Quality of Service, End to End Delays and Overlay Multicast for structured P2P networks like Chord

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Andreas Ruettimann
2011

Leiter der Arbeit:
Professor Dr. Torsten Braun
Institut für Informatik und angewandte Mathematik

Acknowledgment

I would like to thank all the people that supported me so much in achieving the goals of this mastherthesis. First of all I would like to thank my tutor Marc Brogle, who has shown a lot of patience and helped me a lot with his creative ideas and his constant support. I also want to thank my research collegues Luca Bettosini and Sebastian Barthlome who generously offered their help where ever they could. Last I would also like ot thank my professor Dr. Prof. Thorsten Braun for his intelligent questions during my presentations that helped me to keep on track and encouraged me to think over some aspects of this thesis.

Abstract

Even though the Internet is organized in a decentralized manner, many systems still rely on the centralized client server model and on unicasting. This is mostly the more practical approach but also the more expensive one considering the resources like servers and in terms of traffic being used. Apart from that, centralized systems are not as reliable and scalable as their counterparts. This is why fully distributed systems like Peer-to-Peer (P2P) networks were invented. These networks do not rely on a central instance, meaning, there is no single point of failure and therefore, the network is much more robust. But still, data is distributed using inefficient unicast connections. To solve this problem, IP Multicast was introduced in IPv4 but due to political and technical reasons, it has never been deployed widely in the Internet. Since P2P networks with equivalent peers arose it is possible to disseminate data among those peers using Overlay Multicast. In order to receive those Overlay Multicast messages, peers need to be structured, mostly in some kind of tree. This has further advantages. Peers can be placed according to their Quality of Service (QoS) properties allowing them for example to have a guaranteed bandwidth to the root.

On the basis of CHORD, which is a structured P2P system supporting Overlay Multicast and QoS, we will show that data can be distributed much more efficiently while supporting QoS classes (e.g. bandwidth) and furthermore that even End-to-End delay requirements can be fulfilled up to a certain degree.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Internet is very useful for companies to link their different sites together and for fast Information dissemination between multiple users. But it can also be used also for real time applications like video on demand, voice over IP, live video streams, online multi player games and live video conferences. These services often require a lot of bandwidth and a stable network that provides low and static latencies. The problem is that upgrading networks in terms of terrestrial cable, routers, mainframes, etc. is very expensive and is also bound to local policies as we will show later on. Our goal in this masterthesis is to find ways to use the current infrastructure more efficiently in order to satisfy the requirements of the previously mentioned applications.

To be able to achieve such an improvement, we do not need to reinvent the wheel from scratch, but we use technologies that already exist and we combine and modify them. One such technology is almost as old as the Internet itself and is called IP-Multicast. This protocol is suported by the routers and enables hosts to subscribe themselves to different multicast groups in order to receive all the information that is sent to this group. The trick is that a sender of this group has to send only one message to its router, which will then duplicate the message if needed and so on until each host has received the message. This is much more efficient than sending the message to each host separately. Since many messages taking the same way would increase the sender's outgoing bandwidth requirements. Such a mechanism would be a huge advantage to have but unfortunately there are political, billing and security concerns that keep providers from supporting IP Multicast [1] on a large scale.

There is also the possibility to do multicast on the application layer though. These so called Application Layer Multicast (ALM) protocols are often used in Peer to Peer (P2P) networks in order to make data distribution more efficient as has been stated by our research group [2] and by the Survey of Application-Layer Multicast Protocols [3]. These networks connect groups of so called peers (hosts) on the application layer. Meaning, the application decides to which other hosts a connection is maintained and it is also responsible to maintain a consistent structure and to distribute information among its peers. This enables implementing an overlay network that forms its structure independent of the underling physical network. This is important since the links between the peers can be built upon constraints (e.g., latency or bandwidth). A major advantage of most P2P networks is that in contrary to the classical client server model, all hosts are considered equal. This is important because servers need a huge amount of bandwidth and

processing power in order to satisfy all requests. This is very expensive and inefficient if the Internet connection of such a server is broken or if the server crashes. Then, nobody will get the information that was stored on this server. Whereas, in a P2P network, there might be another peer that has the same information stored like a failed peer, enabling the network to keep this information even though hosts constantly leave the network.

Additionally, as we mentioned previously, a P2P network often uses ALM. Meaning, instead of letting routers duplicate packets if multiple receivers are connected to it, the Peer to Peer application on each host will decide to which other hosts it wants to send the information to. The receiving host will then do the same until each host of the network received the message. This is not as efficient as IP Multicast, but it can be used all over the Internet and is still much more efficient than a normal client-server approach.

Using P2P networks with ALM enabled, we can already reduce the bandwidth used, increase the robustness of the network and have the freedom to organize our network as we please. The only prerequisite that is missing is a mechanism that allows us to organize our hosts in the P2P network based on their capabilities (e.g., latency, bandwidth, processing power, etc). This is called Quality of Service (QoS) [4] and there exists already several implementations on a router level. These protocols try to prioritize each packet so that for instance a VoIP packet would be favored over a normal http packet. It is important that the packets of a real time application arrive in time in order not to disturb the communication. On the other hand it does not really matter if a website is being received a little bit sooner or later. These protocols also try to find optimal paths, considering capacities of each link in order not to delay for instance a VoIP communication more than needed. This mechanism sounds very promising but unfortunately it is also not spread widely over the Internet because of billing and other concerns. For example, it would be very difficult to compute the exact costs to have a guaranteed bandwidth from, e.g., a host in Bern, Switzerland to a host in NY, USA. This is due to the fact many routers and many providers have to be passed that all need to have contracts with each other and all the routers need to be configured for this specific connection. Also, it is almost impossible to determine in advance how much traffic in a certain time interval will arise on a certain link, which makes it very difficult if not impossible to be able to provide QoS guarantees.

Fortunately, we are free to structure our P2P network as we please, so we try to organize peers in a way that respects their requirements [5]. This is not possible in every P2P network, but it can be enabled by using a separate overly on top of the P2P network that will enable this network with QoS capability. In [6] such a framework that supports almost every P2P network has already been implemented. This framework is able to handle properties like bandwidth or processing power, but it cannot handle Round Trip Time (RTT) to root guarantees or any other QoS property that relies on multiple hops.

Since we will be using ALM in our P2P network, each message will pass several hosts until it is received by all the members of the network. This makes clear that RTT to root guarantees would be a necessity to be able to support real time applications, since the better this path from the root to the receiver is, the more likely it is that a message is received in time. In order to support this each host will need to find a parent that satisfies its RTT requirements and it needs to be able to change that parent if its requirements are not met anymore. This requires that the P2P network to know more about the underlying physical network, making it more complex and though more

error prone which is why we want to implement it as simple as possible. Our goal is to provide a framework that enables P2P networks to support RTT to Root guarantees.

We use a structured P2P network called Chord [7] providing the previously mentioned improvements like ALM and application layer QoS as a basis. Then, we will setup a custom P2P network in an attempt to provide RTT to root guarantees. This network will be implemented in the Omnett++ [8] simulator, which is a public-source, component based discrete simulator, whose main purpose is to simulate communication systems.

This Master Thesis is structured as follows. The next Chapter will discuss the different forms of multicasting, Peer to Peer networks and Quality of Service possibilities. Chapter 3 will then explain the Omnet++ simulator, the modifications we performed and how we set it up for our project. Furthermore it will show how we implemented Chord on this simulator and how we constructed the framework that provides RTT to root guarantees. In Chapter 4, we will then present optimiziations for the Chord application that make Chord a more robust and reliable network which will help us in achieving our goals. Finally, Chapter 5 will discuss our simulation setup and the results of our simulation runs. sd

Chapter 2

Related Work

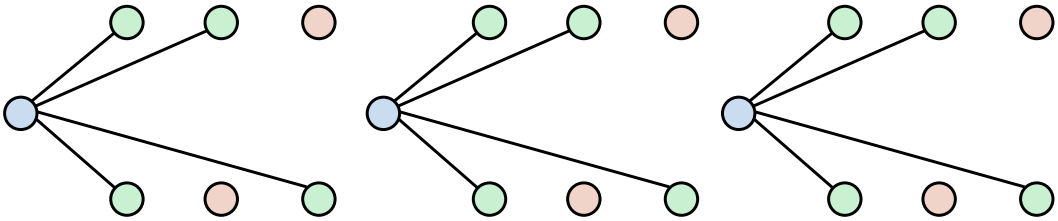
2.1 Introduction

As depicted in the summary, this master thesis is about Overlay Multicast, Quality of Service (QoS) and end-to-end guarantees. This chapter explains how these technologies work and what their advantages are. To be able to use those technologies, two or more computers need to communicate with each other. In today's Internet, this is being achieved by a packet delivery service. Those packets can be addressed to one or multiple receivers, depending on the communication scheme used. There are mainly three such schemes:

- **Unicast** depicted in figure 2.1(a) is used if exactly two network entities want to communicate with each other. Each packet will be addressed by the IP of the receiver in order to get to the designated destination. If more than one network entity wants to receive the information, multiple packets have to be sent.
- **Broadcast** shown in figure 2.1(b) is used if one sender wants to reach all members of a subnetwork. Only one packet, which is addressed to the current subnetwork has to be sent in order to be received by all the network entities.
- **Multicast** illustrated in figure 2.1(c) is used if one sender wants to deliver the same information to multiple receivers simulataneously. Only one packet, addressed to a multicast group has to be sent in order to be received by all members of the designated multicast group.

2.2 Multicasting

Since today's Internet is growing very fast and because of the presence of many different multimedia applications, demanding much more bandwidth, an efficient way to distribute data is required. The obvious tools to do that are multicast protocols for networks, which allow a sender to reach multiple receivers by sending only a single packet. This technique called IP Multicast was introduced many years ago but several problems still constrain a wide deployment which is



(a) Unicast (b) Broadcast (c) Multicast

Figure 2.1: Unicast Broadcast and Multicast Schemes

why other multicast mechanisms called Overlay Multicast were introduced. The following section explains the advantages and disadvantages of the different types of multicasting and where they are being used.

2.2.1 IP Multicast

IP Multicast [1] depicted in Fig. 2.2 is a method to forward a datagram (packet) to several receivers simultaneously. Because it would lead to a huge delay if each packet would be confirmed, this method normally does not provide a reliable service but uses protocols like UDP or RTP for real time streaming applications. Each user that is interested in, e.g., a video stream can join the corresponding multicast group. A multicast group is identified by a special IP address reserved only for multicasting purposes and uses dynamic membership. There is no restriction on how many members a group contains and when they need to join or leave.

As defined by the Internet Assigned Numbers Authority (IANA) [9], the IP addresses ranging from 224.0.0.0 to 239.255.255.255 are reserved for multicast purposes only. These so called D-Class addresses begin with a binary 1110 making them easy to distinguish from unicast addresses. This address space is divided further for different applications. 224.0.0.0/24 is used for local routing only, meaning packets addressed to this space will not be forwarded by routers. 232.0.0.0/8 is reserved for IGMPv3's source specific multicasting. 233.0.0.0/8 is used for IGMPv1 and IGMPv2 and 239.0.0.0/8 is used for administrative purposes only. For a multicastapplication any ip inbetween 225.x.x.x to 232.x.x.x and 234.x.x.x to 238.x.x.x can be used. The remaining addresses are reserved by IANA.

For IP Multicast to work, special routers have to be used. If a router receives a multicast message it has to know to which interfaces it has to forward the message to if the Time To Live (TTL) [10] has not yet run out. Switches do not have to be multicast aware. Each multicast packet is sent to all its ports and the hosts will then decide by themselves if they are interested in the packet or not.

To join a multicast group, a host has to send a join message to the address of this group. Henceforward, the first router to get this request will forward multicast packets sent to this group to the interface the joining request came from. The router will then ask the next router to forward messages addressed to this group, and so on. Using the Internet Group Management Protocol (IGMPv1) [1], the hosts need to periodically send such join messages as long as they want to

receive data from this group. The router will automatically stop forwarding messages to hosts, which did not send this message for a certain amount of time. With IGMPv2 [11], hosts can also send leave messages, thereby reducing the communication traffic needed. In IGMPv3 [12], source specific multicasting was introduced allowing receivers to choose a specific source, from which they want to receive their multicast messages.

IP Multicast is implemented in most of todays routers but still it is not widely available for end users. This is because of security concerns and billing issues. It would be very difficult for providers to find out which packet sent by a user exactly generates how much traffic.

This makes it very unlikely that IP Multicast will be deployed in the Internet any time soon. Fortunately, it is also possible to do multicasting on a the application layer. These Application Layer Multicast (ALM) systems provide an overlay network that facilitates distribution of multicast data using unicast connections through the Internet.

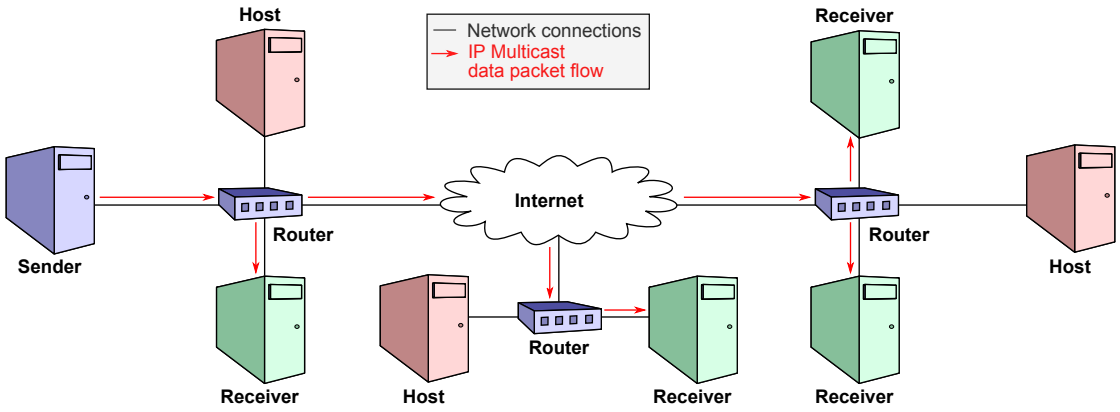


Figure 2.2: IP Multicast Example Setup

2.2.2 Overlay Multicast

As we have seen in Section 2.2.1, IP multicast is a very good way to transmit stream based data much more efficiently over a network. But this has not been used widely due to security and billing concerns. Therefore the Application Layer Multicast(ALM) also called Overlay Multicast was introduced. Protocols using this paradigm provide a transparent overlay over the Internet which allows computers to use multicasting on the application layer. This is mostly done using Peer-To-Peer (P2P) protocols since they already provide an overlay network.

