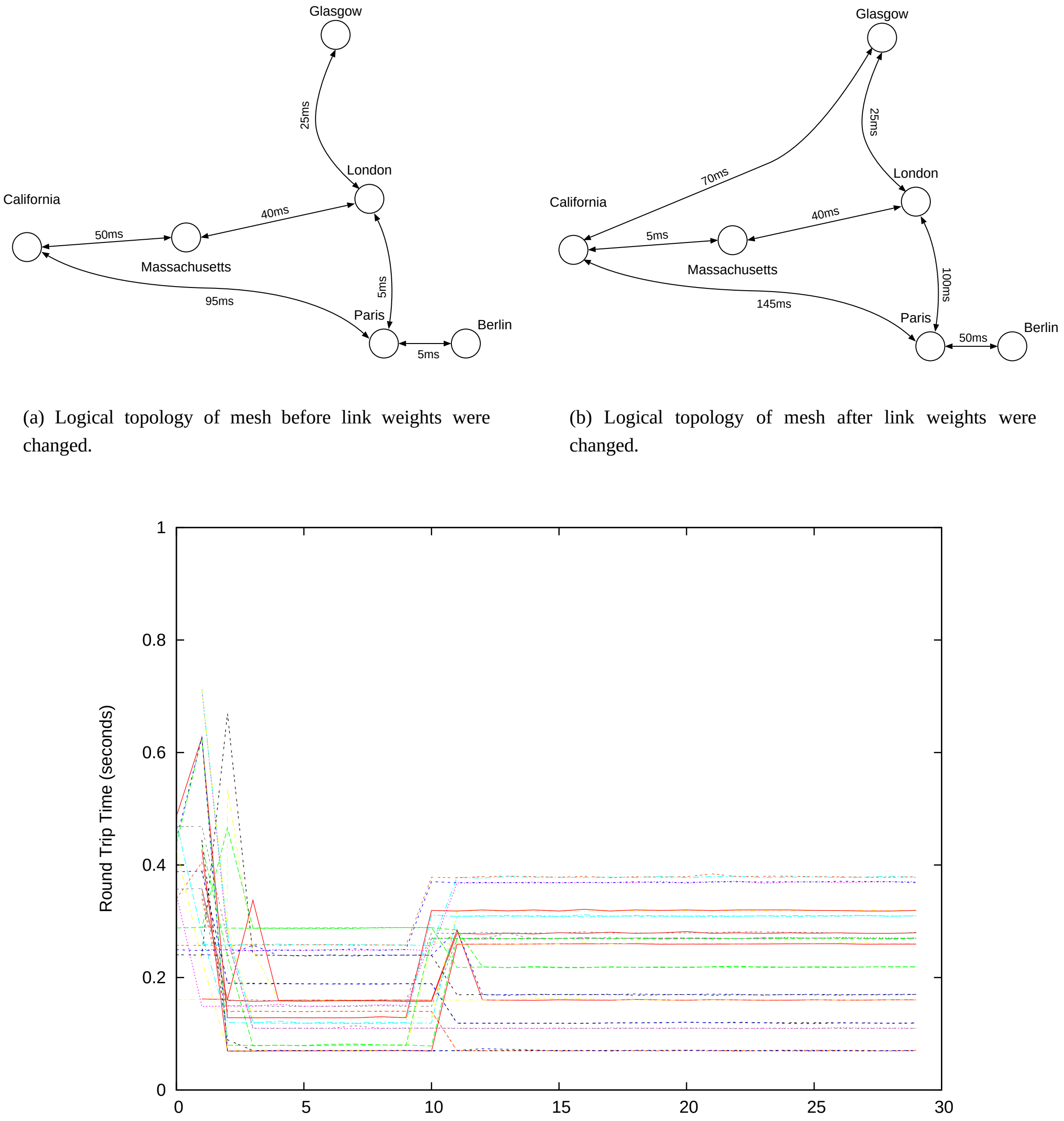
Figure 9.6(c) shows the variation in round trip time against the lifetime of the mesh; around the 10th sample point we see the reconfiguring of the network structure taking place, and the reconfiguration of routing tables and addition of the new link thereafter. It is worth noting here that on recofiguration, the peer group still manages to ensure that no two peers experience a round trip time of over 400ms.

The behaviour of the mesh in these situations shows that Orta can provide reasonable overlays for use in carrying real-time data, and also that the mesh is capable of reacting to changes in the network topology quickly enough to avoid unacceptable delays for extended periods of time between peers when routing through the overlay.

9.5 Normalised Resource Usage

Discussion of the Normalised Resource Usage (NRU), as defined in Section 8.2.4, is more difficult than originally intended, if the UK or Cross-Atlantic dummynets are to be considered. The difficulty lies in the structure of these networks. While Manchester looks, on the surface, like a useful candidate for examining the highest NRU values that the UK dummynet can produce, there is no latency on the access link to the Manchester host. The absence of any noticeable latency from that host means that the calculation of the metric is redundant. In essence, the stress of the link is still valid, but if the latency of the link is essentially zero, then the difference in resource usage between sending one packet or numerous along that physical link is marginal.

If we consider, again on the UK dummynet, the Exeter host, and the distribution tree rooted at that host,

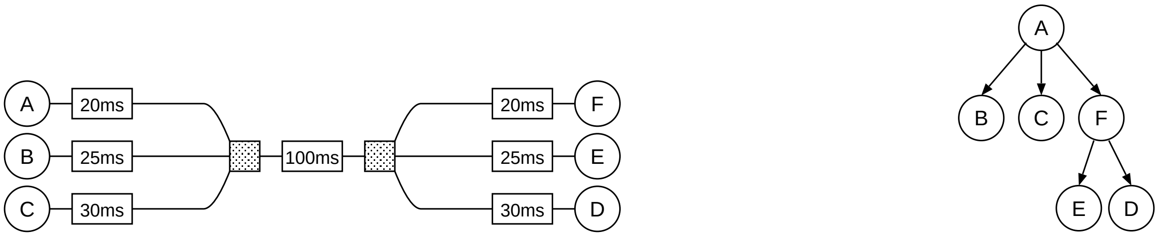


(a) Logical topology of mesh before link weights were changed.

(b) Logical topology of mesh after link weights were changed.

(c) RTTs from all hosts to all other hosts in the EU-US Dummynet. x-axis is represents time, over a 10 minute period.

Figure 9.6: Variability of mesh under changing conditions on the EU-US Dummynet.



(a) Artificial network for examining NRU.

(b) Distribution tree rooted at A.

Figure 9.7: Artificial network structure for discussion of normalised resource usage. (b) is the distribution tree created from A to the rest of the group, shown in (a).

we can derive some minimal results. With reference to the distribution tree from Exeter, 9.1(h), and the physical structure of the overlay, 9.1(a), we can trace the physical links used to distribute one packet throughout the multicast group: Exeter's distribution tree calls for three packets to be duplicated on the physical link from Exeter, and nowhere else. Thus, the total resource usage for that one packet is 3$*$3ms, plus the latency of every other physical link utilised in the mesh. This results in an NRU value of $\frac{69ms}{63ms} = 1.1$, which is reasonably close to the goal of 1.0.

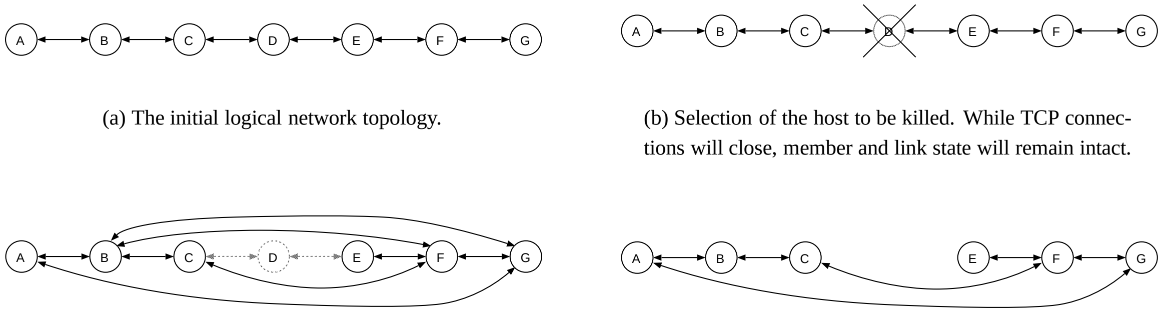
The difficulty with measuring this metric over the 'real-world' dummynets is that not all links have a noticeable delay. To this end, consider the network in Figure 9.7(a); with this physical network structure, we would likely see a mesh structure such as that shown in Figure 9.3(b), and as such, a distribution tree from member A such as that in Figure 9.7(b). In this example, the NRU for the distribution tree from A is $\frac{45ms+50ms+140ms+45ms+50ms}{20ms+25ms+30ms+100ms+20ms+25ms+30ms} = \frac{320ms}{250ms} = 1.28$. Contrasting this value to the NRU for DVMRP of 1.0, and the NRU for a naïve overlay application which might see an NRU as high as 2.12 in the worst case, we can see that Orta almost performs as well as the equivalent DVMRP tree.