An overlay network [3] provides virtual interconnections of hosts on the application layer, meaning an application running on each host maintains all the connections to other hosts in that overlay network. Depending on the P2P protocol used, one or more multicast group can be active in an overlay network. In a scenario with only one multicast group per overlay network, a host that would like to receive multicast messages from a certain group, just joins the corresponding P2P network and henceforward receives all multicast traffic sent to this group. Because the joined host is also part of the P2P network, it has to forward incoming multicast messages to some other hosts as well. Each host only has to forward the multicast message to several other hosts

in order for all members of the P2P network to receive multicast messages. Depending on the multicasting strategy used, this distribution is more or less efficient, but in any case it will never be as efficient as native IP multicast. This is due to the fact that packets are not duplicated by routers but by end systems. This means that all pathlengths are increased and also the same paths might be used several times. But stills ALM provides a much more efficient distribution of information than using client/server distribution machanisms, where the original Sender of a message has to serve each recepient individually.

This brings us one step closer to our goal of providing a distribution mechanism that is up for today's challenges like transmitting video on demand and voice over IP without having to massively upgrade the currently available Internet infrastructure. There are two different ways how Overlay Multicasting can be done: sender- and receiver driven multicast

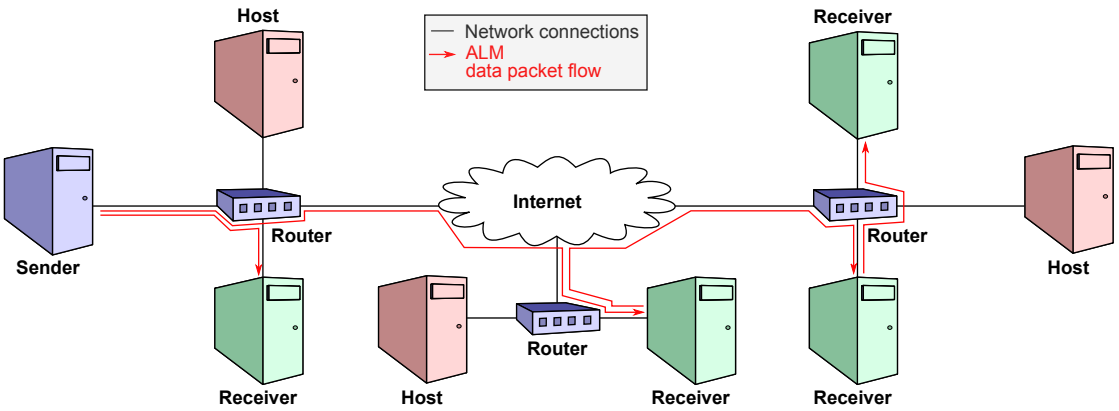


Figure 2.3: Application Layer Multicast Example Setup

2.2.3 Sender Driven Multicast

Using sender driven multicast, a host that receives a multicast message from another host (its parent) can decide by it self to which other hosts it wants to forward the message to. This means that a host cannot estimate what its next parent will be, and thereby is not having control over the message's distribution properties, such as end-to-end delay or path length. But instead, it can regulate the number of other hosts to which it will forward the message to, influence an even distribution of the multicast tree. Hence the multicast algorithms complexity can be reduced to a minimum.

Sender driven multicast is often used in networks that are not depending on delays or where a simple network structure without a huge synchronizing overhead is preferred.

2.2.4 Receiver Driven Multicast

Using receiver driven multicast a host can search its multicast parent (the host from which it receives its multicast messages from) by it self. This has the advantage that the multicast tree can be structured using a certain property for example end-to-end delay or bandwidth requirements.

A joining host will get a position in the tree that guarantees for example a certain maximum Round Trip Time (RTT) to the multicast root or it will have to face the fact that its requirements cannot be fulfilled at the moment. This makes the overlay network much more complex, since it has to measure continously if the properties are met and if a host has to react on changed parameters. For example if a host's maximum RTT to the multicast root is exceeded it has to search for a new parent, inevitably leading to a change of RTT to the multicast root. These children then have to look for a new parent as well, depending on the implementation.

Receiver driven multicast is normally used in networks that are proximity aware or in networks that use other properties like bandwidth to form their structure. This is because the freedom of a host to choose where it wants to get its multicast messages from (depending on one or ceveral constraints) requires the overlay network either to know where it has to route the new hosts join request to or it would take a very long time for the host to find a parent that satisfies its needs. This knowledge of the overlay network is expensive though since it requires a greater complexity and produces more communication traffic.

2.3 Peer-to-Peer (P2P) Networks

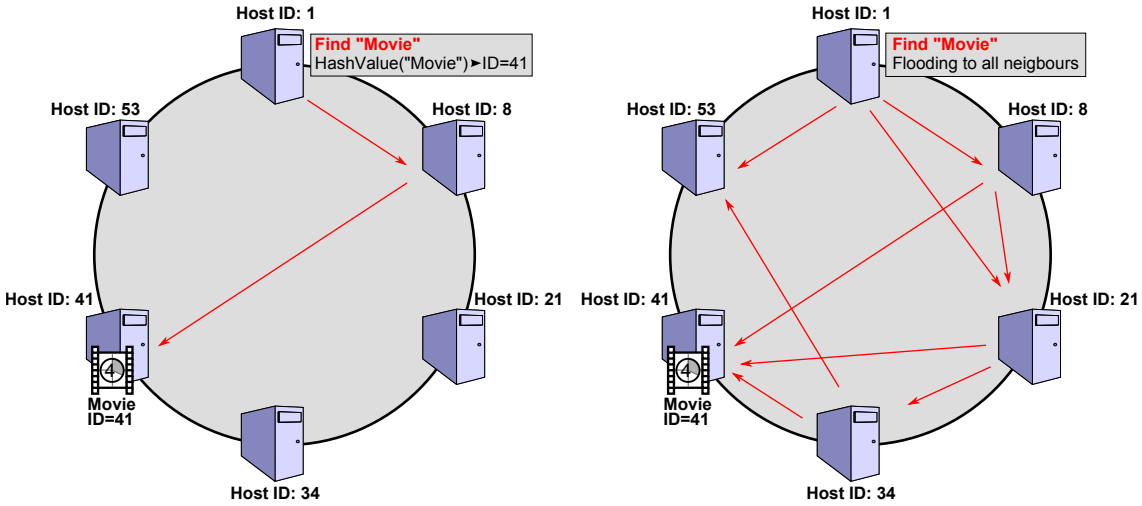
2.3.1 Introduction

As we have seen in Section 2.2, native IP Multicast is not yet widely deployed due to several reasons. But, it is still possible to use a little less efficient multicasting mechanism with P2P networks. These networks built on top of the network infrastructure are in contrast to the classical client/server model and are often organized in a decentralized manner with hosts having equally the role of clients and servers. There are many different P2P networks available depending on the task at hand.

For example, one P2P network could provide a DNS lookup- or a file sharing service, distribute data to many different mirrors, host a voice over IP (Voip) conference, stream a movie and host many other applications.

P2P networks are split into two subcategories: in structured P2P networks presented in Fig. 2.4(a), peers are connected by a previously defined hierarchy and data is stored at specific locations in this hierarchy, whereas data in unstructured P2P networks shown in Fig. 2.4(b) is placed at random positions in the P2P network. This makes searching in unstructured P2P networks much more difficult, since there is no structure that enables us to direct a search query to the host having the designated data. Instead, a flooding mechanism has to be used.

This is also the reason why unstructured P2P networks do not scale so well due to the huge message when trying to find a single file. But, this also makes it much easier to implement unstructured P2P networks since there is no structure that has to be maintained. They also support the search for keywords which would be very difficult to implement in a structured P2P network because of the use of hashes to find data as explained in the Section 2.3.2



(a) Structured P2P Network

(b) Unstructured P2P Network

Figure 2.4: Structured vs Unstructured P2P Networks

2.3.2 Distributed Hash Table (DHT)

As mentioned in Section 2.3, P2P networks are able to locate data at a specific peer instead of flooding a search query to every peer they know by using a so called Distributed Hash Table (DHT) [13]. This is being done by assigning a hash value to each peer and to each data Object in the network. Given a large enough identifier space, each joining host gets a host ID by building a Hash Value from one of its unique identifiers for example its IP address. This host ID will affect the peers position in the network and also for which keys it is responsible for. Keys are generated by building a Hash Value over a unique identifier of a data object (e.g. a file) for example its name. This key (Data ID) will then be placed at a peer having the same or next higher host ID.

In order to find a specific data object a searching peer first has to build the data object's Hash Value and then start a query to find the designated key. This requires a Hash Table to be stored on each peer, mapping all the stored keys to peers possessing the data belonging to this identifier. In addition each peer maintains a routing table to be able to forward a search request to a peer that knows where the data is being stored. How these tables are being generated depends on the P2P network being used as we will see below.

Of course it is possible that two different peers or data objects get the same ID because the identifier space might be smaller than the original file name or the peers identifier. But this case is mostly avoided by a large enough identifier space and a hash function that distributes the hash values very well. Also most P2P networks can detect a duplicate ID.

To simplify our formulation a host with a Chord ID x will be referred to as host x if we refer to the host's IP address we will mention it as such and we will call the Chord Overlay network simply Chord network from here on.

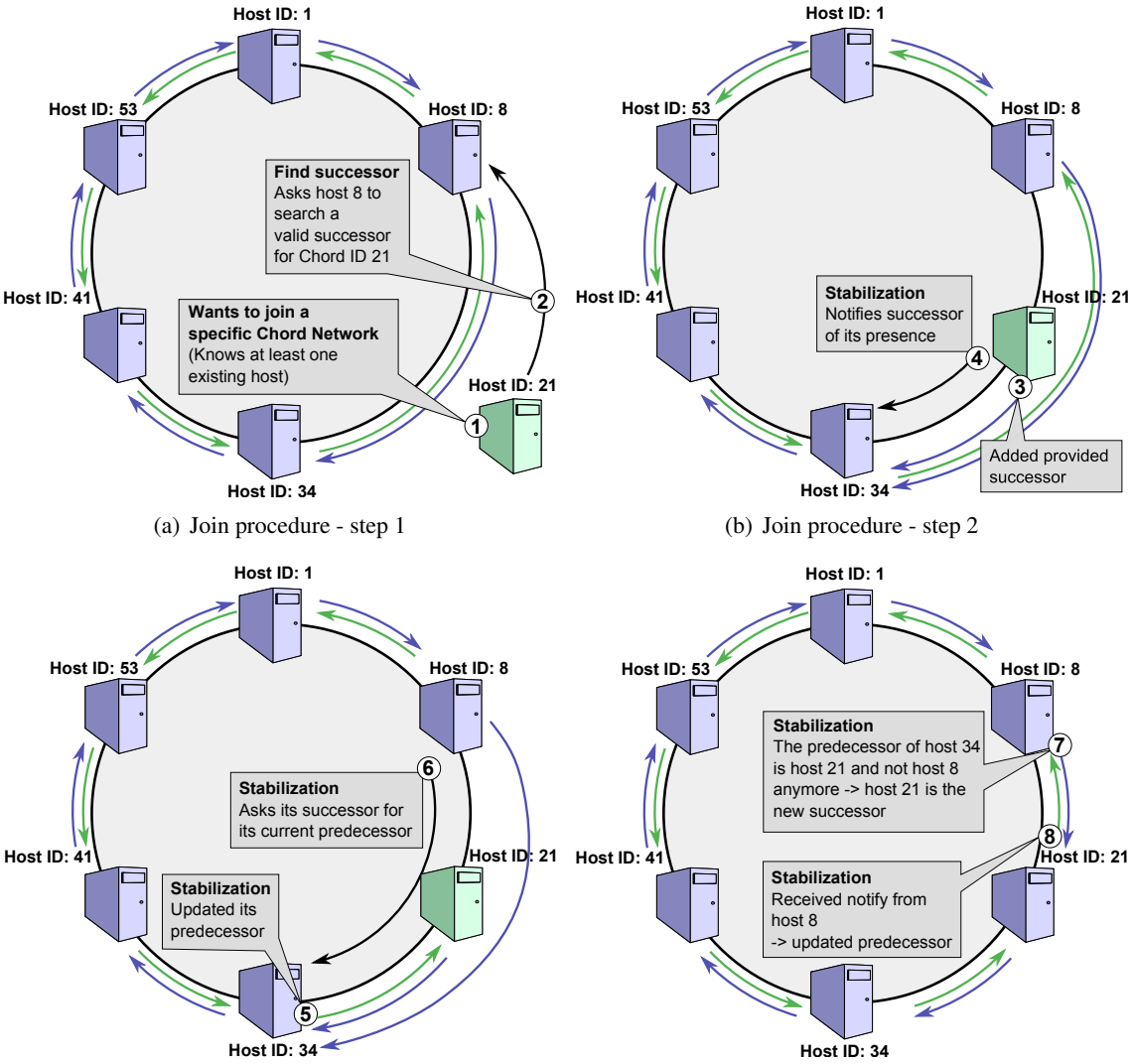
2.3.3 Chord P2P Network

Chord [7] is a DHT based structured P2P network organized using an identifier (ID) ring with IDs clockwise increasing from 0 to $2^{(m-1)}$ (m defines how many bits an ID has). Each peer is given an ID using a hash function such as SHA-1 [14], which determines its position in the ring. The structure of the ring is being maintained by the following connections, also called pointers, which will be stored on each host:

- **Successor:** Pointer to the next succeeding peer, which is the peer with the next higher Chord ID (saves the IP address and the Chord ID of that host);
- **Predecessor:** Pointer to the last preceding peer, which is the peer with the next lower Chord ID (saves the IP address and the Chord ID of that peer);
- **Fingers:** Pointer to a peer being further away (saves the IP address and the Chord ID of that peer) which will be explained later on.

In the following, the basic Chord functionality will be explained:

- **Creating a Chord Overlay Network**
  First a Chord network has to be created. To do that, a host (host 1) adds itself as a successor and leaves its predecessor blank. Then, it has to publish this information to a website or any other central storage in order for other hosts to find it.
- **Joining the Chord Overlay Network**
  As we can see in Fig. 2.5, the Chord network already has 5 members. Now, a new host wants to join:
  1. It has to find the IP address of at least one member (in this example host 8) of the Chord network it would like to join, which is done using the previously mentioned central storage.
  2. By using a hash funtion as explained previously, it gets its Chord ID. Using this ID, host 21 can request its first successor from host 8.
  3. Host 8 will then start a search for a host that satisfies the following condition: $host's\ id < joining\ host's\ ID < hosts\ successor's\ ID$, which will be the host with Chord ID 21. As we will see later, it does not have to answer that search itself but can redirect it to the closest host that does not have a Chord ID higher than the joining host. Anyway, whichever host fulfills the condition returns its ID to the joining host.
  4. After the joining host receives the Chord ID and IP address of its successor, it immediately saves this tuple in its successor list. From now on, host 21 is considered joined even though no other host got to know it yet. But, this is what another main mechanism of Chord is responsible for: the stabilization cycle. As the name suggests, this is a periodically running procedure that makes sure that all pointers in the network are up to date.