9.6 Relative Delay Penalty

Relative delay penalty (RDP) is another metric which suffered from the design and size of the dummynets. The issue is that to route through several hosts to reach a destination on the overlay generally incurs the same cost as when routing directly between the same two hosts. This is in part due to the lack of links available to route connections, as in a real network infrastructure connections between pairs of group members would likely take different paths through the physical infrastructure. Because of this, all numbers returned by the RDP code were close to 1.0 for the dummynets tested.

If we consider again the mesh in Figure 9.7(a), introduced for Section 9.5, we can discuss how the RDP value is affected by the process of routing through the overlay. The RDP measure becomes an important measure as the size of the group expands, as it will directly affect the RTT times reported in Section 9.3.

By observing the distribution tree in Figure 9.7(b), it is clear that the relative delay penalty from A to all of members B, C and F is 1.0, as no additional routing is taking place, and A is forwarding packets directly to those peers.



(a) The initial logical network topology.

(b) Selection of the host to be killed. While TCP connections will close, member and link state will remain intact.

(c) The logical shape of the network after the fixing mechanism has run, showing the network has repaired itself.

(d) The logical shape of the network when peers have formed additional connections to reconnect both sides of the mesh.

Figure 9.8: Logical view of a peer group both before and after a partition has occurred.

The RDP from A to D, however, is easily calculated as $\frac{140ms+50ms}{150ms} = 1.267$, and the RDP from A to E is $\frac{140ms+45ms}{145ms} = 1.276$.

The ideal relative delay penalty is 1.0, offered by a true IP Multicast distribution tree or a naïve overlay, and these figures approach 1.0 due to the overlay only routing a packet across the high latency link when it has no other route to the rest of the group.

Without the ability to test the RDP over larger groups with realistic latencies in place, it is difficult to draw further conclusions about the RDP values returned by Orta. RDP values will naturally rise as the group size increases, provided the overlay is not adding links to every peer in the group, and instead holding onto some reasonable number links betweenmembers, but a larger test network or a network simulator would be required to derive by how much exactly the RDP values would increase. An additional issue with the RDP calculation over the dummynet is that mentioned in Section 9.5, of alternate routes through the network being possible for different connections, therefore offering different latency characteristics.

Despite not having a large enough network available to test this metric further, the numbers derived here look promising for the Orta protocol.

9.7 Time Taken to Repair a Partitioned Mesh

The logical network topology used to test the ability of the group to fix a partitioned mesh is shown in Figure 9.8(a).

In order to form a partition, the code responsible for probing group members and initiating new connections to those members was disabled. With that code disabled, a logical chain of peers was created as per Figure 9.8(a), and shortly afterward, host D was killed (via a “`kill -9`” command). The actual timeline of events from the initial kill signal until the mesh was fully repaired was as follows:

1:	0 seconds –	Member D killed.
2:	43 seconds –	First new connection made to a silent host on the other side of the partition.
3:	72 seconds –	Member D declared dead by another group member.
4:	190 seconds –	All member state brought back up to date.

Not shown on the diagrams are the attempts made by group members to connect to group member D, between stages 1 and 3 in the table above. In terms of how the peers were actually behaving here, peers were attempting to form a TCP connection to D, which was being refused (as no process was running to accept the connection). The peers did not clear the member and link state of that member until the timeout had expired to declare the member dead; in the implementation, this is set to 70 seconds.

State changes on one side during the partition are not observed on the other, hence the time taken for the routing tables to provide valid distribution trees again. Peers must wait for refresh packets to arrive in order to update the link-state. This requirement for link-state information to be readily available, preferably within the next refresh cycle, is at odds with the bandwidth saving suggestion in Section 6.3 that if the state of a link has not changed recently, then a host sends information about that link less often. This additional delay in the sending of some link state explains why it took so long for the group to arrive at step 4 in the table above.

The time taken to repair the mesh is a side-effect of two independent mechanisms. First, the time taken to rejoin the two halves of the mesh and declare the killed member as being dead, and second, the time taken to remove the dead member from the local state of each peer.

The time taken to repair the mesh here is clearly of little use for a conferencing application. The partition repairing scheme, taken from Narada, that Orta uses provides is not suited for this type of application. Given that TCP connections are used to maintain connections between members, it would be reasonable to assume that if a connection is closed unexpectedly, without any prior notification from the peer, then that peer has failed unexpectedly. All neighbours would flood this information outward; even if the mesh became partitioned through the death of this peer, the act of removing the relevant member and link state immediately at each group member aids the reconstruction of routing tables on reconnecting the mesh. As is normal on receipt of a member leave notification, peers recalculate their routing tables; it would not be entirely difficult during this process to spot members who are no longer reachable, and add links to those members until all members can be reached once again.

While this would require a lot of communication and computation, it would be feasible to fix a partitioned mesh due to failure of a peer within seconds. The only issue might still be that of link state changing on either side of a partition before the mesh is reconstructed, even though the partition may only exist for a matter of seconds; an easy solution might be that on noticing a member which is unreachable, to ensure that all link state is retransmitted at the first refresh cycle following the repartitioning effort. This, at least, would guarantee for the total recovery of all distribution trees within 30 seconds.

It is worth noting that under normal operation a partition formed in the mesh, perhaps from a member failing, will be repaired before the code dedicated to fixing a partition will be initiated. The reason for this is that the mechanism to randomly ping peers for the purposes of evaluating new links is likely to ping a member on the other side of the partition. In this situation, it will still take some time for the local state at all peers to be repaired if any state has changed during the partition.

Monitoring TCP connections does not protect the mesh from all member failures, however. TCP state is maintained in software, and a severed physical link, or a machine powered down unexpectedly, does not close a TCP connection. In fact, the connection would likely timeout long after the mesh had noticed that peer was not responding. If more complex methods of tracking responses to normal ping packets to neighbours were employed, a peer would assign some large weight, less than infinity, to the link in question, thus allowing for the mesh to route around a failed peer before it has even been declared dead. With a well-formed mesh, the routing changes happen swiftly; if the peer formed the only route between two halves of the mesh, normal link discovery mechanisms would quickly allow for new links to route over. In either eventuality, it appears that it should be possible to repair the mesh in a matter of seconds, rather than minutes.

The mechanisms in place for fixing a partition should be replaced before Orta can be used for real-time applications.

9.8 Volumes of Control Traffic Sent

This experiment made use of the undergraduate lab machines; each of the various peer group sizes tested lasted for the same five minutes. The nature of the flooding process to distribute control information deserved attention, and so the volume of control traffic sent at various group sizes was monitored.

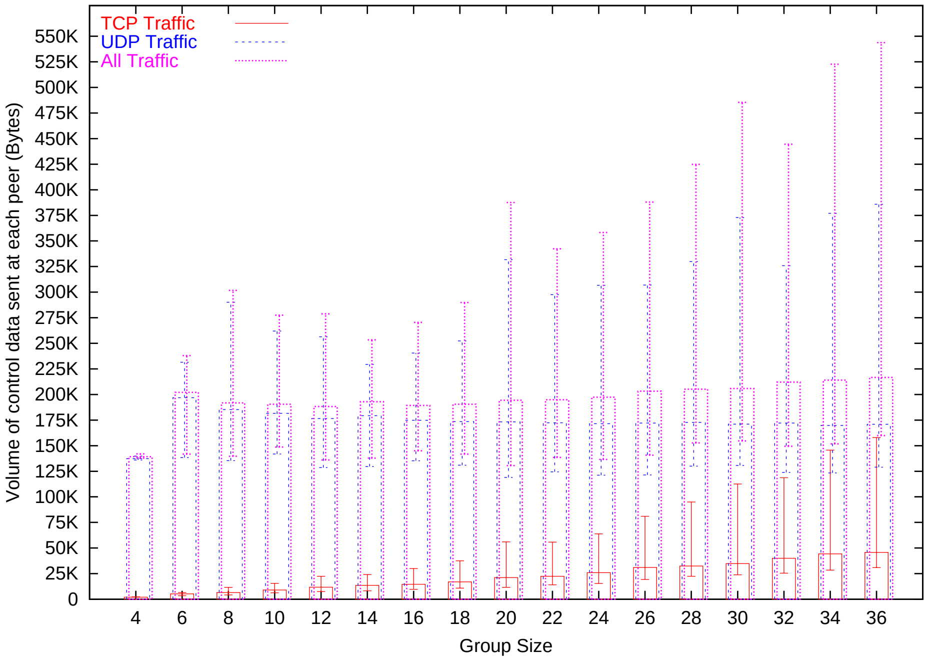
The average volume of control traffic sent by each group member over TCP, as shown in Figure 9.9, appears to rise linearly. This makes sense, in that provided link state remains reasonably constant, then most TCP traffic being sent is the regular refresh packets from each peer. The average volume of UDP traffic sent over the lifetime of the group appears to remain constant, presumably due to the number of neighbours each peer has in the mesh does not vary significantly. As group sizes increase, so too does variation in the maximum volume of data sent on each type of connection, suggesting that even though the average number of neighbours a peer has in a group remains constant, larger groups see some peers with larger neighbour sets than others. More neighbours in the mesh requires that a peer flood or forward a flood message to more links when it receives such a message, and likewise, sends more ping packets to existing neighbours over the UDP sockets, hence the increase in volume of data sent by each traffic type. Given that these experiments were run on lab machines, no conclusions can be drawn about the connectivity of the hosts which attract most connections.

Plotting the total control traffic sent over the whole peer group in Figure 9.9(b), we see that the total volume of control data required to maintain the group rises linearly with group size. This is a useful property in that the limiting factor for larger group sizes is not the volume of control data, but may actually the amount of computation involved at each peer. Further testing would be required to determine if this was the case.

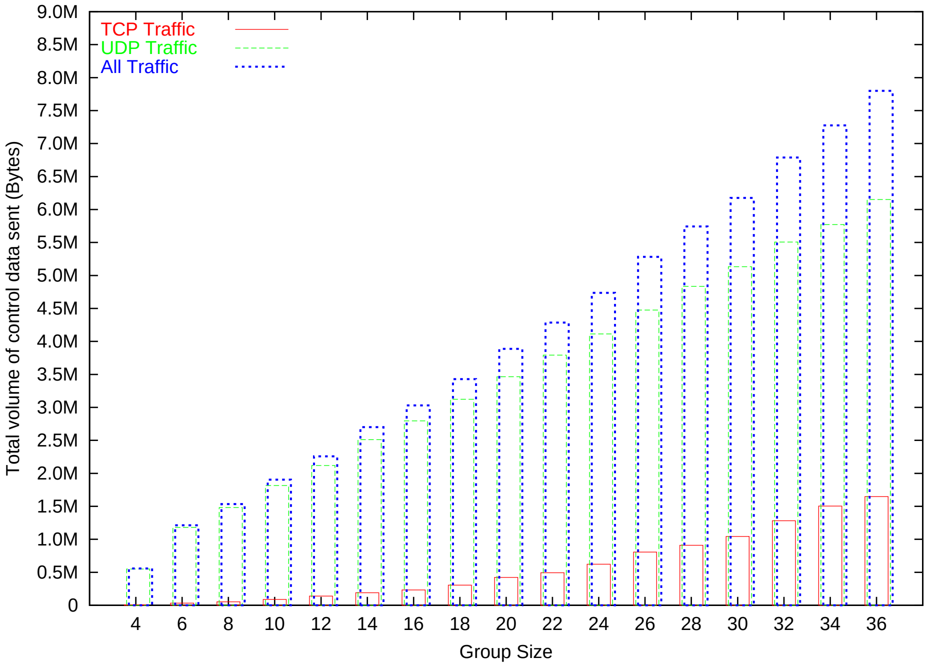
The volume of control traffic sent at group sizes 4 and 6 over the dummynet were similar to volumes sent at those group sizes on the lab machines. Larger networks with ‘real-world’ properties would have to be tested or simulated to determine whether this holds for larger group sizes.

9.9 Reliability of Overlay: Lost & Duplicated Packets

This experiment was run for 5 minutes at all varying group sizes, with each host transmitting an increasing sequence number from a test application every 20ms. A peer generating a new data packet every 20ms will

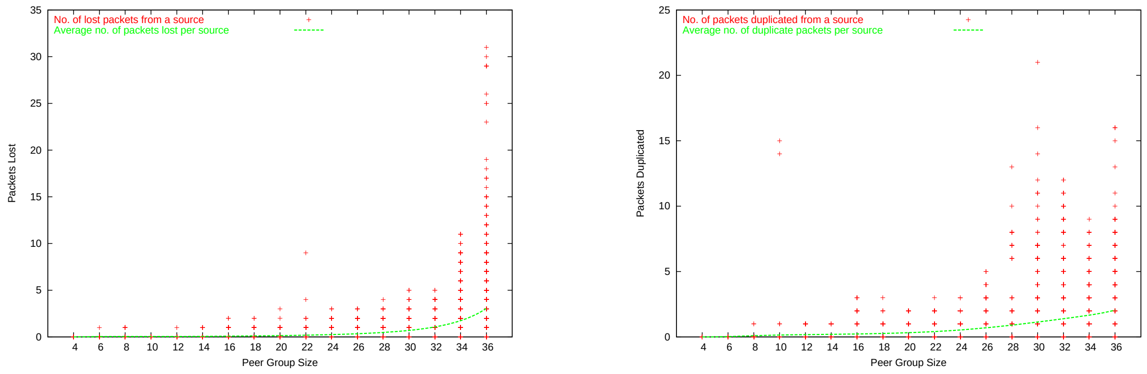


(a) Average volume of control traffic sent from each host, including minimum/maximum variation in this value. Graph shows TCP traffic, UDP traffic, and the combined total.



(b) Total volume of control traffic sent throughout the entire group during a session at each group size. Graph shows TCP traffic, UDP traffic, and the combined total.

Figure 9.9: Variation in volumes of control traffic sent at varying group sizes during the lifetime of the mesh at various group sizes.



(a) Graph of packets lost over varying group sizes. Total number of packets lost for each group member, from any other individual group member, at varying group sizes.

(b) Graph of packets duplicated over varying group sizes. Total number of packets lost for each group member, from any other individual group member, at varying group sizes.

Figure 9.10: Packets lost and duplicated for varying group sizes, all peers sending data every 20ms.

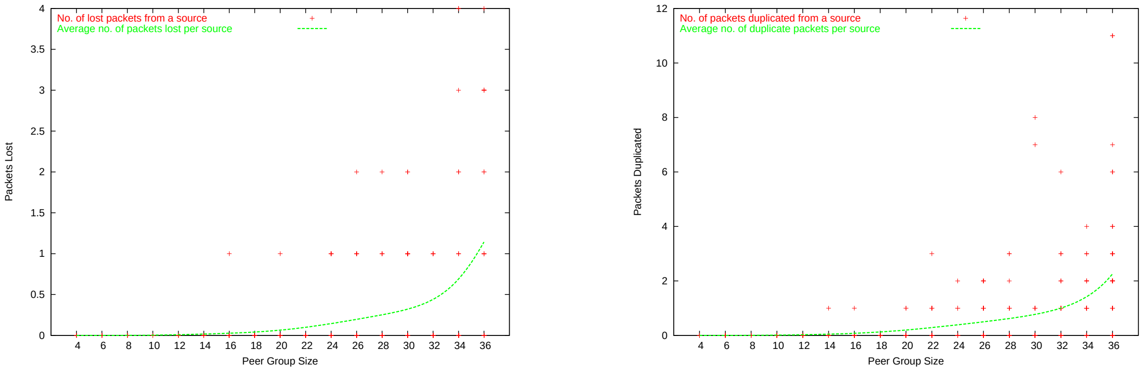
generate up to 15,000 packets during the lifetime of the overlay. The test code monitors packets received from each host, and logs both packets lost and packets duplicated from every other host. Since streams of data from each host in an audio conference are completely separate, we only need consider the packets lost from any individual source, not cumulative packet loss from the entire group.

Results from this first run can be seen in Figure 9.10. Both packet loss and packet duplication rates appear acceptable for all group sizes used, with higher loss rates due to more frequent reconfiguration of links and inefficient computation at end hosts.

Some packet loss should be expected due to normal reconfiguration of the mesh. This is particularly true of running such group sizes over a local area network where latencies between hosts are measured in microseconds, rather than milliseconds. While the number of links in the group is reasonably stable (varying by at most ±4 links for larger groups), there is still some variation in how many links each peer holds onto. Erroneous ping times can lead to incorrect latencies being reported, causing the mesh to add or drop links. Packet losses and duplications due to this natural reconfiguration are what we can see at all group sizes, and explains the increase in packet loss and duplication rates at larger group sizes. Duplicate packets arise for the same reason of periods of misconfiguration.

It’s worth noting that as these tests were run, by necessity, on machines all occupying the same LAN, that ping times were notoriously close to each other, and generally reported to be the floor value of 3ms (some larger, presumably due to unfortunate scheduling of the user processes by the Linux kernel).