(a) Join procedure - step 1

(b) Join procedure - step 2

(c) Join procedure - step 3

(d) Join procedure - step 4

Figure 2.5: Joining procedure

5. The stabilization protocol will first notify the new successor of its presence, because as shown in Fig. 2.5(b), host 34 still has host 1 as its predecessor instead of host 21.
6. After host 34 gets this notification, it will see that its predecessor has changed and will update it immediately.
7. A little bit later, the stabilization protocol of host 8 will start checking its pointers. To do this, it starts by asking its successor (host 34) for its predecessor.
8. As host 34 answered the request, host 8 does see that it is not the predecessor of host 34 anymore, but also that host 21 is now. So, it just adds host 21 to its successor list and notifies it that it may have a new predecessor.

9. Host 21 receives the notification and adds host 8 to its predecessor list.

After this procedure finishes, the network is in a consistant state again and host 21 is considered fully joined. As one can imagine, there is a lot of things that can go wrong during this procedure (failing hosts, too many simultaneous joins / leaves, etc.). But a lot of these issues will be taken care of by the stabilization protocol and Chapter 4 offers many improvements to that procedure.

- **Leave Procedure** If a peer wants to leave a Chord network, it has to notify its neighbors of their new successor/predecessor in order to maintain a consistent network. It is possible that the Chord ring gets broken because of too many hosts leaving simultaneously. This problem can be minimized by allocating more successors. The number of successors then equals the number of consecutive hosts minus one that can fail simultaneously without breaking the Chord ring.

As already mentioned in Section 2.3.2, data in a structured DHT based P2P network data can be found using their hash value (key). If a peer wants to add new data to the P2P network, it first builds a hash value over for example the data's name resulting in a key and then sends a message to the peer with the closest succeeding host ID of this key. The destination peer will then insert the hash value of the data and the IP address of the host possessing this data to its hash table. From now on, each peer that wants to receive this data can route a request to the data's hash value in order to find out where it is stored.

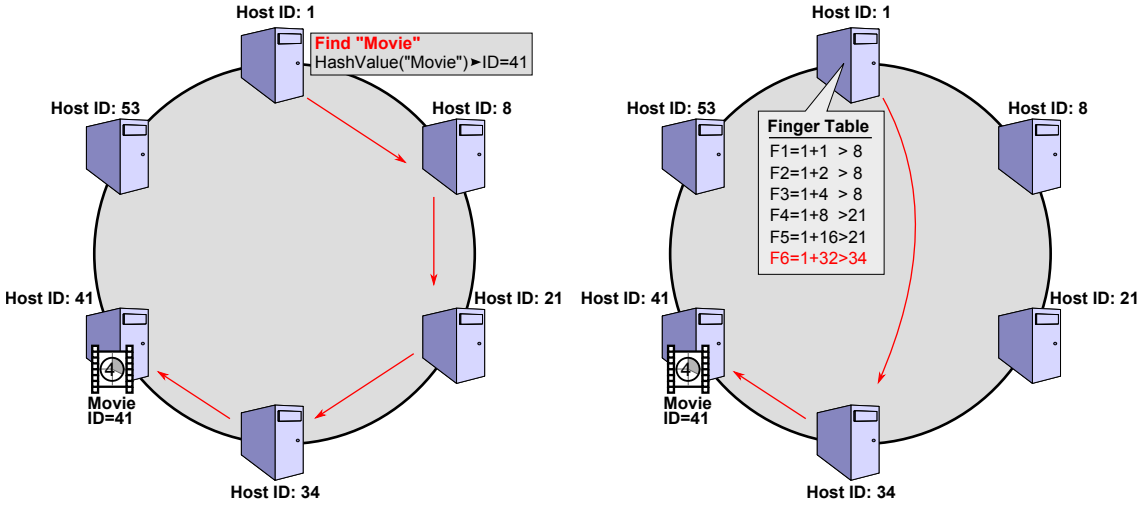
As one can imagine, this network design is not very efficient since a host, which wants to route a message to a preceding host that is not in its successor list would have to route the message from one host to the another throughout the whole network.

An example is shown in Fig. 2.6(a). The host with Chord ID 1 wants to find a file with a key (hash value) of 41. Because it only knows its successor (the peer with host ID = 8) it has to forward the request to the only host it knows: its successor. The successor will then forward the request to its successor and so on until a host is reached, which has a Chord ID greater or equal than the requested file ID. This host will then return the address of the host with the designated data to the source of the request. What one should keep in mind is that Chord is a ring, meaning that the successor of the peer with the highest ID is the peer with lowest ID. This is specifically important for the implementation.

As we have seen, basic Chord routing is not very efficient which is why the concept of finger tables was introduced. Such a finger table holds $log_2(m)$ pointers to hosts being 2^i (m=indentifier space size, i=index from $log_2(m)$) IDs away. To build the finger table, a host looks for another host with a Chord ID greater or equal than its own Chord ID + 2^i. This means that several fingers can point to the same host since normally there are many free ID's between two existing hosts. Using this finger table the routing complexity is reduced from $O(n)$ to $O(log_2(n))$ making the network very scalable.

Figure. 2.6(b) shows how this works. Again, the host 1 wants to find a file with key 41. This ID cannot be found in its successor list or in its finger table, so it forwards the request to the closest host from its finger table, which is in this case the host with Chord ID 34. This host knows that its successor (host 41) must be responsible for for that key 41 because there is no other host

in between. As we will see in Chapter 3 it is also possible to do Quality of Service (QoS) and receiver/sender driven multicast using chord.



(a) Basic Chord (b) Chord with Finger table

Figure 2.6: Lookup without and with finger table

2.3.4 NICE

In Section 3.3, we will use NICE [15] as an example for a proximity aware, structured P2P network that could support RTT to root guarantees with minimal modifications. Hence, we first have to explain how NICE works. In contrast to Chord, NICE is much more structured as shown in Fig. 2.7. It is organized in different layers bottom up from layer 0 to layer n. Those layers in turn are organized into clusters. Each cluster has a cluster leader, which is being determined by building the graph-theoretic center taking phyisical proximity into account. With other words, the cluster leader is the host having the lowest maximum sum of the RTTs to all other cluster members. It is also the parent for each cluster member serving them with multicast messages.

Each cluster has to hold at least k and a maximum of 3k-1 hosts. This prevents having too small clusters, which would lead to a very complex and inefficent structure, since the hop count and the number of clusters to be maintained would increase. On the other hand, too large clusters would lead to a very high Fan Out of the root or other cluster leaders.

Furthermore, the number of layers determines the maximum Hop Count (leaf hosts) and also the Fan Out (root, cluster leaders). There are no specifications though how many layers are optimal. This is application dependent and can either be set as a static parameter or could also be implemented dynamically (e.g., if the numbers of clusters on a layer exceeds a certain value).

The layers are interconnected with each other via the respective cluster leaders that reside on different layers at once. NICE needs to obey the following rules in order to function properly:

- A host belongs to only a single cluster on each layer where it is present.

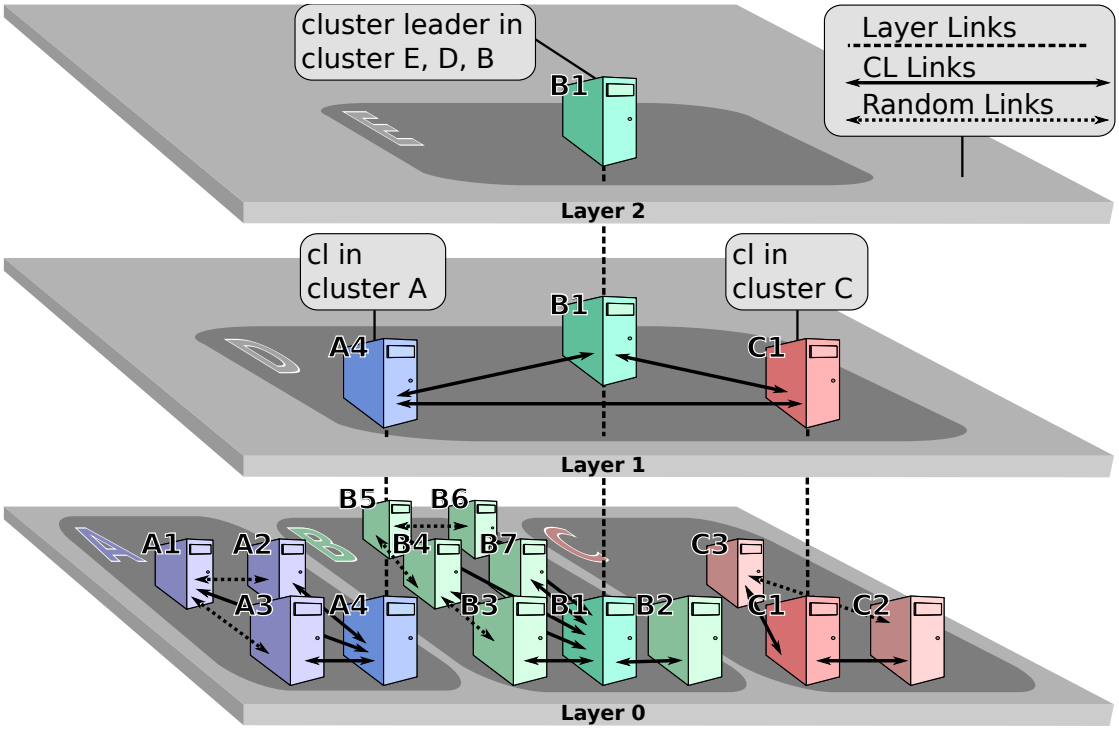


Figure 2.7: NICE structure

- If a host is in layer L, it is also located in layers $L-1, ..., 0$.
- A host, which is not present in layer L, cannot be present in any higher layer $(L+i, i \geq 1)$.
- The cluster size is between k and $3k-1$, where k is a constant with $k > 0$.
- There are at most $log_k\ N$ layers, the highest layer only contains one host (root host).

This should already describe how NICE is basically functioning. A joining host will look for the cluster leader to which it has the lowest RTT to in order to ask it to join its cluster. If the cluster is already full, it has to be split in order for the new host to join one of the newly created clusters. Vice versa, if a host leaves and the cluster is getting too small, clusters have to be merged.

These two examples already show clearly that it is not very easy to maintain such a complex structure imposed by so many constraints. Additionally to the splitting and merging of clusters, the cluster leaders have always to be aware of their cluster mate's status and their respective RTTs. If the RTTs change, a new cluster leader may need to be determined. In order to do that, each host periodically sends out so called heart beat messages which enables exchanging their status and measure their RTTs to each other. This is problematic, because it causes a lot of communication traffic and can also lead to synchronization problems.

Another issue is that especially the root and also hosts that are in the upper layers can have a very high Fan Out because they are responsible to distrubute multicast messages to all their

cluster members implying multiple clusters on multiple layers. This problem can be omitted by a substructure in each cluster, where the cluster leader delegates the multicast distribution in the cluster to another host, sparing it the distribution to all the clustermates. This though makes NICE even more complex.

But, besides those disadvantages, NICE is a very good, proximity aware, scalable and structured P2P network that could support RTT to root guarantees and also reduces the avarage RTT to root very well.

For a sample NICE implementation using Omnet++ you can have a look at the bachelor thesis of Sebastian Barthlomé [16], which is a member of our research group.

2.4 Overlay Multicast Quolity of Service (OmQoS)

So far, we have seen that it is possible to distribute data much more efficiently using ALM but also that at the moment, we cannot provide any guarantees for bandwidth, jitter, delay or other so called Quality of Service (QoS) properties. But, such a feature would be very appreciated because the Internet is growing fast, hence also multimedia applications, which often demand near constant jitter and low delays in order to function properly.

One of the most difficult problems is that the importance of these different properties can change depending on the application. For example, if a P2P network hosts a video conference, it is vital that it provides low latencies and a low jitter because otherwise the participants would have trouble understanding each other. Contrary to a conference, a mirroring service is not depending on jitter or delay but on bandwidth. If not enough bandwidth is provided by the P2P network, some peers may not receive all data resulting in an inconsistent state between mirrors.

Because these parameters are very application dependent, we decided to combine different QoS properties into one so called QoS class citebrogle:qos and weighting them according to their P2P importance. Each peer in the network uses such a QoS class, which affects its position in the network. This QoS class could either be computed using network measurements or could be set using parameters agreed between clients and network service provider (ISP).

The technology to provide QoS directly on a TCP/IP level exists already. But, to realize a service on a global scale would be almost impossible because it would require more control over todays networks, more sophisticated routers, new billing mechanisms, agreements between providers, and so on.

Considering this situation, it is not very likely that this service will be available for a Internet wide community anytime soon. But, what we can do is using P2P networks to provide application layer based QoS support **??**. We have to be careful though. Because P2P system also depend on the underlaying Internet, which as mentioned above does not provide QoS, we can only try to optimize QoS support but we will never be able to provide 100% guarantees.

In order to support QoS, the above mentioned QoS classes must fulfill the following properties:

1. A total order relation for all QoS classes must exist.
2. The different parameters of the QoS classes must be independent of link length and number of hops in the network.

3. There is only a finite number of QoS classes.

These restrictions facilitate QoS support for all peers in the P2P network, because we are able to arrange them according to their QoS class. Additionally, we need a multicast distribution tree with monotonically decreasing QoS classes from the root host to all leaf hosts in order to support QoS. Furthermore the multicast tree has to fulfill several properties too;

1. The host with the highest QoS class is the root of the multicast tree
2. A child has to have a smaller or equal QoS class than its parent host

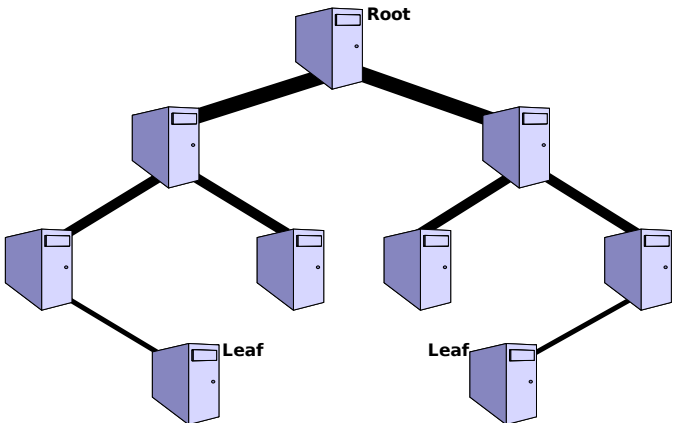


Figure 2.8: Multicast tree with monotonically decreasing QoS classes

Fig. 2.8 shows a Multicast tree that meets the above defined requirements. The thickness of the edges equals to, e.g., the maximum bandwidth between two peers. We can see that all paths from the root host to all leaf hosts have monotonically decreasing bandwith because peers at the bottom of the multicast tree do not have as much bandwidth to their parents as peers being closer to the root host.

Later we will also show that the end-to-end delay between a leaf host and the root host can be used as an additional QoS parameter. To support this we will implement an additional overlay to the P2P network according to the same rules, in order to provide a framework that supports QoS classes and additionally supports RTT to Root guarantees in many of todays structured P2P networks.

2.5 Summary

In this Chapter, we presented the background theory and related work to understand the contribution of this thesis.

We showed the different transimission mechanisms: unicast, broadcast and multicast.

We presented the IP Multicast protocol and the concept of Application Layer Multicast (ALM) to overcom the problem of limited avialability of IP Multicast in today's Internet.

We showed P2P Overlay networks which facilitate the deployment of ALM. Furthermore, we presented the Chord P2P network, which is the target of the contribution of this thesis. Finally, we presented how Quality of Service (QoS) for ALM could enhance the current's users experience and showed how QoS supporting multicast trees have to look like.

In Chapter 3, we will present the implementation of Chord in a network simulater called Omnet++. Furthermore we will discuss two different multicast approaches for Chord and how to provide end-to-end delay guarantees.

Chapter 3

Implementing Quality of Service and Application Layer Multicast in structured P2P networks using Omnet++ simulator

3.1 Basic OMNet++ Implementation

Before we describe the Chord implementation, we first need to determine the enviroment, in which we want to implement and evaluate it.

We decided to use Omnet++ [8], an open source, component based, discrete network simulator, running on C++.

Omnet++ will be explained in the next section. Then we will explain a Filter Chain concept that facilitates our network model and the framework based implementations. Finally, we will have a look at the distance Matrices we used to build our network.

3.1.1 Network Design in Omnet++

There are two things we have to consider when implementing a P2P network protocol with the Omnet++ simulator: first, we have to define a topology of the network we want to simulate, and second, we need to tell the hosts in this topology what they actually have to do.

Topologies are defined in a XML style formated .ned file, which will be read by Omnet++ to construct the network.

Listing 3.1 defines a simple ChordNode (node is the terminology for a host in Omnet++). It just has an in and an out gate because we only want to simulate a single network. At the moment, this host does nothing since it has no connections to other hosts, no knowledge of what it has to do if a message arrives and no network topology has been built yet.

Listing 3.2 shows how a network topology is being built. First, we give the network a name by defining a module called ChordNetwork. Then, we have to insert submodules, which are in this case all instances of ChordNode, meaning both of the defined hosts have an in- and an out- gate. In a second step, these gates are being connected to each other with a symmetric delay of 100 ms. The last line at tells Omnet++ to construct a network called hord of the type ChordNetwork.

Listing 3.1: ChordNode.ned defining a hosts in and out gates

```
simple ChordNode
    gates:
        in: in;
        out: out;
endsimple
```

Listing 3.2: ChordNode.ned defining a network

```
module ChordNetwork
    submodules:
        node[0]: ChordNode;
        node[1]: ChordNode;
    connections:
        node[0].out --> delay 100ms --> node[1].in;
        node[1].in <-- delay 100ms <-- node[0].out;
endmodule
network chord : ChordNetwork
```

Now, we can define what a ChordNode actually does by creating a ChordNode.cc file containing the ChordNode class, which inherits from an Omnet++ class called cSimpleModule. There are four functions inherited from cSimpleModule that have to be overwritten.

- **void initialize()**
  This method is called after the module is created.
- **void handleMessage(cMessage *msg)**
  This method is called by the simulation kernel when the module receives a message.
- **void activity()**
  This method is in contrast to handleMessage(cMessage *msg) always listening for new incoming traffic. Because it is very memory consuming if every host in the networks is listening all the time, it is discouraged to use this method. HandleMessage is used instead, since it is only called if a new message arrives and does not require a thread to be running all the time.
- **void finish()**
  This method is called when the simulation terminates correctly and is often used to record statistics, which were collected during the simulation

After defining this class, a ChordNode is able to react on incoming messages and to send messages as it pleases.

Listing 3.3: ChordMessages.msg: base message from which all other messages inherit

```
message OverlayMessage
{   fields:
        unsigned int source;
        unsigned int destination;
}
```

A message in Omnet++ corresponds to to a packet, or more accuratly, to a stream of packets in a real network, because its datavolume is not limited. A message is also defined by a class, inheriting from the Omnet++ class cMessage. Omnet++ presents a more practical way though. As shown in Listing 3.3, we can define the messages name (its class) and all the fileds it holds in a .msg file, that when compiled will be transformed to a proper class file by Omnet++, providing getter and setter methods for the different fields.

As of this moment, a message does not have a source nor a destination because it is just sent over the out gate of the host and will then be received by the in gate of the receiver that is connected to this out gate. But since we want to be able to send a message from any host to another using a destinations address, we defined an OverlayMessage, from which all other custom Chord messages inherit, adding two additional parameters: source and destination as presented in Listing 3.3. Now, we have messages that know who sent them and where they need to go, but the network still does not have a clue what to do with them because there is no routing mechanism defined yet. To solve this, we introduced a central instance called distanet, which is connected to each host via their in and out gates. Each message will first be sent to the distanet, which will read the destination address delay it according to an RTT matrix and then forward the message to the receiver. More about the distanet and RTT matrices will follow in Section 3.1.3.

Furthermore, we can define special parameters for all hosts using the omnet.ini file. They will be passed on to the distanet, and from there to the hosts as they join the network. This enables us telling a host how long it is supposed to be in the network or in which mode it should be operating and so on. Most of these parameters can be used to control the networks behaviour and to adapt it to the needs of the different applications. In Chapter 5, we will see the results of some of the different parameters.

3.1.2 Filter Chain Concept

We extended the above described Omnet++ functionality with a chain of filters, each of them having a different purpose. For example, a ping filter would add a time stamp to the message on the sender side and answer with a pong containing this time stamp on the receiver side.

As shown in Fig. 3.1, if a message arrives at a host, it will be processed by the first filter and then be passed on to the next filter and so on until the last filter has been passed. After that, the message can be deleted because it will not be read by any instance of the host anymore.

An outgoing message will do the same as an incoming message but vice versa. It will be created

on the host, processed by all the filters and then be passed to the network.

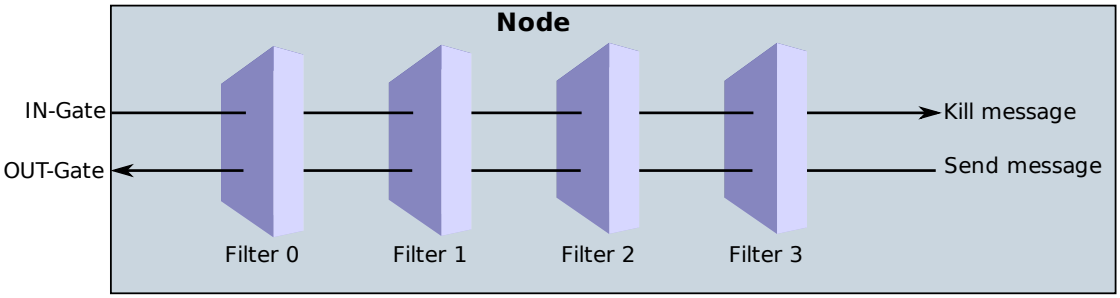


Figure 3.1: Extended modularity by encapsulation into filters

This approach helps us implementing a framework, which will be able to read all the messages coming from / going to other filters. Whether the preceding filter is responsible for a Chord network or any other network does not matter, as long as the framework understands the type of an incoming message. This requires the P2P overlay's messages to inherit from a parent message, which will be defined in the framework. Another advantage is that the code will be better encapsulated because we have different files for different filters and we can define general functions that can be used by other filter based implementations as well.

3.1.3 Distance Matrices

Because our goal is to simulate networks with up to 2000 peers, it would be very cumbersome to define each connection manually as we have seen in Section 3.1.1. Therefore we used a different approach. Instead of building a network containing routers, switches and hosts manually, we use a distance matrix that contains the delays between pairs of hosts. This matrix can be generated using a network topology generator like Brite [17].

The following configuration has been used for Brite:

```
BriteConfig

BeginModel
        Name =  1               #Router Waxman = 1, AS Waxman = 3
        N = 2000                #Number of nodes in graph
        HS = 5000               #Size of main plane (number of squares)
        LS = 5000               #Size of inner planes (number of squares)
        NodePlacement = 1       #Random = 1, Heavy Tailed = 2
        GrowthType = 1          #Incremental = 1, All = 2
        m = 2                   #Number of neighboring node each new node connects to.
        alpha = 0.15            #Waxman Parameter
        beta = 0.2              #Waxman Parameter
        BWDist = 1              #Constant = 1, Uniform =2, HeavyTailed = 3, Exponential =4
        BWMin = 10.0
        BWMax = 1024.0
EndModel

BeginOutput                     #**Atleast one of these options should have value 1**
        BRITE = 1               #0 = Do not save as BRITE, 1 = save as BRITE.
        OTTER = 0               #0 = Do not visualize with Otter, 1 = Visualize
        DML = 0
        NS = 0
        Javasim = 0
EndOutput
```

We generated 13 such matrices, each one of them containing $2000 * 2000$ values, which correspond to the delay between two hosts respectively. These delays are not just randomly generated but computed by the shortest path between two hosts throughout the network, which was generated by Brite. To be able to use such a topology in Omnet++ we had to change its behavior. Instead of connecting hosts with each othe,r each host is connected to a central instance called distanet as shown in Fig. 3.2.

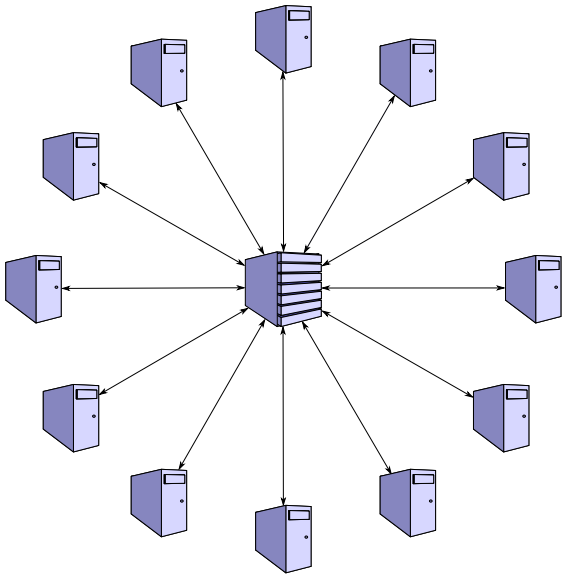


Figure 3.2: Distanet

The distanet generates hosts as described in Listing 3.2 and connects them using delays from a generated distance matrix. Table 3.1 shows the minimum, maximum and average values over all the 13 distance matrices we used for our simulations.

3.2 Chord Implementation

In Section 2.3, we summarized how Chord works in principle. This chapter will now explain in more detail how we implemented Chord and what kind of problems arose related to our special applications.

Furthermore, in Section 3.1 we have seen that we can use the omnet.ini file to define additional parameters and that those parameters are being passed to the distanet during the network initialization. Some of these parameters, like how many peers the network contains and how fast the hosts should be joining, are used only by the distanet since it is responsible for setting up the network as we will see in Section 3.2.2. Other parameters, like how long a peer stays in the network or in which multicast mode it is supposed to operate, are then passed on to each host by a host initialization message, which tells the host to join the network.

Table 3.1: Distance Matrices Characteristics

Matrix	min RTT (ms)	mean RTT (ms)	max RTT (ms)
Matrix 0	0.08	22.47	48.44
Matrix 1	0.09	30.35	90.48
Matrix 2	0.05	30.56	94.58
Matrix 3	0.05	29.76	90.23
Matrix 4	0.07	23.26	57.52
Matrix 5	0.09	22.78	51.82
Matrix 6	0.04	22.77	49.24
Matrix 7	0.08	23.27	52.30
Matrix 8	0.05	22.91	53.92
Matrix 9	0.05	23.27	50.83
Matrix 10	0.08	22.47	48.44
Matrix 11	0.05	22.91	54.00
Matrix 12	0.01	23.13	54.17

3.2.1 Bootstrapping in Chord

In order for a host to join a Chord network it has to perform a process called bootstrapping. This process involves initializing the host with its Chord ID and the search for an existing peer, which can provide the necessary information. Either the joining host already knows an existing peer, or this information has to be provided by another instance. Such a source could be a website containing IP's of existing peers or it could be provided by a fully distributed list that for example uses DNS fallback if the default host is not reachable. To simplify this process, we decided to use the former. A special host called bootstrap host, which will be the first active host (host with IP 0) will be responsible to provide a list of active peers. We implemented the bootstrap functionality in a separate filter because it is not a member of the Chord network and additionaly with this modular approach we could also support multiple bootstrap hosts. Of course, we could also use existing peers, but since they can join and leave as they please, it is better to use hosts like servers that are permentantly online.

The bootstrap process works as follows. As we have see in Section 2.3.3 and will be described in more detail in Section 3.2.2 a joining host is first given a host ID corresponding to its IP address and a Chord ID, which will be randomly generated. After this is done, the host sends a ChordBootstrapQuery message to the host with id 0. This host will return a list of potential peers using a chordBootstrapResponse message, depending on the chord ID provided. It is not necessary to use the chord ID in the selection process but it helps joining faster later on. If the bootstrap host does not know any other active hosts and no request is pending, it just returns a blank list. This is necessary because the first host that would like to join the network will have to create it because it is not yet existing.

After receiving a list of existing peers from the bootstrap host, the newly initialized host will be able to join the network by asking one of those peers for its successor. After having received its successor as we will see in more detail in Section 3.2.2, it informs the bootstrap host of its new

status. This will lead to an insertion of the just joined host to the bootstrap host's active peers list and the removal of the oldest peer. This ensures that the bootstrap host always has an updated list and also that this list does not get too large.

Contrary to a joining peer, a leaving peer will invoke its removal from this list and additionally provide its neighbours for the bootstrap host to use as a replacement.

Theoretically, the bootstrap host could run out of active peers if all its known peers fail simoultaniously. In such a case, a new network would be created. To prevent this, we did not allow ungraceful leaves for the moment. We could also notify the bootstrap host periodically in a large time interval or use several bootstrap hosts to provide more robustness.

3.2.2 Initialization and Joining Process

As we have seen previuosly, the distanet is responsible to set up the network. To do that, it sends a host initialization message to each host containing several initialization parameters. These messages will be sent in a time interval of 0.5s as defined by one of the omnet.ini parameters. Each host getting this message will know its host IP, how long it needs to stay in the network and many other properties as we will show later.

After receiving the initialization message, a joining host (h1) will assign itself a randomly generated Chord ID (as explained in Section 2.3.2), which defines its position in the ring and also serves as destination address for hosts that want to communicate with it.

After initializing its lists and other class variables, h1 will send a message to itself that forces it to leave after a certain time. For our simulations, we used life times of a minimum of 1000 seconds to a maximum of 1500 seconds and a simulation time of a minimum of 1600 seconds and maximum of 2500 seconds. These values are computed based on the following:

- x = number of hosts. Minimum: 100 hosts , maximum: 2000 hosts.
- Each 0.5 seconds host y_n joins the network at time t_{1_n}.
- Each host will stay for at least 1000s.
- Additionally they will stay for a random time (t_{2_n}) between 0 and 500 seconds.