The graphs shown here do not demonstrate the possibility of a peer leaving, and splitting part of the mesh structure, though this would be a very real concern under more realistic settings, with members joining and leaving at more arbitrary moments than simply leaving after a well-defined 5 minutes. One possible solution to force a member to keep at least two links to the rest of the mesh might be to require that a an absolute threshold value close to zero for the case of a peer having 1 neighbour be used instead of the algorithm nor-



(a) Total number of packets logged as being lost from the source by each group member, at varying group sizes.

(b) Total number of packets logged as lost from the source by each group member, at varying group sizes.

Figure 9.11: Packets lost and duplicated over the lifetime of various sizes groups.

mally used to generate a threshold value. This would require further investigation, to determine whether the setting of an absolute threshold value for some group members would upset the stability of the mesh, particularly if that threshold was significantly different from that of what would otherwise be calculated.

One further issue would be that of attempting to balance the threshold in groups large enough that routing tables at peers could become large enough to overflow outgoing UDP buffers, thus losing packets, such that this does not happen, but also that peers still retain at least two neighbours, and distribution trees do not become too deep.

9.10 Discussion of the Threshold Value

Both the Narada protocol and the Orta protocol rely heavily on the definition of a threshold value, which acts as a seemingly arbitrary number that dictates whether any given link is worthy of being included in the mesh.

It appears from the Dummynet structures shown above that the chosen threshold value works and generates correct meshes (in that they are fully connected), but it is not capable of providing at least two links to each node in the group at all times.

Part of the difficulty lies in finding a threshold value that works for varying group sizes, over varying network topologies. For example, peers on the lab network grouped together to achieve between 2 and 5 neighbours, but this behaviour was not reflected on the dummynet networks tested.

Further difficulty in choosing the threshold value is that it causes the overlay to behave differently in different types of environments. It might be beneficial to have different thresholds ‘hardwired’ for different environments, so a threshold might be more appropriate for conferencing between home users over ADSL links, while another might be better for groups of users on the same LAN. Adaptation of the threshold calculation during the lifetime of the overlay could be achievable, perhaps based on group size, number of sources, variability of link types, latencies between group members, to name a few. Evolutionary algorithms could perhaps be of use here, to allow the system itself to decide what threshold to use based on results from past attempts at threshold values. This approach would require substantial testing before real-world usage.

9.11 Summary

Orta appears to provide desirable properties for real-time applications. The volume of control traffic sent during the lifetime of a peer group is reasonably predictable, based on group size. Round trip times achieved on the test networks are within the limits required for conversational audio to take place, and packet loss and duplication rates are minimal. Orta achieves worst case stress of substantially less than that of a naïve unicast application. While more testing over a larger variety of networks would be required to boost confidence in the new protocol, these results are promising.

Mechanisms used to fix partitions are unsatisfactory, however. The packet loss incurred at some peers would be too high, especially if the algorithm takes in the order of minutes to repair the mesh. Potential improvements to this algorithm have been highlighted in Section 9.7.

In summary, the Orta protocol defines a peer-to-peer overlay capable of carrying real-time data, which demonstrates useful properties for the carrying of this data. While there are some issues to be resolved when a partition has to be fixed, these issues are well understood, and modification to the protocol would allow for swift reconfiguration of the mesh structure, leading to repair in seconds rather than minutes.

Chapter 10

Conclusions & Future Work

10.1 Future Work

Further investigation is required into determining the ideal threshold values for different environments, and into trying to find a method of calculating threshold values that allow for groups of variable size to stabilise, and which enforces that each peer has at least two links to other members of the group. The main issue with the threshold value is that it's arbitrary; if the threshold could adapt during the runtime of the system (without destabilising the mesh by doing so, as thresholds are calculated locally, not globally), the group would theoretically be able to adapt to different network environments. For instance, using the same threshold value for the dummynet experiments and the lab-based experiments in Chapter 9.

Currently, the adding of new links is performed if the utility of that link is deemed to be above some arbitrary threshold. One method of removing the arbitrary threshold might be to add a link if it is roughly as good as or better than the links already in the group; removal of links could occur if a link is essentially redundant, but a link would be left in place if it left the local peer or the foreign peer with only one link remaining to the group.

The work presented in [28] offers a way of scaling up peer-to-peer groups which use mesh-based approaches to generating distribution trees. It would be interesting to extend the Orta protocol to accommodate these clustering techniques to allow for a mesh of meshes; the addition of more layers would make it more difficult to meet the 400ms upper bound on RTT for conversational audio, but would reduce the volume of control traffic sent, and allow for larger groups to be generated. This is particularly relevant for Orta, where the volume of computation increases substantially as the size of the peer group increases. The problem faced by this hierarchical approach is of how to cluster the group members into individual clusters, so the problem becomes both a problem of how to cluster peers, and then to form a mesh within those clusters. An idealised view of this algorithm might be to have a UK cluster, and a US cluster, with the clusters themselves exchanging minimal levels of control information.

As stated in Section 7.7, there are inefficiencies in the packet formats used, in that fields aren't packed tightly into the packet. Fields which could fit into 1 or 2 bits take the form of a 32 bit unsigned int. During a redesign, it would not be difficult to leave a few bits to signify what hierarchy level a control packet was intended for (the only nodes that should ever see packets from a different hierarchy level being those who are elected as leaders for their cluster).

Perhaps further experimentation with sending control information over UDP, perhaps merging the control and data channels to use the same port (using the channel abstraction presented to the application), and removing explicit “new member” flood packets in favour of the soft-state refresh flood mechanism.

With computation becoming an issue for Orta peers at larger group sizes, it would be possible to cutting down computation at every peer in the group by having peers advertise themselves as participants in the session (i.e.: sources of data), or merely observers of the session. This might be useful for scenarios such as lectures or presentations. Shortest path trees would not have to be calculated for those hosts who are merely observers, though observers would still have to calculate shortest path trees from the sources. An alternative method of distributing routing state might be to have participants generate their own shortest path spanning tree over the link-state and flooding this tree structure (with each receiving peer noting the parts of this data structure relevant to themselves, and forwarding it on to neighbours). This latter approach might be beneficial on topologies where the link-state is not likely to change frequently, and there are few sources of data.

Given that the intention of Orta was to carry real-time audio, there is the possibility of mixing audio streams in transit, thus reducing the number of data packets which must be sent at each peer. This would require that an overlay layer were aware of the type of traffic is were carrying, and as such an additional layer between the plain overlay and the application could be designed. The ideal would be to keep most routing decisions completely generic, but to allow some knowledge of the application which the overlay was serving.

Investigation into and implementation of the ideas discussed in 9.7 would allow for faster repartitioning, more suited to real-time applications. This would alleviate some of the issues regarding members leaving, but should not be considered the ideal solution. Having to repair a damaged mesh should be considered a last resort for the protocol to take. Some effort would have to go into investigating how best to proceed when a peer has failed or left the group, causing a partition.

Further, Orta could introduce bandwidth-saving measures such as probabilistically choosing the group member which offers the best chance for a good new link from existing link state, or reducing the rate of random ping packets to members which the protocol has tried a few times already, assuming that if a connection wasn’t acceptable on the first probe, the chances of it being acceptable on the second probe are slim.

10.2 Conclusions

By altering the mechanism the Narada protocol used for the distribution of control state to group members from a distance vector algorithm to a link state flooding algorithm, the Orta protocol allows for a more responsive peer-to-peer overlay, geared toward the carrying of real-time audio between many recipients.

While this change has brought with it increased computational loads at all peers in the group, it has allowed for a more accurate mechanism for the purpose of dropping links, and also removed the requirement on peers to continue to forward data for some time after leaving the group.

The immediate advertisement of a new member to the group allows the new member to participate with the group immediately (leaving the overlay to attempt to improve itself gradually over time). The current state of the protocol is not entirely promising in the situation that the mesh structure becomes partitioned. Potential improvements to this algorithm have been discussed in Section 9.7, but essentially involve taking advantage of the nature of TCP connections between neighbours to facilitate swift repartitioning of the mesh; more concrete methods of updating member and link state throughout the peer group after a partition may be required to repair the mesh state. Once improvements are made to this part of the protocol to facilitate much faster repartitioning, the Orta protocol appears very much to be useful for the purposes of conferencing applications.

The only other major obstacle the protocol has to counter is that of the threshold used to determine whether or not a link should be added or dropped from the mesh; threshold value calculation currently has to be almost hardcoded to the type of network that the overlay will be running across. This is certainly not ideal, and is an area that might require considerable effort to derive a proper solution.