Note: all time indications are in simulation time. This allows the following conclusions:

1. Host y_n leaves at simulation time = ($y_n * 0.5s + 1000s + t_{2_n}$) seconds
2. The total simulation runs between ($x * 0.5s + 1000s + min(t_2)$) and ($x * 0.5s + 1000s + max(t_2)$) seconds. Actually, this interval is too large since $max(t_2)$ could also be the random time of the second last host.

After h1 is sure that it leaves when it is supposed to, it joins the Chord network by sending a bootstrapQueryMessage to the bootstrap host as shown in Fig. 3.3. The first host to join the network will receive an empty list of potential successors from the bootstrap host because there are no hosts in the network yet. This tells h11 to create a new network. This is achieved by

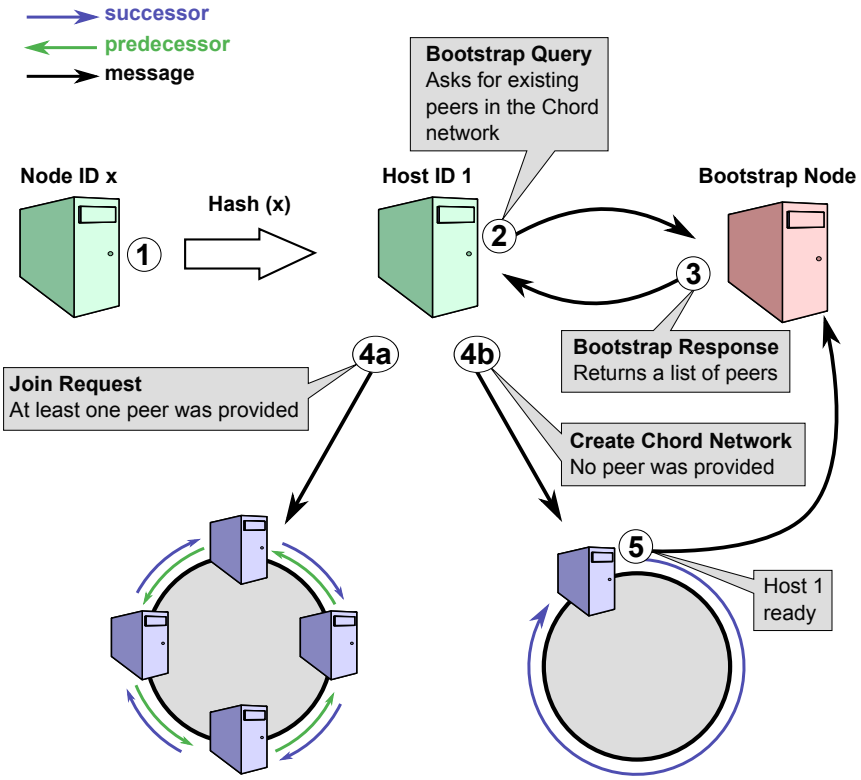


Figure 3.3: Chord bootstrapping example

adding itself as a successor, setting its predecessor to -1 and sending a bootstrapInformReady message.

If the list returned by the bootstrap host is not empty, then the same joining procedure is performed as has been shown in Section 2.3.3. In the following, the joining procedure will be explained in more detail based on Fig. 2.5.

Having received a list of existing hosts in the Chord network the joining host, which in our example got a Chord ID 21, will now search this list for the host having the next smaller Chord ID than its own (host 8) to provide its first successor. The successor of host 21 could either be the current successor of host 8. In this case if we look at Fig. 2.5(a) the successor of host 8, which is host 34 really is the successor of host 21. But if the host did not get Chord ID 21 but Chord ID 35, host 8 would have to route the request to the closest predecessor of the joining host it knows about, which would then return the successor or forward the request again.

After receiving its first successor, host 21 is considered joined and can then inform the bootstrap host of its active state.

But at this point, if we have a look at Fig. 2.5(b), host 21 neither knows its predecessor nor any fingers and the Chord ring is not yet closed. To solve these problems, host 21 first notifies its successor of its presence. The successor will notice that host 21 is its new predecessor and updates this information immediately. After this is done, the successor of host 21 (host 34) will

send back two messages. One that allows host 21 to decide if it really has the correct successor (e.g. the predecessor of its current successor is it self) and one containing a list of fingers, which will serve as an initial finger table because it does not know any nodes apart from its successor yet.

The first message will contain host 21's successor's (host 34) correct predecessor. It might be that this is not host 21 because a wrong successor has been provided in the first place. This happens if another host also joined in between host 21 and its successor or if the predecessor of host 21, which is the host (host 34) that provided the initial successor did not have the right successor yet. In this case, host 21 would ask its successor's predecessor and so on until it gets to know its right successor.

The second message will initialize the finger table of host 21. After that, host 21 will search for new fingers for the positions in the finger table that have not yet been populated. This is being done as described in Section **??** by searching for a host with a Chord ID of $(21 + 2^i) mod(identifierspace)$ with i being the index of the new finger in the finger table of host 21. The host having the closest Chord ID will then answer to the request and thereupon host 21 will add it as a new finger at the given position. The next finger will be added / updated in the next stabilization cycle.

Now that host 21 has a correct successor and several fingers, only the predecessor (host 8) is left to be found. host 21 will not be able to find it by itself but has to wait for the predecessor to run a stabilization cycle in order to discover it as is shown in Fig. 2.5(c) and Fig. 2.5(d). This procedure will also update its predecessor's (host 8) successor (host 34) to host 21. After this is being done, host 21 is seamlessly integrated into the Chord ring and the ring itself is consistant again, meaning that there is no gap in the Chord ring and most of the host pointers are correct.

3.2.3 Leaving Process

As has been described in the previous section, a host sends itself a leave message during the initialization process which will force it to leave after a certain time. Each host will stay for a at least 1000 seconds but can stay longer for a additional of 500 seconds depending on the value generated by the random generator.

If a host 1 receives such a message it begins to leave gracefully, meaning it tries to notify its neighbors about leaving. This is being done by sending them a leafeInformMessage containing either the predecessor or the successor depending on which neighbor the message is being sent to. The neighbors will then delete all pointers to host 1 and add the host contained in the message as a new successor/predecessor.

If this process completes correctly, host 1 will be removed from the network and the Chord ring will be closed again. But, as one can imagine, this is not always the case.

If two hosts next to each other leave at the same time or almost at the same time, the pointers sent to their respective neighbors would not be correct because they point to a host that has left already. In such a case, the next stabilization cycle will repair this if the host has enough successors before the gap occures. In the previously described situation, this would require at least 3 successors, since two hosts left. Or at least 2 successors have to be communicated to the

hosts left neighbors.

As mentioned in Section 3.2, the more successors a host stores and the more often the stabilization cycle runs, the more robust the network is. But, this is problematic, since finding / maintaining successors produces traffic and storing them needs memory. Furthermore, stabilization generates a lot of unnecessary traffic even if in most cases nothing needs to be updated. This behavior can be improved by several optimizations as we will show in Chapter 4.

Informing the neighbours while leaving works very well but if we enable Sender Driven Multicast as will be explained in Section 3.2.4, each host has to forward multicast messages to each finger in a certain range. This has a major impact on the leaving process, since hosts having fingers pointing on host 1 are not being notified because they are not known by h1. This leads to a significant decrease of totally received multicast messages, since a whole subtree might not be served because of a finger pointing to an unreachable (left) host. Normally, stabilization takes care of that, but in an application where hosts join and leave all the time, the stabilization mechanism would take far too long to correct the wrong finger entries, since only one finger is updated at a time to conserve bandwith.

The only way to eliminate this problem is by host 1 storing each host that has a finger pointing to it and then notifying each of those hosts when it leaves. This requires many hosts to be stored that do not directly contribute to the function of the network. But as we will show in the evaluation section in Section 5, this really brings a huge advantage. What also has to be considered is that:

- the number of fingers per host is limited depending on the identifier space, so there is no additional overhead generated from a host as soon as its finger table is filled.
- fingers are not updated very often due to the large stabilization interval, which we could easily increase using the optimizations described in Chapter 4, meaning there is not a lot of changes in the finger table that could cause additional communication traffic.

Unfortunately, it is not enough for host 1 to notify the fingers pointing to it and then just leave, because before host 1's parent receives this message, it may send another multicast message, which would not be forwarded by host 1. To prevent this, host 1 has to delay leaving the network until it gets a confirmation by its parent. Of course, during this time span, it should not react on any messages but the multicast messages and the confirmation by the parent in order to not distribute wrong information, especially to its neighbours.

3.2.4 Sender Driven Multicast

The basics of Sender Driven Multicast have already been explained in Section 2.2.3. Here we will show how this is done in Chord.

This is very straight forward, since Chord uses fingers that are well distributed over the whole network. What else would be better than using those to actually distribute our multicast messages?

Because we are using a core based tree, a host that would want to send a multicast message has to always send it first to the root. The root then forwards this message to each finger it has stored in its finger table, thereby attaching a range to the next finger so that the receiver knows

for which range it is responsible for. Figure 3.4 shows the root (host 1) distributing a multicast

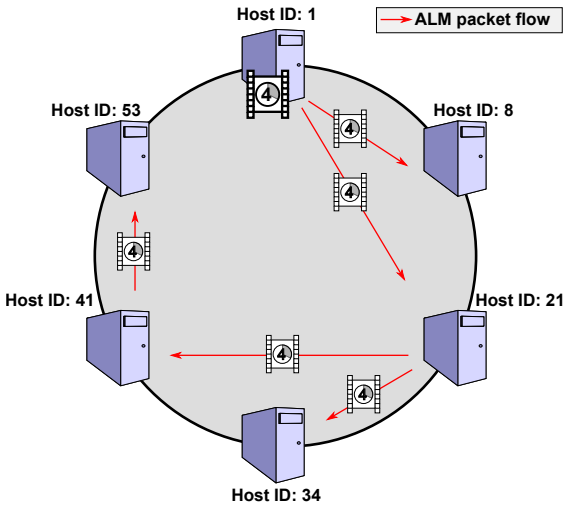


Figure 3.4: Chord Sender Driven Multicast Example

message (e.g. a part of a movie) over the whole network. Because it only knows host 8 and host 21, it forwards the message to them. host 8 will receive a range from 8 to 20 and host 21 a range from 21 to 0. Since host 8 knows no hosts in between itself and its successor, it does not forward any messages. Host 21 knows host 34 and host 41, so it has to forward the message to them. This process is being repeated until each host in the network received the message.

Even tough this is a very simple approach, it has its advantages. First, no additional structure has to be maintained to be able to receive/send multicast messages, since fingers are needed anyway. This saves communication traffic and memory, but it also guarantees a high availability since as soon as a host joined the network, it is also able to receive multicast messages. We also assume that we take the predecessor improvement described in Section 4.1 into account. Second, it is very flexible, since a host can choose the fingers it wants to send the multicast messages to (at least in our improved version, respecting a Fan-Out limitation). The third advantage is that if we only send the multicast message to a selection of all fingers and try to have as many Chord ID's as possible between the selected fingers, we get a very well balanced tree. This is important because the better the distribution of the tree is, the shorter are the paths from the root to the leaf hosts. Therefore, the overall hop count and RTT to Root values are decreased.

Unfortunately, the Sender Driven Multicast approach also has its disadvantages. The major issue is that receivers do not have any control where they get their multicast messages from and thereby they are not able to influence their RTT to parent or their RTT to root. This may lead to jitter, because it is possible that the parent or any other host from the root to the receiver changes all the time, and therefore, some messages might take a different path. This means that any time critical application like VOIP (Voice Over IP) or a video conference cannot rely on a Sender Driven Multicast service. This problem can be solved either by adding more complexity to a proximity aware network like NICE as we will show in Section 2.3.4 or by using an additional overlay network, which allows the hosts to search for a parent supporting their requirements. As

wee have seen in Section 2.2.4 this is exactly what Receiver Driven Multicast is for as we will present in Section 3.3.

3.2.5 QoS

In Section 2.4, we have already described how several QoS parameters [2] can be weighted and aggregated into a single value (the QoS class) following a total order relation. This enables us to build the multicast tree according to the host's QoS classes, meaning the host with the highest QoS class will be the root and the one with the lowest a leaf of the tree.

In Chord, this can be achieved very easily by dividing the whole identifier space into as many segments as there are QoS classes to be supported. Before joining the network each host already knows its QoS properties like bandwidth or cpu power and how to weight them in order to get its QoS class. This weighting of course differs from application to application.

For the given QoS classes (e.g., bandwidth and CPU power) it would be reasonable to say that the bandwidth is weighted higher (e.g. 0.75) and the CPU power only with 0.25. The CPU power would be useless if not enough bandwidth is provided, since the CPU would simply not have enough data to process.

Using its QoS class and the total number of QoS classes (provided by the host initialization message), a joining host can now compute the segment it needs to join to. As we can see in

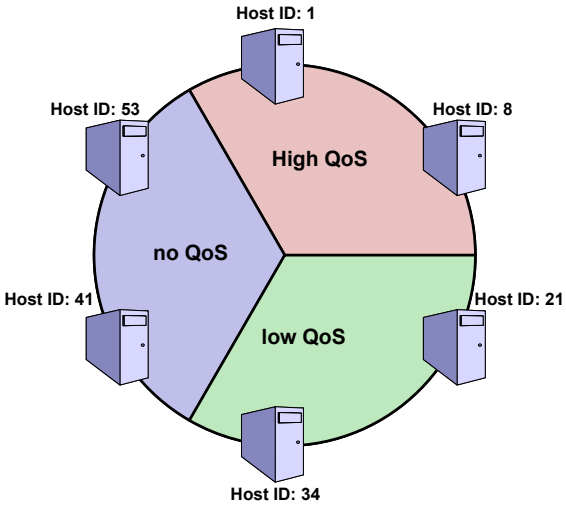


Figure 3.5: Chord QoS

Fig. 3.5 the total order relation as explained in Section 2.4 and the core based tree shown in Section 2.8 are fulfilled if only the root of the network sends multicast messages.

But we still have a problem. What if a host notices that its QoS properties are not met by the network anymore (e.g., not enough bandwidth is provided)? In such a case, this host can try to rejoin (which will change its ChordID) to the same segment which will give it the opportunity to receive Multicast Messages from another host, thereby having a chance to receive traffic accordingt to its QoS class. Of course, it is possible that it rejoins at the exact same position or

that it self cannot guarantee the required QoS class. In this case, it is forced to reduce its QoS class. Hence, it has to rejoin in another segment because also hosts that are in the same segment served by this host would not have their QoS properties met.

3.3 RTT to Root Guarantees

With the growth of the Internet, also multimedia applications like VoIP or muliplayer games are also increasing. The problem is that they usually require a lot of bandwidth and often depend on a low latency and low jitter. We have shown before that if we extend Chord with Application Layer Multicast as explained in Section 3.2.4 and the support for QoS as explained in Section 3.2.5 we can significantly reduce this problem since the overall bandwidth is reduced by using multicast and bottlenecks are removed by using QoS mechanisms. Using these extensions, Chord can support many static properties like bandwidth or CPU power. But it cannot support dynamic properties that rely on a a end-to-end path (to the root) like RTT to Root, overal Hop Count, etc. This Sction will now show a new way how these problems can be solved or at least present methods that try to solve the problem. As mentioned before a perfect RTT to Root guarantee will not be possible because of the dependency to the underlying physical network.

3.3.1 Receiver Driven Multicast in Chord

We have shown in Section 3.2.4 that Sender Driven Multicast is a very simple and handy approach to distribute messages over a Chord network supporting basic QoS properties. But, we also showed that it lacks support for RTT to Root guarantees and the freedom for a host to be able to choose where it gets its messages from.

This is why a new approach based on the Receiver Driven Multicast model as explained in Section 2.2.4 has to be used.

But, since Chord's hosts are not organized by physical proximity, they do not have any information about hosts that could serve as a parent supporting their required RTT to Root. There are other P2P networks though, such as NICE (shown in Section 2.3.4, that do respect physical proximity which we could use to implement a framework that provides RTT to Root guarantees. But because the parent is selected based on the graph-theoretical-center and not selected by the receivers and their requirements it is a sender driven multicast network. This would need to be changed even though the receivers get good RTTs but if they are not able to search specifically for a parent that supports their RTT to Root requirements they will maybe never find one. It would first require the hosts to know their RTT to Root, which could be done similar to the approach we are going to use later in this Chapter and second the hosts would need to find a cluster leader based on their RTT to root requirements and not based on the RTT to parent. This would be pretty easy to implement, but the downside is that NICE is very complex and uses much communication traffic, mainly due to the usage of heart beat messages and the reorganization of clusters. Another disadvantage is that the Fan-Out can be very high, especially for the root since it is responsible to distribute messages in a cluster on each layer. An option would be to reduce the cluster size, but this would still not facilitate control over the Fan-Out, since the number of

layers would need to be increased and also the clusters would have to be split more often. Also, because of its complexity, it would not be the best technology to implement as a framework. A framework should be as simple as possible in order to reduce the communication traffic with the underlying network to a minimum which should prevent many synchronisation problems.

So, we decided to implement an approach with less structure supporting RTT to Root guarantees and offering control over the hosts Fan-Out. In our opinion, this is very important in order to support many different applications and many hosts with different QoS reqirements (bandwidth, latency , processing power etc.,)

The following Sections will describe how a host finds its parent, how it reacts if its RTT to Root is not fulfilled anymore and finally how it interracts with the framework (leave, join, etc.) that it left and what the consequences of these actions are.

3.3.2 Mechanism for searching parents in Chord with ALM support

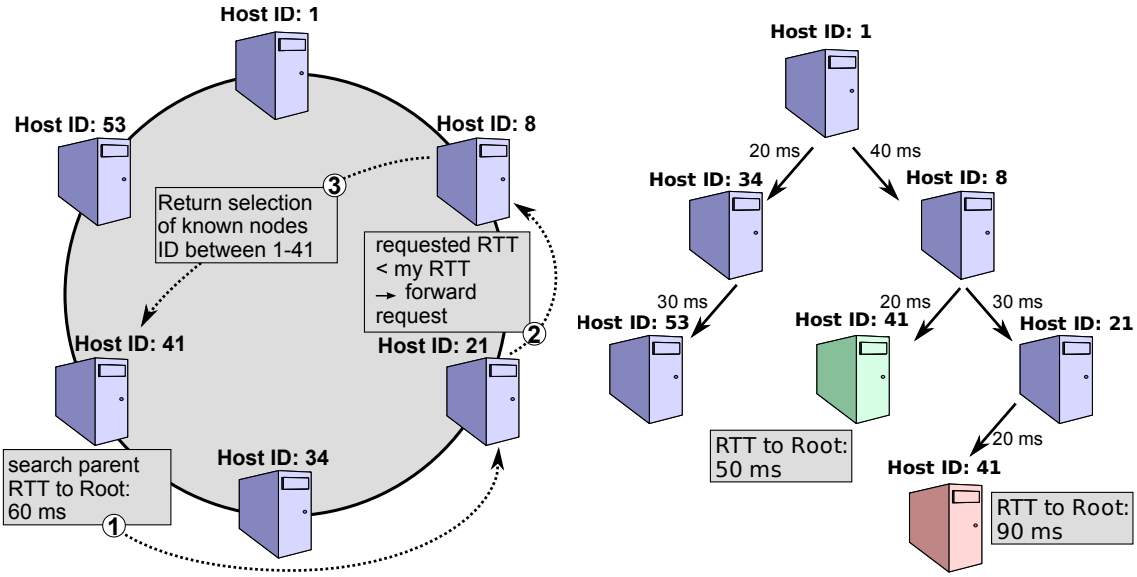
If a host joins a Chord ring, it first has to find a parent that can deliver Multicast Messages supporting the hosts RTT to Root constraints and that does not serve too many children yet.

The required RTT to Root is application dependent and in our case will be given by a random parameter in the host's initialization message. This paramter is between 100 and 200 ms, simulating the host's capabilities. In a real network, this parameter would first be set to the lower bound of the application's demands (e.g., 100ms for an audio conference application) and then be increased to the upper bound (e.g., 200ms) if, for example, the host would have a very bad Internet connection. As we will show later, a host will need to search for a new parent if its requirements are not fulfilled anymore. In a real network, we could additionally use this information (e.g., the host had to search a new parent 5 times in 1 minute) to adapt the parameter even further, because obviously in such a situation the host has either problems with its Internet connection or it is too far away from any other hosts of the network.

The constraint to look for a parent that does not serve too many children yet has been introduced to limit the Fan-Out as we will show later on.

The mechanism to find a parent is explained with the example in Fig. 3.7. As soon as host 41 joined the network, meaning it has at least one valid successor, it can start searching for a parent. But, since it does not have any proximity information to direct its search to a specific host, we decided to concentrate on a even distribution of the multicast tree if we look for a potential parent. This can be done by the newly joined host (host 41) by first sending a search parent message containing its required RTT to Root to the next host that satisfies the following condition: $chordID >= (myChordID/2)$. In Fig.3.7 this would be host 21. This also assures that the root (the host with the lowest chord ID: host 1) will not be asked first, since this would be the optimal parent for every host and would not match to the bottom-up approach we want to use.

As soon as the targeted host (host 21) receives this parent request, it checks if itself already has a valid parent (host 8) and that the RTT to root over this parent (e.g., 70ms) is smaller than the requested RTT to root (e.g., 60ms). If this host does have a valid parent and finds out that its RTT to Root is already higher than the newly joined host's requestet RTT to root it forwards



(a) Chord Network

(b) Receiver Driven Multicast Tree

Figure 3.6: Search a parent with forwarding

the request to its parent (host 8), since this host will have a lower RTT to Root. host 8 then assembles a list of potential parents consisting of itself, its children, neighbors and fingers, and sends it back to the requesting host (host 41). What also has to be considered is that this list has to consist of hosts having Chord IDs smaller than the one of the requesting host (host 41) in order not to violate the QoS principle explained in Section 3.2.5. In a large network, this list could get very large, but this can be controlled by a parameter distributed in the host initialization message. To reduce the lists size, host 8 first inserts all the children, fingers and the predecessor as before, but then it will remove every second host in the list until the list is smaller or equal to the given parameter. The removal of every second host ensures that the host requesting a parent gets a good distribution of hosts all over the network. Providing this list of hosts is actually, apart from the multicast messages, the only thing we have to consider when implementing a P2P network that wants to use the RTT to Root framework. The framework will send a request containing the host ID, P2P ID and the QoS class of the host that searches a parent (host 41) in order to receive a list of hosts from the network, which it can use to find a parent as will be explained in the next paragraph. The P2P ID or the QoS class is required in order for the P2P network to get a selection of hosts that also support the QoS properties.

In any case, the requesting host will receive a list of hosts sooner or later. These hosts can then be QoS pinged in order to find out what their RTT to Root is and how much time is needed to reach them. A QoS ping works similar to a normal ping but it mesures the delay over the whole path from a host to the root. To do that it first pings its parent, which will in turn ping its parent and so on until the root is being reached. The pongs will then be added together and sent back to the QoS pinging host. Adding the RTT to the pontential parent and the potential parent's RTT to root together will result in the RTT to Root of the requesting host (host 41) if it were to use

the just pinged host. After getting back all the requested delays the host saves this information to speed up future searches and to prevent pinging the same host twice. Now, it can decide how to proceed further.

Lets assume for the example presented in Fig. 3.7 that host 8 provided a list of hosts, which were QoS pinged by the joining host 41.

There are the four cases to consider:

1. There is at least one host that does not have too many children yet and can support the requested RTT to Root.
   → Randomly select one of those hosts fulfilling the requirements and ask it if it can serve as a parent. It may be that this host cannot be the parent at the time the request comes in because other hosts may registered themself as children in the meantime and there is no more space in the children list or the potential parent wants to leave or has already left.

2. There is no host supporting the requested RTT to Root.
   → Search the host with the best RTT to Root (lets call it host x) and do one of the following:

   (a) If host x is a neighbor (predecessor, finger, successor) of the host that initially provided a list of potential parents (host 8), ask its parent if it meets the requirements (RTT to Root, Fan-Out). If it does not, ask it to provide another list of hosts so the joining host 41 can QoS ping them in order to find more possibilities to attach itself to. For example, host x could be host 53 (e.g. if it is a finger of host 8) and its parent would be host 34, which will naturally have a better RTT to Root than host 53.

   (b) If host x is a child of the host that was initially asked to provide a list of potential parents (host 8), then ask this parent's parent if it can add the joining host 41 to its children list, supporting its RTT to Root requirements. This is done because the host that provided the list in the first place (host 8) was also QoS pinged. It was included in that list and obviously did not support the required RTT to Root. Or, it already had too many children, because otherwise the joining host could have chosen this host as a parent.

3. There is at least one host that supports the required RTT to Root but it has already reached the maximum of children it can serve.
   → Do the same as in case 2. The most intelligent action to take here would be to ask a sibling (in the multicast tree) of the host that would support the RTT To Root but has no more space in its children list to be the parent since the number of hops would be identical. But, since a host in the tree only knows which hosts its parent and its children are, this is not possible without adding more complexity.

4. Only the root is being returned, because the host is one of the root's successors. But the root already reached the maximum children it can serve.
   → Ask the root to delete one of its children, which is not in its successor list and add the current host instead.

Because the process above repeats itself, the host will sooner or later find a valid parent if there is one. It can be that there are simply too many hosts in the network. In such a case there would be no host that could satisfy the request because all the multicast paths are getting too long. Or, all hosts that could serve as a parent with a QoS class higher or equal to the requesters have already reached the maximum of children they can serve.

To prevent indefinite loops, each host tries to find a parent for a maximum of 6 times. If it does not find one, it waits for 10 seconds and tries another 6 times. This makes sense especially if other newly joined hosts hinder its search or if hosts leave and make space for new ones. Also, the search paths will vary because of the random selection of potential parents.

But, in any case, the host has to have a parent in order to receive Multicast traffic. Since even with a RTT to Root that is higher than the peers requested minimum, it may still be that the host is able to use the provided information. To do that, it will simply attach itself to the host with the best RTT to Root of the previously QoS pinged hosts that still has space in its children list and that has a lower or equal QoS class. After that, it will start a wait timer and try to find a parent that can support its RTT to Root later on.

Now that the newly joined host has a valid parent, it may still be that its RTT to Root changes because of leaving or joining peers. This situation will be discussed in the next two sections.

3.3.3 Dynamic Behavior of Peers

As we have shown in the previous section, the RTT to Root of a host can change over time. The following situations can cause a RTT to Root change in host A:

1. A peer in the receiver driven multicast tree positioned between host A and the root leaves.
2. The underlying physical network conditions change.

In order to handle the second case, we would have needed to simulate a real network with all its flaws and random congestion. Unfortunately, this is out of the scope of this master thesis which is why we only considered the first case.

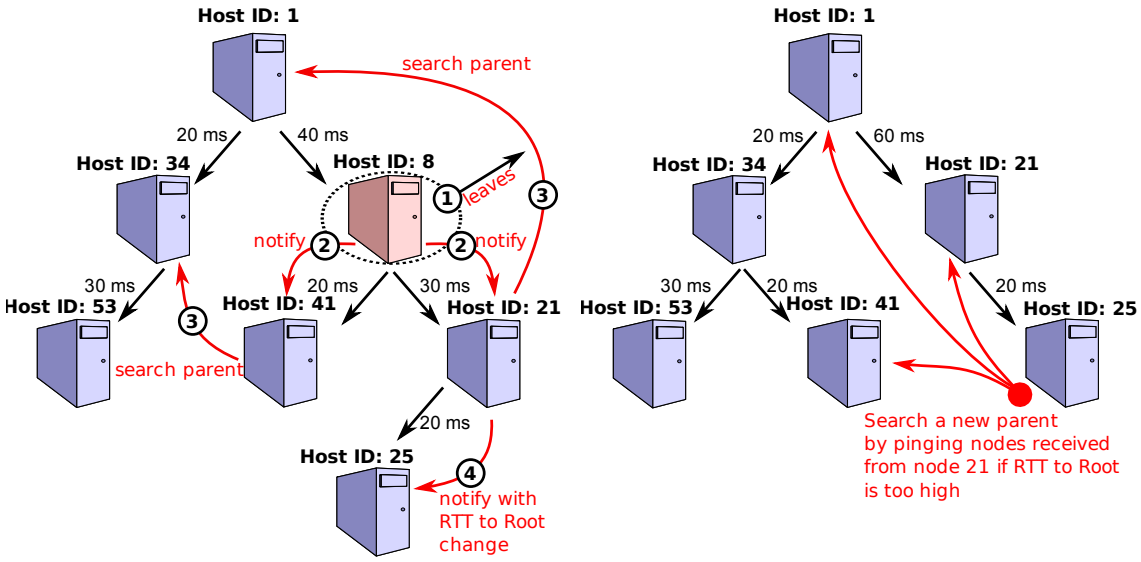
But, with the first case at hand, we can also handle the second one since the network knows how to adjust to changes. The only thing that would be left to implement would be a detection of an RTT to Root change. This could either be done by measuring how long a multicast message was on the way which would require a synchronization of the clocks of all the peers or by using a periodic QoS ping measurement.

But, if the underlying physical network is not stable, meaning the paths change all the time and jitter varies due to congestion, it is very difficult if not impossible to support RTT to Root functionality. The only way to realize proper RTT to Root guarantees for people all over the world would be a native implementation on the physical layer. There are many different ways how to do this but in the end all of them have the same problem: It would require massive changes in the routers, meaning many of them would need to be replaced It would alsobe very hard to introduce a billing mechanism and it would require a lot of contracts between providers. And these are just a few problems. As of this moment it looks like a world wide QoS functionality on a physical layer will not be available anytime soon.