To conclude, Orta is a new protocol appropriate for the carrying of real-time data to small or medium-sized conference groups, in the order of 10s of members. The protocol offers distribution trees optimised for each source, and will reconfigure in light of current network conditions, offering ideal conditions for the carrying of real-time data. As has been demonstrated, the implementation of Orta is capable of carrying data from many sources to all members of the group at the same data rates as a real-time audio conferencing application might send, a fact backed up by the implementation of RAT using the Orta library for data carrying.

Appendix A

The net_udp API

Functions are listed with a list of occurrences by file (from the root of the source directory) and line number.

Note that Mbus entries, in other words any relevant function calls which appear in `mbus.c`, was not be replaced, as it was the mechanism through which the core components of the application communicated.

Now that overlay code is more stable, the Mbus entries should theoretically be replaceable.

Function: `udp_addr_valid( const char* addr )`
Occurrences: `rat/main_control.c:80`
Notes: This function simply checks the validity of an IP address or hostname. This can be copied & pasted directly into Narada code, for the purposes of completeness.

Function: `udp_init( addr, rx_port, tx_port, ttl )`
Occurrences: `common/src/mbus.c:470`
`common/src/sap.c:81`
Notes: Neither occurrence of this function is called from RAT code. At any rate, this function merely calls udp_init_if(), with iface set to NULL.

Function: `udp_init_if( addr, iface, rx_port, tx_port, ttl )`
Occurrences: `common/src/rtp.c:1054`
`common/src/rtp.c:1055`
Notes: Essentially binds rx_port to a socket descriptor. This function contains lots of multicast code, making it look scarier than it actually is. tx_port is not used directly, but is held in the struct socket_udp* which is returned by this function.

Function: `udp_recv( socket_udp, buffer, buflen )`
Occurrences: `common/src/mbus.c:761`
`common/src/rtp.c:1359`
`common/src/rtp.c:1879`
`common/src/rtp.c:2925`
`common/src/sap.c:106`
Notes: This function simply lifts incoming data from the socket descriptor in socket_udp.

Function: `udp_send( socket_udp, buffer, buflen )`
Occurrences: `common/src/mbus.c:340`
`common/src/rtp.c:2239`
`common/src/rtp.c:2674`
`common/src/rtp.c:2862`
Notes: Simply performs a sendto() with the socket descriptor found in the struct socket_udp; replace with a mesh_t type.

Function: `udp_sendv( ... )`
Occurrences: `common/src/rtp.c:2320`
Notes: This function is WIN32 only.

Function: `udp_host_addr( socket_udp )`
Occurrences: `common/src/rtp.c:918`
`common/src/rtp.c:1064`
Notes: Gets local hostname; checks that hostname is valid. Straight copy and paste of code, for completeness. Outcome of second call here is passed straight into init_rng, i.e.: INITiate Random Number Generator.

Function: `udp_fd( socket_udp )`
Occurrences: Seemingly unused.
Notes: Simply pulls the socket descriptor out of the struct socket_udp. Is there a reasonable equivalent to this? Provide tags for groups? Create a group ID (easy as local host name + timestamp...).

Tag becomes some equivalent to the socket descriptor? Or, given that only one socket will be open for sendto/recvfrom for UDP transfers, can we just expose this to higher layers?

Function: `udp_select( timeout )`
Occurrences: `common/src/rtp.c:1872`
`common/src/rtp.c:2923`
`common/src/sap.c:102`
Notes: -

Function: `udp_fd_zero( void )`
Occurrences: `common/src/rtp.c:1869`
`common/src/rtp.c:2921`
`common/src/sap.c:100`
Notes: Clears set of file descriptors available for reading by select().

Function: `udp_fd_set( socket_udp )`
Occurrences: `common/src/rtp.c:1870`
`common/src/rtp.c:1871`
`common/src/rtp.c:2922`
`common/src/sap.c:101`
Notes: Adds a socket descriptor to the read set for reading by select().

Function: `udp_fd_isset( socket_udp )`
Occurrences: `common/src/rtp.c:1873`
`common/src/rtp.c:1876`
`common/src/rtp.c:2923`
`common/src/sap.c:103`
Notes: Queries whether or not a given file descriptor is in the read set ready for reading by select().

Appendix B

The Orta API

Function: `int mesh_addr_valid( const char *addr );`

Description: Returns TRUE if the given address (hostname or IPv4 address) is valid, FALSE otherwise.

Function: `const char *mesh_host_addr( );`

Description: Returns a character string containing the external network address for the local host.

Function:

```
int mesh_init(mesh_t **m, uint16_t udp_rx_port, uint16_t
udp_tx_port, int ttl);
int mesh_init_if(mesh_t **m, const char *iface, uint16_t
udp_rx_port, uint16_t udp_tx_port, int ttl);
```

Description: Creates overlay state for use on connecting to an existing mesh, or to allow incoming connections from other hosts. Returns a pointer to a valid mesh_t structure on success, NULL otherwise. The iface version is currently in place for completeness against the net_udp interface; the iface argument will simply be ignored.

Function: `int mesh_connect( mesh_t *m, const char *dest );`

Description: Attempts to connect to a peer already in the peer-group using 'addr', which should be a character string containing an IPv4 network address.

Function: `int mesh_disconnect( mesh_t *m );`

Description: Closes all connections into peer-group cleanly, and clears state held within the mesh_t struct.

Function: `void mesh_destroy( mesh_t **m );`

Description: Destroys the state used by 'mesh' and frees up any occupied memory.

Function: `int mesh_register_channel( mesh_t *m, uint32_t channel );`

Description: Registers a new data channel at this host for the purposes of receiving UDP data on that particular channel. Channels can be seen as an emulation of system ports.

Function:

```
int mesh_select( mesh_t *m, struct timeval *timeout, int*
channels, int *count );
```

Description: Scans the data channels until data is available on at least one of them.

Function:

```
int mesh_recv( mesh_t *m, uint32_t channel, char *buffer, int
buflen );
int mesh_recv_0( mesh_t *m, char *buffer, int buflen );
int mesh_recv_timeout( mesh_t *m, uint32_t channel, char
*buffer, int buflen, struct timeval *timeout );
```

Description: The recv calls all deal with removing data from the input queues from the network; `_recv()` is the basic call, in which a channel must be specified. It blocks until data has been retrieved, and on retrieval returns the length in bytes of that data. If an invalid channel is specified, -1 is returned.

`_recv_0()` is exactly the same as the first call, except it's hardwired to check channel 0. Allows for a slightly simpler interface into the overlay if only one channel is required.

`_recv_timeout()` behaves exactly as the first call, except it will also return -1 should the timeout be hit before any data arrives on the channel specified.

Function:

```
int mesh_send( mesh_t *m, uint32_t channel, char *buffer, int
buflen );
int mesh_send_0( mesh_t *m, char *buffer, int buflen );
```

Description: The send calls handle the bundling of data and forwarding to the relevant hosts in the overlay; the application need not be aware which hosts the overlay is forwarding to.

As with the recv calls, `_send()` sends the data to the channel specified, and `_send_0()` sends on channel zero, should only one channel be required.

Appendix C

Modifications made to RAT

The following details the modifications made to the software tool RAT, [4], to make use of the Orta library presented here. Patches are presented here in sections C.2.1 and C.2.2. A minimal number of source code modifications were required, and aside from modifications to the Makefiles present in the `rat` and `common` directories, only `rtp.c` had to be altered.

The modifications made to `rtp.c` were generally to replace calls which were previously made to functions in `net_udp.c`, which are themselves covered in Appendix A. These are the functions RAT uses to send and receive real-time data. Had the functions contained in `net_udp.c` been used only for the transmission of real-time data, then the required modifications could have been placed in there. However, the main components of RAT communicate using a message bus (mbus, [37]) system, which makes use of these functions. To aid stability, the decision was taken to leave the `net_udp.c` functionality in place for use by the mbus functions, and instead point the RTP calls to the Orta functions. Had the mbus not made use of these function calls, calls into Orta would have been made within the `net_udp.c` functions to directly replace system network calls.

New function calls are merely fenced off by an `#ifdef` ... `#endif`, to be activated if the ORTA tag is defined during compile time.

C.1 Data Structures and Constants

The RTP code in the `common` source tree bundles all RTP state up into one large struct; within this struct are two UDP sockets kept for the purposes of sending and receiving RTP data, and RTCP data [44]:

- rtp_socket
- rtcp_socket

To make `rtp.c` work with the Orta library instead then, the sockets were no longer necessary. The sockets can be replaced with one reference to a `mesh_t` structure. In other words, simply declare

- mesh_t $*$mesh

Thus the RTP code carries with it a reference into the overlay, from where it can send and receive data, after it has connected to a group.

To emulate the two ports, the patched code defines two Orta channels, one for RTP data and one for RTCP data. On receiving any data on either of these channels, the same code as was run prior to the patch is executed.