This makes clear that we can only approximate RTT to Root guarantees, but also that within a more or less stable network, we would get a huge benefit anyway.

Now that we have clarified, that the overlay could also react on real changes in the physical network, the next Section explains how hosts react if other hosts leave on their path to the root.

3.3.4 Leaving Process



(a) Reaction of directly affected hosts

(b) Reaction of indirectly affected hosts

Figure 3.7: Impact of a host leaving the network and reaction of its children

In Fig. 3.7, host 8 leaves the network. In Section 3.2.3, we described how the host leaves if we use Sender Driven Multicast. Now, we want to know what happens if a host leaves in a Receiver Driven Multicast environment.

First, it does pretty much the same as described in Section 3.2.3, notifying its neighbors and its parent. But now, it does not need to inform hosts that have a finger pointing to it since the multicast messages are distributed using a separate tree. What it has to do is to notify its children because otherwise they would not receive multicast messages anymore. It does this by sending them a list of potential parents as it would if it got a search parent request as explained in Section 3.3.2. Its direct children (host 41 and 21) will then start searching a new parent by QoS pinging the received hosts. As soon as they found a new parent supporting the required RTT to Root, they will again inform their children (host 25) with the new RTT to Root. These children can then evaluate if their new RTT to Root is within their requirements or not. If they are, these hosts will inform their children as their parent did. If the requirements are not met, they will search for a new parent as their parent did before and then inform their children about the changed RTT to Root.

In the example presented in Fig. 3.7(b), host 25 was lucky because its new RTT to Root is even

better than before. Hence, it does not have to search for a new parent. It could also have been the case that its parent (host 21) attached itself to another place in the tree (lets assume host 53). Then, it would be very likely that its RTT to Root is not fulfilled anymore. Hence, it would also need to QoS ping the hosts communicated by host 21 in order to find a new parent as indicated in the example in Fig. 3.7.

Chapter 4

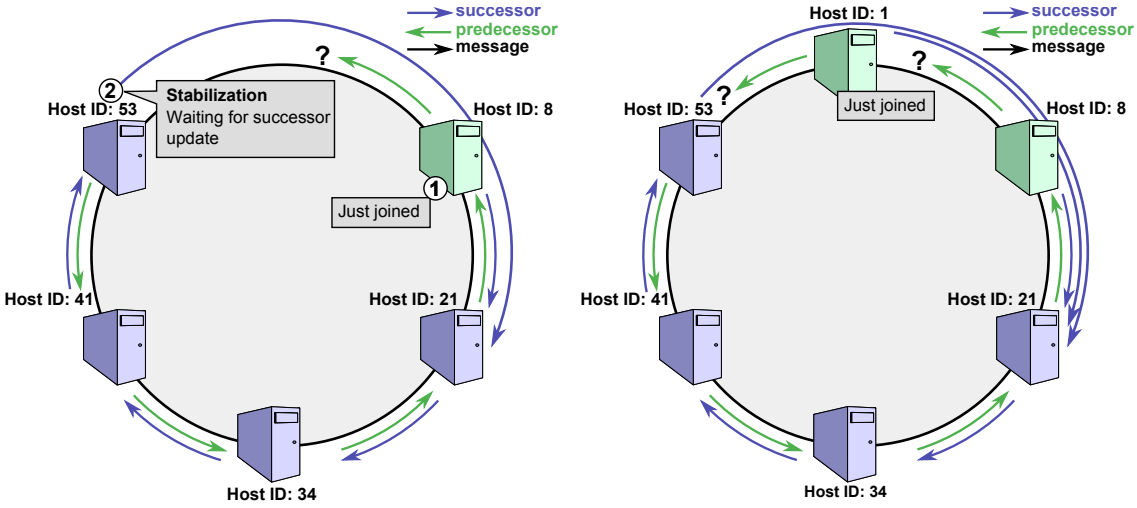
CHORD Robustness and ALM Improvements

As mentioned previously we need to make some adjustments to the native Chord network in order to enhance the multicast reliability, improve its robustness, reduce redundancy and thereby shorten search paths and the path to the root. Since Chord's structure is built upon successors, fingers and a predecessor, it was obvious that we need to improve their lookup, diversity and their correctness. Sections 4.1 to 4.4 will discuss the problems that arose during the implementation and how we fixed them.

4.1 Predecessor optimization

Predecessors in Chord are used for stabilizing successors. This makes clear that correct predecessors are important in order to have a consistent network, since the network relies on correct successors. The problem is that predecessors are, similiar to the successors, only updated within the stabilization procedure.

As shown in Fig. 4.1, a newly joined host (in this example the host with Chord ID 8) does not have the possibility to search its predecessor itself but has to wait for its predecessor to stabilize in order to get to know it. During this time, it will not receive any multicast messages since just succeeding hosts had a chance to get to know it. Also, its predecessor would not answer successor or finger requests correctly and its predecessor would distribute wrong pointers if it were to leave. The problem gets even worse if 2 hosts join next to each other in a short time interval ($\Delta t < stabilization\ period$). This happens pretty often if there are not many hosts in the network yet but there are many hosts joining (for example if a new multicast group was opened). To overcome these problems, we enabled the joining hosts to search their predecessor themselfs as shown in Fig. 4.2. This is being done by sending a request to the successor since this is the only host they know after joining. This successor will then forward the request to the closest preceding host of the requester it knows about, which will then do the same until the predecessor is reached. The predecessor will then answer the request and add the newly joined host to its successor list. If we compare Fig- 4.1(b) with Fig. 4.2(b) we can see that there are much less wrong pointers. What has not been shown is that host 53 in Fig. 4.1(b) would actualy



(a) Situation of a newly joined host

(b) Impact of 2 simultaneously joined hosts

Figure 4.1: Joining hosts producing wrong pointers

have 2 wrong successors, since we work with 3 successors and none has been updated because stabilization did not yet kick in. Of course, all these problems could be solved by a shorter stabilization period but this would generate much more unnessesary communication traffic.

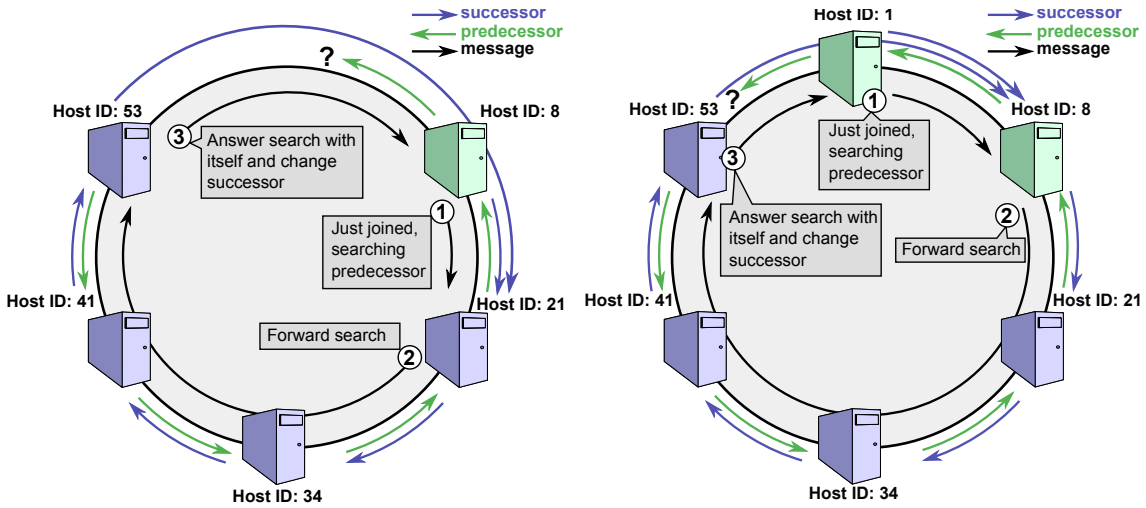
Additionally, we try to react better on updated predecessors. Looking at Fig. 4.2(a), if host 21 receives a new, better predecessor (host 8), it is very likely that its old predecessor (host 53) does not have the right successor anymore. Otherwise, it would not have used host 21 for stabilizing its pointers. In this case, host 21 can inform host 53 that host 8 might be the more accurate successor.

Advantages

- More robustenss due to to faster correction.
- Faster multicast readiness.
- Stabilization does not have to run as often as with the native approach.

Disadvantages

- Slight overhead generated by lookup. But because this, in contrast to a more frequent stabilization, only affects one host this is not really a problem since overall the overhead is being reduced.



(a) Newly joined host searching its predecessor

(b) Impact of 2 newly joined hosts in succession

Figure 4.2: Self predecessor discovery

4.2 Finger optimization

As explained in Section 2.3.3, having a finger table speeds up lookups and as being described in Section 3.2.4, it is also responsible for the multicast message distribution. Unfortunatelly, it is possible that many fingers point to the same host as shown in Fig. 4.3(a). This is an example using a very short identifier space of 100 possible IDs but in a real world application we would have an identifierspace of 2^{128} (SHA1) possible IDs. This would, especially for sparsely populated networks, lead to big gaps between IDs of two consequitive hosts. This results in many fingers pointing to the same host because a host x tries to get fingers that are 2^i IDs away from its own Chord ID. This means that for a small i, there is a high probability that the next host will be host x's successor.

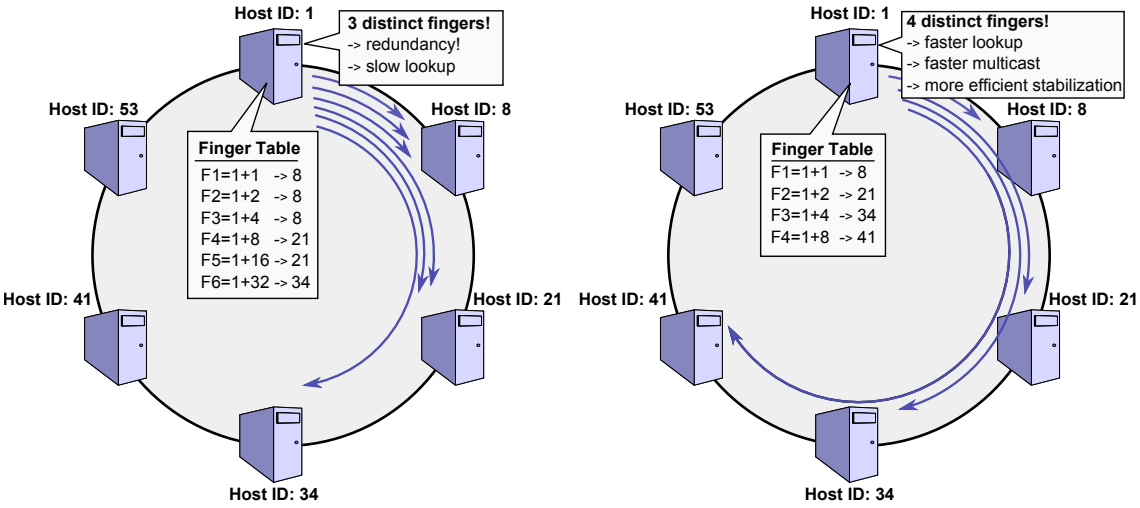
In Fig. 4.3(b), we show how the fingertable of host 1 could look like if we use the following approach:

1. After joining, insert fingers provided by the successor at position i if $fingerID >= (myChordID + 2^i) mod (identifierspace)$.

2. Always insert the first successor if it is updated.

3. Consequitivly, try to get a finger at an unpopulated index until the maximum of $(identifierspace) log(2)$ fingers is being reached, then try to find better fingers for incrementing positions in each stabilization cycle.

4. Inform new fingers in order for them to be able to notify host 1 if they leave. This prevents host 1 from sending multicast messages to left hosts if Sender Driven Multicast is being



(a) most fingers not distinctive

(b) optimized fingers

Figure 4.3: Distinctivness of fingers

used. If QoS is enabled, only fingers with higher IDs have to be notified since only they can receive multicast messages from the current host.

5. Never replace fingers but move them back in the list if a better finger was found for the current position. This ensures that we get a greater diversitiy of fingers all around the Chord ring.

6. Only delete fingers that point to left hosts, fingers that do not satisfy the condition defined in point 1 (because they were moved backwards in the fingertable due to better suited pointers) or if the maximum of $(identifier\ space)log(2)$ fingers is being exceeded.

7. Insert fingers independent of the stabilization cycle (e.g., random hosts the current peer comes accross). They are only inserted if their ChordID is between $(myChordID + 2^i)mod(identifierspace)$ and the finger's ID at position i. Also, the host will not be inserted if the message was a Timeout or a Leave message.

8. If a finger left, immediatelly update its position.

The following list show all the advantages gained through the optimized finger table. There are no disadvantages as far as we can tell.

Advantages

- Faster lookup (more distinctive fingers).
- Faster finger aquisation. Especially in a new network where many hosts join, many pointers to hosts are inserted according to point 7 and are only deleted if one of the conditions in point 6 is met.

- Shorter multicast paths since more distinctive fingers are known.

- Simplified multicasting. Because successors are always updated in the fingertable as well, they do not need to be considered in Sender Driven Multicasting.

4.3 Sender Driven Multicast optimizations

As explained in Section 3.2.4, fingers are being used to distribute multicast messages. The problem is that especially the root can have a very high Fan-Out, depending on the size of the identifier space. In a normal Chord network with a 128 bit identifier space, the root would in the worst case need to send Multicast Messages to 128 hosts. Depending on its Internet connection and the bandwith used for the communication, this would be too high, especially if there are other solutions.
In Fig. 4.4(a), we see an example of a very small network illustrating the problem. The root knows every host except host 53. It also serves all those hosts but host 53, which will be served by host 41. The problem can be omitted very easily by only using portions of the fingertable as shown in Fig. 4.4(b). The question now is which fingers should be selected for multicast distribution since there are different ways to do that. The following algorithm is very simple but achieves good results:

1. Select all fingers.

2. Check if ($\#selected\ fingers < max\ Fan\text{-}Out$) if true proceed with the multicast distribution, if not proceed with step 3.