C.2 Patches

The following patches can be applied to code obtained from anonymous CVS servers at UCL[1], as was current as of the end of May, 2005.

C.2.1 rat patch

```
diff -Naur rat/Makefile.in /users/students8/level8/strowesd/Project/src/rat/Makefile.in
--- rat/Makefile.in 2005-02-01 08:55:15.000000000 +0000
+++ /users/students8/level8/strowesd/Project/src/rat/Makefile.in 2005-05-28 03:30:10.000000000 +0100
@@ -5,11 +5,15 @@
 #
 # Configure substitutes variables here... #####################################

+ORTASRC = ../mesh/
+ORTANAME = mesh
+
+
 DEFS    = @DEFS@ -DHIDE_SOURCE_STRINGS
 CFLAGS  = @CFLAGS@ $(DEFS)
-LIBS    = @LIBS@ @MATHLIBS@
+LIBS    = @LIBS@ @MATHLIBS@ -L$(ORTASRC) -l$(ORTANAME) -lpthread
 LDLIBS  =
-INCLUDE = @COMMON_INC@ @AU_INC@ @TCL_INC@ @TK_INC@ @G728_INC@
+INCLUDE = @COMMON_INC@ @AU_INC@ @TCL_INC@ @TK_INC@ @G728_INC@ -I$(ORTASRC)
 CC      = @CC@
 AR      = ar
 RANLIB  = @RANLIB@
```

C.2.2 common patch

```
diff -Naur common/examples/rtp/Makefile.in /users/students8/level8/strowesd/Project/src/common/examples/rtp/Makefile
--- common/examples/rtp/Makefile.in 2001-04-04 14:36:36.000000000 +0100
+++ /users/students8/level8/strowesd/Project/src/common/examples/rtp/Makefile.in 2005-05-26 02:29:38.000000000 +0100
@@ -4,14 +4,17 @@
 #

 # Location of includes and library
-CSRC   = ../../src
+CSRC    = ../../src
+ORTASRC = ../../../mesh/

 # Library name
-LNAME  = uclmmbase
+LNAME    = uclmmbase
+ORTANAME = mesh

-DEFS   = @DEFS@
+DEFS   = @DEFS@ -DORTA
 CFLAGS = @CFLAGS@ $(DEFS) -I$(CSRC)
-LIBS   = @LIBS@ -L$(CSRC) -l$(LNAME)
+LIBS   = @LIBS@ -L$(CSRC) -l$(LNAME) -L$(ORTASRC) -l$(ORTANAME) -lpthread
+INC    = -I$(ORTASRC)
 CC     = @CC@
```

[1] :pserver:cvsanon@scary.cs.ucl.ac.uk:/cs/research/nets/common0/starship/src/local/CVS_repository

```
 TARGET = rtpdemo
diff -Naur common/src/Makefile.in /users/students8/level8/strowesd/Project/src/common/src/Makefile.in
--- common/src/Makefile.in 2003-05-28 12:38:56.000000000 +0100
+++ /users/students8/level8/strowesd/Project/src/common/src/Makefile.in 2005-05-26 01:58:23.000000000 +0100
@@ -3,9 +3,10 @@
 # This probably requires GNU make.
 #

-DEFS   = @DEFS@
+DEFS   = @DEFS@ -DORTA
 CFLAGS = @CFLAGS@ $(DEFS)
-LIBS   = @LIBS@
+INC    = -I../../mesh/
+LIBS   = @LIBS@ -L../../mesh/
 CC     = @CC@
 AR     = ar
 RANLIB = @RANLIB@
diff -Naur common/src/rtp.c /users/students8/level8/strowesd/Project/src/common/src/rtp.c
--- common/src/rtp.c 2004-11-25 17:25:22.000000000 +0000
+++ /users/students8/level8/strowesd/Project/src/common/src/rtp.c 2005-05-28 14:52:31.000000000 +0100
@@ -61,6 +61,10 @@

 #include "rtp.h"

+#ifdef ORTA
+#include "mesh.h"
+#endif
+
 /*
  * Encryption stuff.
  */
@@ -97,6 +101,13 @@
 #define RTCP_BYE  203
 #define RTCP_APP  204

+
+#ifdef ORTA
+#define RTCP_CHANNEL 1
+#define RTP_CHANNEL 2
+#endif
+
+
 typedef struct {
 #ifdef WORDS_BIGENDIAN
  unsigned short  version:2; /* packet type            */
@@ -218,8 +229,12 @@
  */

 struct rtp {
+#ifdef ORTA
+ mesh_t          *mesh;
+#else
  socket_udp *rtp_socket;
  socket_udp *rtcp_socket;
+#endif
  char *addr;
  uint16_t  rx_port;
  uint16_t  tx_port;
@@ -866,7 +881,7 @@

 #define MAXCNAMELEN 255

-static char *get_cname(socket_udp *s)
+static char *get_cname(struct rtp *s)
 {
         /* Set the CNAME. This is "user@hostname" or just "hostname" if the username cannot be found. */
         const char              *hname;
@@ -915,7 +930,11 @@
 #endif

         /* Now the hostname. Must be dotted-quad IP address. */
+#ifdef ORTA
+ hname = mesh_host_addr();
+#else
         hname = udp_host_addr(s);
+#endif
  if (hname == NULL) {
  /* If we can’t get our IP address we use the loopback address... */
  /* This is horrible, but it stops the code from failing.         */
@@ -1051,17 +1070,38 @@
  session->rx_port = rx_port;
  session->tx_port = tx_port;
  session->ttl = min(ttl, 127);
-session->rtp_socket = udp_init_if(addr, iface, rx_port, tx_port, ttl);
-session->rtcp_socket = udp_init_if(addr, iface, (uint16_t) (rx_port+1), (uint16_t) (tx_port+1), ttl);
+#ifdef ORTA
+ mesh_init_if(&session->mesh, iface, rx_port, tx_port, ttl);
+
+ if (session->mesh == NULL) {
+ xfree(session);
+ return NULL;
+ }
+
+ if ( !strcmp( addr, "localhost" ) ) {
+ mesh_connect(session->mesh, NULL);
+ }
+ else {
+ mesh_connect(session->mesh, addr);
+ }
+ mesh_register_channel(session->mesh, RTCP_CHANNEL);
+ mesh_register_channel(session->mesh, RTP_CHANNEL);

  init_opt(session);
+ init_rng(mesh_host_addr());
+#else
+ session->rtp_socket = udp_init_if(addr, iface, rx_port, tx_port, ttl);
+ session->rtcp_socket = udp_init_if(addr, iface, (uint16_t) (rx_port+1), (uint16_t) (tx_port+1), ttl);

  if (session->rtp_socket == NULL || session->rtcp_socket == NULL) {
  xfree(session);
  return NULL;
  }

+ init_opt(session);
         init_rng(udp_host_addr(session->rtp_socket));
+#endif
+

  session->my_ssrc            = (uint32_t) lrand48();
  session->callback           = callback;
@@ -1108,7 +1148,7 @@

  /* Create a database entry for ourselves... */
  create_source(session, session->my_ssrc, FALSE);
-cname = get_cname(session->rtp_socket);
+ cname = get_cname(session);
  rtp_set_sdes(session, session->my_ssrc, RTCP_SDES_CNAME, cname, strlen(cname));
  xfree(cname); /* cname is copied by rtp_set_sdes()... */

@@ -1356,7 +1396,11 @@
  buffer12 = buffer + 12;
  }
```

```
+#ifdef ORTA
+ buflen = mesh_recv(session->mesh, RTP_CHANNEL, buffer, RTP_MAX_PACKET_LEN - offsetof(rtp_packet, fields));
+#else
         buflen = udp_recv(session->rtp_socket, buffer, RTP_MAX_PACKET_LEN - offsetof(rtp_packet, fields));
+#endif
  if (buflen > 0) {
  if (session->encryption_enabled) {
  uint8_t   initVec[8] = {0,0,0,0,0,0,0,0};
@@ -1866,6 +1910,28 @@
 int rtp_recv(struct rtp *session, struct timeval *timeout, uint32_t curr_rtp_ts)
 {
  check_database(session);
+#ifdef ORTA
+ /* FIXME: Constant buried in code. */
+ int channels_set[2];
+ int count, i;
+
+ if (mesh_select(session->mesh, timeout, channels_set, &count)) {
+ for ( i= 0; i<count; i++ ) {
+ if ( channels_set[i] == RTP_CHANNEL ) {
+ rtp_recv_data(session, curr_rtp_ts);
+ }
+ if ( channels_set[i] == RTCP_CHANNEL ) {
+ uint8_t  buffer[RTP_MAX_PACKET_LEN];
+ int  buflen;
+ buflen = mesh_recv(session->mesh, RTCP_CHANNEL, buffer, RTP_MAX_PACKET_LEN);
+ printf( "Recv RTCP\n" );
+ rtp_process_ctrl(session, buffer, buflen);
+ }
+ }
+ check_database(session);
+                return TRUE;
+ }
+#else
  udp_fd_zero();
  udp_fd_set(session->rtp_socket);
  udp_fd_set(session->rtcp_socket);
@@ -1882,6 +1948,7 @@
  check_database(session);
                 return TRUE;
  }
+#endif
  check_database(session);
         return FALSE;
 }
@@ -2236,7 +2303,13 @@
  buffer_len, initVec);
  }

+#ifdef ORTA
+ /*printf( "- Calling mesh_send.\n" );*/
+ rc = mesh_send(session->mesh, RTP_CHANNEL, buffer + offsetof(rtp_packet, fields), buffer_len);
+ /*printf( "- Call complete.\n" );*/
+#else
         rc = udp_send(session->rtp_socket, buffer + offsetof(rtp_packet, fields), buffer_len);
+#endif
  xfree(buffer);

  /* Update the RTCP statistics... */
@@ -2317,7 +2390,11 @@
  }

  /* Send the data */
+#ifdef ORTA
+ fprintf( stderr, "Windows functionality not implemented...\n" );
+#else
  rc = udp_sendv(session->rtp_socket, my_iov, my_iov_count);
```

```
+#endif

  /* Update the RTCP statistics... */
  session->we_sent     = TRUE;
@@ -2671,7 +2748,12 @@
  }
   (session->encrypt_func)(session, buffer, ptr - buffer, initVec);
  }
+#ifdef ORTA
+ mesh_send(session->mesh, RTCP_CHANNEL, buffer, ptr - buffer);
+ printf( "Sent RTCP\n" );
+#else
  udp_send(session->rtcp_socket, buffer, ptr - buffer);
+#endif

         /* Loop the data back to ourselves so local participant can */
         /* query own stats when using unicast or multicast with no  */
@@ -2859,7 +2941,11 @@
  assert(((ptr - buffer) % session->encryption_pad_length) == 0);
  (session->encrypt_func)(session, buffer, ptr - buffer, initVec);
  }
+#ifdef ORTA
+ mesh_send(session->mesh, RTCP_CHANNEL, buffer, ptr - buffer);
+#else
  udp_send(session->rtcp_socket, buffer, ptr - buffer);
+#endif
  /* Loop the data back to ourselves so local participant can */
  /* query own stats when using unicast or multicast with no  */
  /* loopback.                                                */
@@ -2918,13 +3004,23 @@
  timeout.tv_sec  = 0;
  timeout.tv_usec = 0;
  tv_add(&timeout, tv_diff(session->next_rtcp_send_time, curr_time));
+
+#ifdef ORTA
+ buflen = mesh_recv_timeout(session->mesh, RTCP_CHANNEL, buffer,
+    RTP_MAX_PACKET_LEN, &timeout);
+ if ( buflen != -1 ) {
+ rtp_process_ctrl(session, buffer, buflen);
+ }
+#else
  udp_fd_zero();
  udp_fd_set(session->rtcp_socket);
+
  if ((udp_select(&timeout) > 0) && udp_fd_isset(session->rtcp_socket)) {
  /* We woke up because an RTCP packet was received; process it... */
  buflen = udp_recv(session->rtcp_socket, buffer, RTP_MAX_PACKET_LEN);
  rtp_process_ctrl(session, buffer, buflen);
  }
+#endif
  /* Is it time to send our BYE? */
  gettimeofday(&curr_time, NULL);
  new_interval  = rtcp_interval(session) / (session->csrc_count + 1);
@@ -2981,8 +3077,13 @@
         }
  */

+#ifdef ORTA
+ mesh_disconnect(session->mesh);
+ mesh_destroy(&(session->mesh));
+#else
  udp_exit(session->rtp_socket);
  udp_exit(session->rtcp_socket);
+#endif
  xfree(session->addr);
  xfree(session->opt);
  xfree(session);
```

Appendix D

Project Management

The following sections cover the project timeline, the tools used to achieve the project aims, and how the work effort was divided.

D.1 Resources and Tools

A project of this nature requires many hosts on which to test the running of the system and ensure that not only is the code stable, but that the nodes are behaving correctly. To this end, not only were undergraduate lab machines used for the evaluation in Chaper 9, but they were also used for testing on progressively larger groups as the code stabilised. The machine at ISI, `kame.isi.edu`, was useful for testing of the code which dealt with latencies during development, since the latencies between lab machines was so small.

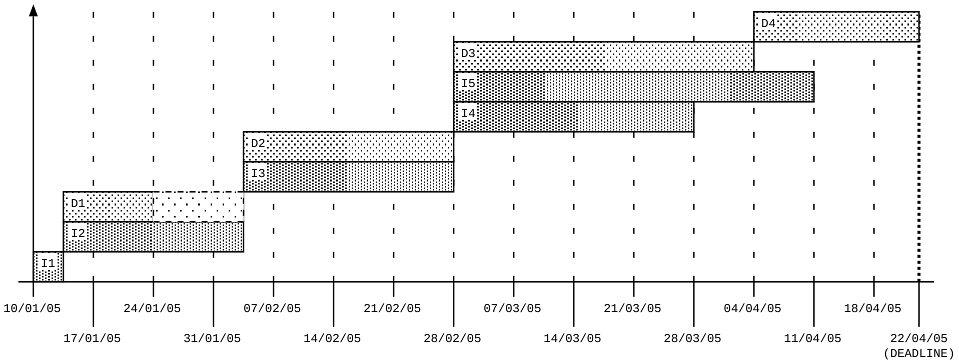
For the purposes of the evaluation then, as was covered in 8 and 9, both lab machines and a smaller network of machines connected by transparent bridges with dummynet functionality enabled were employed. In all these experiments, end hosts which ran test code was running some form of Linux; transparent bridges all ran FreeBSD 4.11,
The integration with RAT was tested over some of the machines in G081, though due to drivers only offering half-duplex audio on these machines, actual running of the application was not possible until Support installed the correct sound drivers for the hardware to work correctly, as RAT requires full-duplex audio hardware support.

Aside from the above, the usual array of tools was used to complete the project, including, but not limited to, LaTeX, gcc, gdb, awk, gnuplot, etc. The ability to chain many of these tools together and to build scripts to automate testing made the evaluation easier, if no less time consuming.

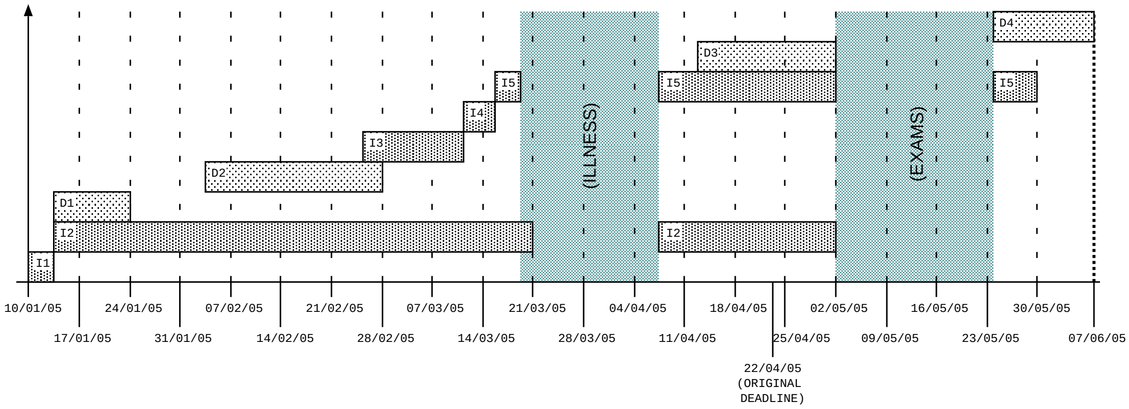
D.2 Division of Labour

All projects suffer unexpected delays, and this one was certainly no different. For an overview of the timeline of the project, see Figure D.1.

The original project timeline was as charted in Figure D.1(a). The labels on the tasks are detailed below.



(a) Original project timeline.



(b) Actual project timeline.

Figure D.1: Gantt charts charting project timeline.

Task Label	Description
I1	Design of sensible external APIs. Identify important code structures apparent before implementation begins.
I2	Implementation of the mesh structures.
I3	Implementation of data distribution mechanisms.
I4	Integration of code into RAT.
I5	End of testing; code freeze to focus on evaluation and dissertation.
D1	Initial document work, setting down LaTeXfiles and scripts, etc.
D2	Draft sections for the mesh structures.
D3	Draft sections for the distribution tree design, data delivery, routing, etc.
D4	Finalise dissertation.

As with any project schedule, it is difficult to predict what will happen in the future. As such, as can be seen in D.1(b), I3 took considerably less time to implement than I2 did, and debugging of I2 ran for most of the project. I4 was a step which was surprisingly easy to perform.

D.3 Project Log

12/01/05 Created basic LaTeXframework for report, splitting sections into (empty) `.tex` files.

17/01/05 Initial definition of data types to be presented to the application set in code.

18/01/05 Experimentation with packet formats; how to reliably transmit flatten data structures for sending, and reassemble structures at the receiving end. Some existing RTP code used for as example code.

23/01/05

27/01/05 Many peers able to connect to each other without too many stability problems. Members are now passing current known group membership tables to new peers on a new peer joining the group.

Looked into ICMP for sending of ‘real’ ping packets between peers; determined that timestamps in packets should work well enough, but over UDP rather than TCP.

04/02/05 Code restructuring, toward more of a link-state style of routing rather than distance vector routing. Peers are still sending link-state on a regular basis, rather than when link-state changes, and data structures are such that a shortest path algorithm can be run on top of them. These regular packets contain group membership and last known sequence number from each member, and a list of link states local to the sending peer. Custom ping packets are now being sent using UDP rather than TCP.

07/02/05 Dijkstra’s shortest path algorithm in testing.

08/02/05 Started working on code to evaluate the benefit of adding a new link.

17/02/05 Dijkstra’s shortest path algorithm now being used to calculate routing tables. Improved simple test harness to make use of mesh_send and mesh_recv functions. Also fixed code to run on FreeBSD.

23/02/05 Code cleanup. Removal of a busywait on the incoming data queue for data intended for the application, and instead signal when data has arrived on the queue. This emulates a normal blocking recv() quite nicely.

06/03/05 Major restructuring to do proper link-state flooding on major state changes (new member, dropped member, new link, removed link), with regular floods from each member to update the group on current link weights from each peer.

16/03/05 Code stabilising after the change to link-state flooding.

18/03/05 - 21/03/05 First bout of illness.

06/04/05 - 20/04/05 Continued debugging, starting of evaluation properly, building of scripts, etc. Outlined remaining dissertation sections.

21/04/05 - 01/05/05 Study break.

24/03/05 - 06/04/05 Second bout of illness.

02/05/05 - 24/05/05 Exam season.

25/05/05 - 07/06/05 Finalising of testing & evaluation, dissertation writing.

07/06/05 Final deadline.

Bibliography

[1] Homepage of the BitTorrent project. http://bittorrent.com/.

[2] Homepage of the Skype instant messaging and real-time audio conferencing application. http://www.skype.com/.

[3] Homepage of the UCL Network and Multimedia Research Group. http://www-mice.cs.ucl.ac.uk/multimedia/.

[4] Robust-Audio Tool homepage. http://www-mice.cs.ucl.ac.uk/multimedia/software/rat/.

[5] A. Adams, J. Nicholas, and W. Siadak. Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol Specification (Revised). http://www.ietf.org/rfc/rfc3973.txt.

[6] Z. Albanna, K. Almeroth, D. Meyer, and M. Schipper. IANA Guidelines for IPv4 Multicast Address Assignments. http://www.ietf.org/rfc/rfc3171.txt.

[7] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 205–217. ACM Press, 2002.

[8] S. A. Baset and H. Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol, September 2004.

[9] S. Birrer, D. Lu, F. E. Bustamante, Y. Qiao, and P. A. Dinda. Fatnemo: Building a resilient multi-source multicast fat-tree. In *WCW*, volume 3293 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2004.

[10] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth content distribution in a cooperative environment. In *IPTPS'03*, February 2003.

[11] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 2002. To appear.

[12] Y. Chawathe. Scattercast: an adaptable broadcast distribution framework. *Multimedia Syst.*, 9(1):104–118, 2003.

[13] Y. Chu, A. Ganjam, T. S. E. Ng, and S. G. Rao. Early experience with an internet broadcast system based on overlay multicast, 2004.

[14] Y. Chu, S. G. Rao, S. Seshan, and H. Zhang. Enabling conferencing applications on the internet using an overlay multicast architecture. In *ACM SIGCOMM 2001*, San Diago, CA, Aug. ACM.

[15] Y. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Measurement and Modeling of Computer Systems*, pages 1–12, 2000.

[16] D. Clark. The design philosophy of the DARPA internet protocols. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 106–114. ACM Press, 1988.

[17] M. Coates and R. Nowak. Network tomography for internal delay estimation. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '01)*, volume 6, pages 3409–3412, 2001.

[18] S. E. Deering. Multicast routing in internetworks and extended lans. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 55–64. ACM Press, 1988.

[19] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification. http://www.ietf.org/rfc/rfc2362.txt.

[20] P. Francis. Yoid: Extending the internet multicast architecture, April 2000. http://www.isi.edu/div7/yoid/.

[21] A. Ganjam and H. Zhang. Connectivity restrictions in overlay multicast. In *Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, pages 54–59. ACM Press, 2004.

[22] M. T. Goodrich and R. Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*. Wiley, 2002.

[23] M. Handley, S. Floyd, J. Padhye, and J. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. `http://www.ietf.org/rfc/rfc3448.txt`.

[24] V. Hardman, M. A. Sasse, M. Handley, and A. Watson. Reliable audio for use over the Internet. *Proceedings of INET, Oahu, Hawaii*, 1995.

[25] M. Hefeeda, A. Habib, B. Boyan, D. Xu, and B. Bhargava. PROMISE: peer-to-peer media streaming using collectcast. Technical report, August 2003. CS-TR 03-016, Purdue University. Extended version.

[26] M. Hefeeda, A. Habib, D. Xu, B. Bhargava, and B. Botev. Collectcast: A peer-to-peer service for media streaming.

[27] International Telecommunications Union, Recommendation G.114. One-way transmisesion time, February 1996.

[28] S. Jain, R. Mahajain, D. Wetherall, and G. Borriello. Scalable self-organizing overlays. Technical Report 02-02-02, UW-CSE, Feb 2002.

[29] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. pages 197–212.

[30] X. Jiang, Y. Dong, D. Xu, and B. Bhargava. GnuStream: A P2P Media Streaming System Prototype. In *Proceedings of the International Conference on Multimedia and Expo (ICME)*, volume 2, pages 325–328, July 2003.

[31] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *IEEE Infocom 2004*, March 2004.

[32] M. Kwon and S. Fahmy. Topology-aware overlay networks for group communication. In *Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 127–136. ACM Press, 2002.

[33] J. Lennox and H. Schulzrinne. A protocol for reliable decentralized conferencing. In *NOSSDAV '03: Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, pages 72–81. ACM Press, 2003.

[34] J. Liebeherr and T. K. Beam. Hypercast: A protocol for maintaining multicast group members in a logical hypercube topology. In *Proceedings of 1st International Workshop on Networked Group Communication (NGC '99)*, pages 72–89, July 1999.

[35] M. R. Macedonia and D. P. Brutzman. MBone provides audio and video across the internet. *IEEE Computer*, 27(4):30–36, April 1994.

[36] J. Moy. Multicast Extensions to OSPF. `http://www.ietf.org/rfc/rfc1584.txt`.

[37] J. Ott, C. Perkins, and D. Kutscher. A Message Bus for Local Coordination. `http://www.ietf.org/rfc/rfc3259.txt`.

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

[39] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.

[40] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *Proceedings of IEEE INFOCOM'02*, June 2002.

[41] S. Ratnasamy, M. Handley, R. M. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Proceedings of the Third International COST264 Workshop on Networked Group Communication*, pages 14–29. Springer-Verlag, 2001.

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

[43] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, nov 1984.

[44] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. `http://www.ietf.org/rfc/rfc3550.txt`.

[45] K. Sripanidkulchai, A. Ganjam, B. Maggs, and H. Zhang. The feasibility of supporting large-scale live streaming applications with dynamic application end-points. *SIGCOMM Comput. Commun. Rev.*, 34(4):107–120, 2004.

[46] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[47] D. Tran, K. Hua, and T. Do. Zigzag: An efficient peer-to-peer scheme for media streaming, 2003.

[48] D. Waitzman, C. Partridge, and S. Deering. Distance Vector Multicast Routing Protocol. `http://www.ietf.org/rfc/rfc1075.txt`.

[49] W. Wang, D. A. Helder, S. Jamin, and L. Zhang. Overlay optimizations for end-host multicast. In *Proceedings of the International Workshop on Networked Group Communication (NGC)*, October 2002.

[50] D. Xu, M. Hefeeda, S. Hambrusch, and B. Bhargava. On peer-to-peer media streaming. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 363. IEEE Computer Society, April 2002.

[51] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

[52] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, pages 11–20. ACM Press, June 2001.