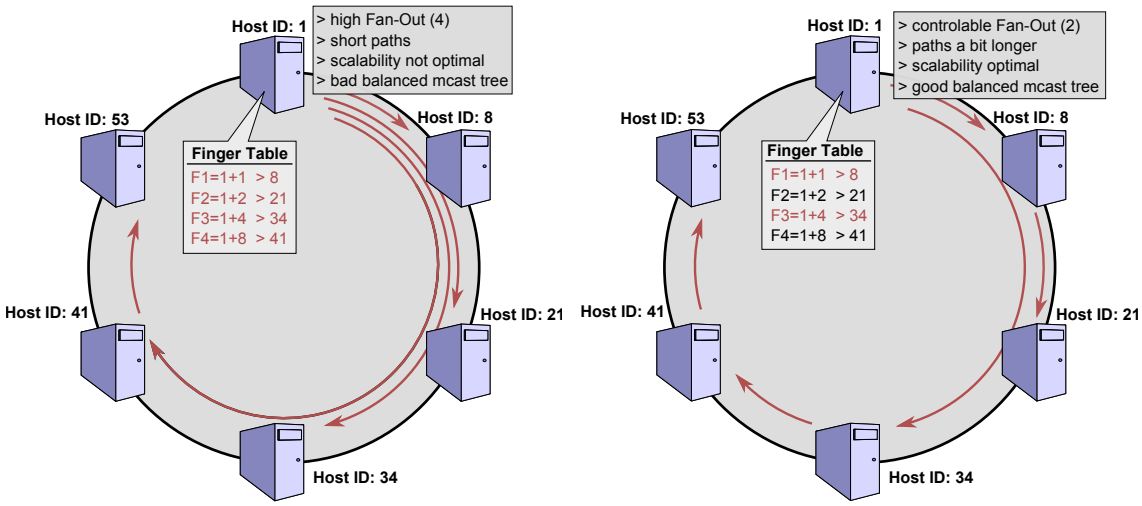
3. Beginning with the second finger (second successor), deselect every second host as long as the number of selected hosts exceeds the maximum Fan-Out. The first finger (successor) cannot be deselected because otherwise it would not be served by any host.

4. Repeat step 2 until the number of selected fingers equals the maximum Fan-Out.

The previously described procedure ensures that fingers are evenly distributed over the network, since it selects every second host. It also priorizes fingers that are far away by starting to deselect hosts at the beginning. This is important because Chord hosts know anyway more about their closer neighborhood because of their successors. Figure 4.4(b) illustrates the optimized finger selection and the impact on the multicast distribution. Instead of serving 4 hosts, the root now just has to serve two.
Advatages

- Controlable Fan-Out.

- Better distributed multicast tree because also hosts close to the root can distribute messages.



(a) native multicast distribution

(b) optimized multicast distribution

Figure 4.4: Native vs optimized multicasting

- Flexibility. Depending on the application the Fan-Out can be adjusted. Very delay critical applications would select a rather high Fan-Out (short paths) while a data mirroring service would reduce the Fan-Out (because delay does not matter in this application but the throughput must be high in order to be able to distribute the data more quickly).

Disadvantages

- The lower the Fan-Out, the longer the paths from the root to the leafs of the multicast tree.

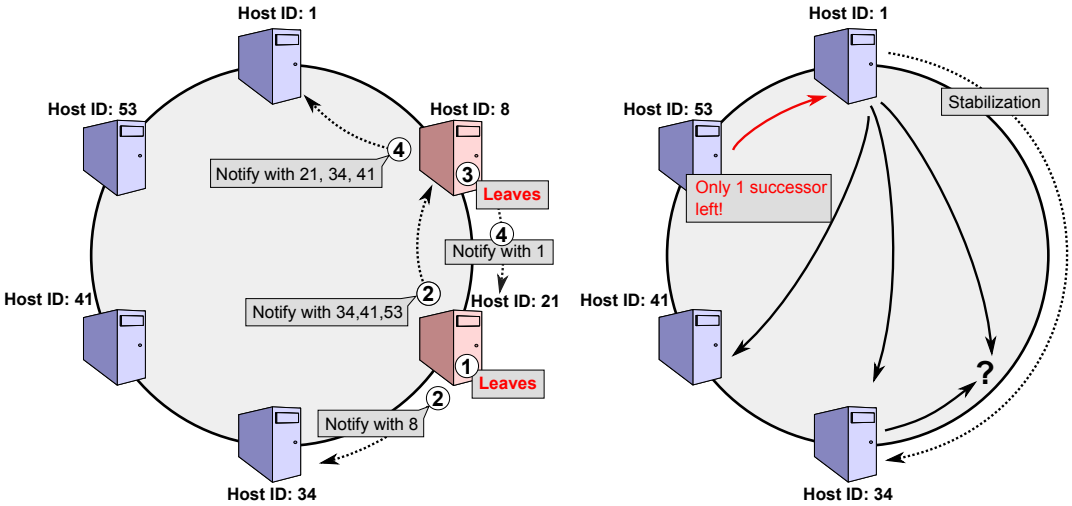
We were very satisfied with the results of these optimisations, but there is one thing that could be optimized even further, namely, the selection of fingers. It would result an even better distribution if the algorithm would try to select hosts with $IDs = myID + (identifierspace/maxFanOut) * i : i \in 1...maxFanOut$ to ensure that two consecutively selected fingers always have more or less the same number of IDs in between. But, this would be pretty hard to implement since the finger table by nature is not linearly distributed.

4.4 Leave procedure optimization

If too many hosts leave the Chord ring in a short time interval, it can happen that the network gets irreparably broken. For example, if k is the number of successors, then it would need k hosts to leave / fail in order for the ring to be broken. Of course, we could increase robustness by storing more successors, but we would still have the problem of invalid pointers that can lead to strange behaviors. And of course, it would generate even more overhead since most of the pointers would not need to be updated during the stabilization.

In Fig. 4.5(a), we can see that host 8 leaves shortly after host 21 does, leading to the situation shown in Fig. 4.5(b). host 1 still has 2 valid successors and when its stabilization cycle runs the next time, it will find out that its first successor is not valid anymore and update it respectively. But, if for example a new host (in this example the host with Chord ID 9) were to join before host 1 had the chance to stabilize its pointers, it would receive the wrong successor 21 and host 9 would need to re-bootstrap in order to get into the network correctly. The same applies for finger lookups.

The worse problem though is that host 53 has only one valid successor left which would lead to a broken Chord ring if host 1 were to leave before host 53 could stabilize. Apart from that, it would also distribute wrong information if pointers were requested.



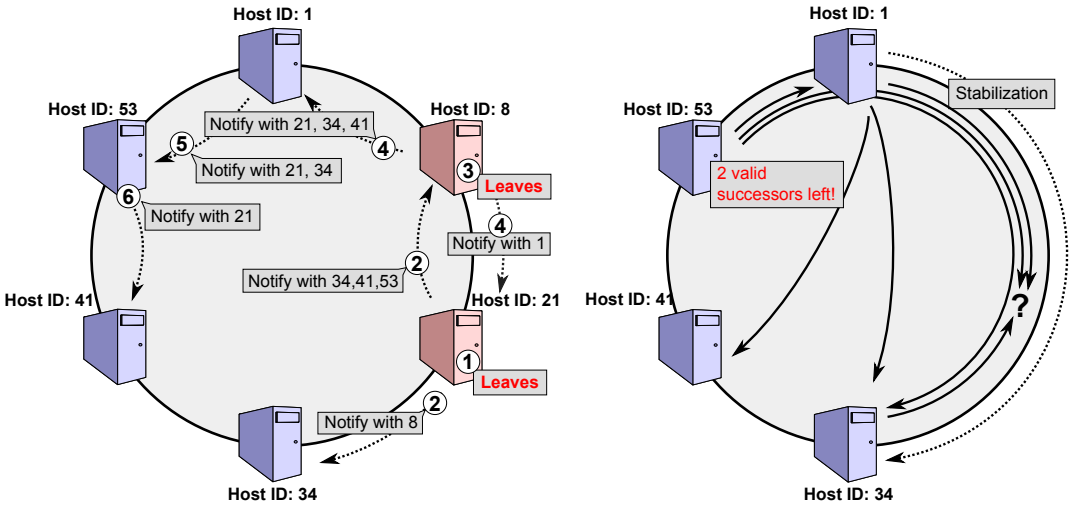
(a) simultaneous leave

(b) impact of simultaneous leaves

Figure 4.5: Impact of hosts leaving simultaneously

To solve this problem, we performed two improvements. First, as shown in Fig. 4.6(a), we introduced a pointer back propagation of left hosts. Meaning, instead of just notifying direct neighbors, other neighbors are indirectly informed as well, depending on the number of successors (in this example 3). In Fig 4.6(b), we can see that host 53 has now two valid successors left making it less probable for the Chord network to get into an inconsistent state.

This makes the network more robust but still the problem of a broken Chord ring remains if three successors leave in a short time interval. To fix this, we introduced a mechanism that enables the hosts to use fingers as successors because they might still have a valid finger pointing to a host on the other side of the gap. In most cases, this will not be the perfect successor, but stabilization will take care of that. Furthermore, we have introduced a list of recently left hosts that prevents hosts from adding pointers to left hosts to their successor, predecessor or finger list if for example a host falsy provided this information because it did not get the leaveNotifyMessage of the host that left. A host is only deleted from the recently left hosts list if itself communicates with the current host and not if it is provided as information from another host. This is because only then



(a) optimized simultaneous leave

(b) impact of an optimized leave

Figure 4.6: Impact of the optimized leaving procedure

we can be sure that the host is being reactivated or another host joins using the old ChordID.

Advatages

- More robustness (because it takes more to break the Chord ring).
- Better lookup of fingers / successors.
- Repair mechanism takes care of a broken ring.

Disadvantages

- More overhead for communicating the successors to a larger neighborhood. But, this is not really a disadvantages if we look at the overhead that can be saved if finger / successor requests are answered correctly

Chapter 5

Evaluation and Results of our Optimized Chord Implementation

5.1 Evaluation Scenarios

We will now show the results of our Chord implementation with different parameters. As described in Section 3.1.3 we used Brite to generate network topologies which resulted in 13 distance matrices as shown in Table 3.1. These matrices contain delays between each pair of hosts. We ran several scenarios as shown in Table 5.1. A scenario is defined by Chord specific parameters that for example enable QoS or Sender-/Receiver- Driven Multicast support. We consiedered to run more scenarios because there are much more Chord parameters that could have an effect on the results. The following list shows the most important parameters and why we did not vary them.

- **Number of Successors**: Looking at the improvements presented in Section 4 we can see that all pointers (Successors, fingers, predecessors) are updated automatically and do not necessarily require stabilization. This leads to much more accurate pointers and also to quicker refreshingings. Performing runs with different numbers of successors did not have an effect on the results because either the runs completed successfully or they did not at all. If the run completed, it showed the same results because we did not consider overal communication traffic since we were only interested at the multicast traffic. In the runs with a lower number of successors, the hosts needed to use a finger to fix the gap which should only be used in emergency situations. The runs with only one successor did not finish all the time because the successors could not update quickly enough if many consequitive hosts left the network, resulting in a fragmented Chord ring. Since a fragmented or open ring should never occure, we did not consider this, leading to a abort condition that was never met and due to that the simulation never stopped. Most of the runs would have completed with 2 successors as well, but we wanted to have a very stable network that can manage up to three consecutive hosts leaving simultaneosly.
- **RTT to Root limit** (only in Receiver Driven Multicast): This parameter has a major influence on the RTT to root fullfilled property, which measures if the hosts designated

RTT to root is met or not. Each host is randomly given an RTT to Root that simulates the host's demands. This value is being generated by a intuniform distribution from 100ms to 200ms. This would result in a minimum delay of 100ms and a maximum response of 200ms in the worst case, which is still acceptable. Choosing a higher delay would of course result in a higher RTT to Root fullfilled value but also to longer latencies. The same but vice versa is true for lower delays. Performing tests from a minimum of 90ms to a maximum of 250ms, we found out that this setup is suited best for an improved Chord network with a maximum number of 2000 hosts.

- **Stabilization**: This parameter has an effect on the robustness of the network. But if hosts leave gracefully which they do in our network and and if they use the improvements explained in Chapter 4 this parameter is dispensable Hence, we set this paramater to 10s in order not to generate too much network traffic. In a real network, the stabilization procedure would need to run more often, depending on the frequency of leaving hosts because hosts can leave ungracefully without telling anyone. In such a case, the improvements also work because if a host detects a failed/left host, it will inform some of its neighbours, depening on the maximum number of successors. But, Stabilization is still necessary to ensure the left host is detected soon enough.

- **Max Fan-Out**: This parameter is set to 7. Meaning, each host distributes a multicastmessage to a maximum of 7 other hosts. If we lower this parameter, the RTT to Root will be longer because of an increased Hop Count and vice versa. In a real scenario, this parameter would be an important parameter to modify if one wants to configure Chord for a special application. This would depend on the bandwith the users have at their disposal and how important low latencies are. It would even be possible for hosts to lower or increase their maximum Fan-Out depending on how many packets get lost or on other network measurments. Because we did not test a real application and because of the varaity of applications that Chord could support, we decided to fix this at a reasonable value that gives us good results considering the RTT to Root limitations.

Considering all the improvements we implemented, it would make sense to compare the improved with the original version. But since the improvements were an immediate reaction of problems we encoutered, they were not implemented as modules that could easily be switched off but integrated into the core functions of our Chord. Since this masterthesis wants to show that it is possible to support QoS and RTT to Root at the same time using Chord, the comparison between the improved and the native Chord version is not in its scope. This means that the advantages of the improvements as described in Section 4 cannot be verified by statistics. But, since we also tried to list every disadvantage we came across one can imagine that these are really necessary and make Chord a faster, better distributed and more stable network that scales at least as good as a native Chord network does. Each scenario involves runs with 100 hosts up to 2000 hosts in steps of 100. Each step is run with all the 13 generated network matrices and for each matrix we used 3 different seeds resulting in a total of 780 runs per scenario. The seeds are used by the random number generators (RNG) as input variables. We have used the Mersenne Twister RNG [18], which has a period of 219937 - 1, and it assures the 623-dimensional equidistribution

Table 5.1: Chord scenarios

Chord - No rejoins - No QoS support - 3 successors - sender driven multicast
Chord - No rejoins - QoS support (32 QoS classes) - 3 successors - sender driven multicast
Chord - No rejoins - QoS support (32 QoS classes) - 3 successors - receiver driven multicast

property. Our network uses this RNG for defining random leave intervals and the generation of Chord IDs. Table 5.2 shows the seeds used. The statistics will always contain plots with a

Table 5.2: Random input seeds

1768507984
33648008
1082809519

maximum, minimum and an avarage graph. The values were computed based on the 39 runs (13 matrices * 3 seeds) of a given amount of hosts. Furthermore we removed 0.5% of the maximum and 0.5% of the minimum outliers, resulting in a confidence interval of 99%. The following attributes were considered in order to be able to compare the different evaluation scenarios.

- **Fan-Out**: The number of children of a host in the multicast tree.
- **Node to Root RTT** The Round Trip Time (RTT) between the host and the root of the multicast tree.
- **Hop-Count**: The number of times a multicast message needs to be forwarded until it reaches its destination.
- **Average duplicates per multicast message**: The number of duplicates a host receives for one multicast message.
- **Received total multicast messages**: The total number of received multicast messages.
- **Node to root QoS**: Measures the number of received messages that passed no host having a lower QoS class divided by the total number of received multicast messages.

5.2 Optimized Chord using Sender Driven Multicast with no QoS support

Figure 5.1 shows the evaluation of our Chord implementation using the optimizations explained in Section 4.

These statistics refelect the native Chord besides more stability, the possibility to limit the Fan-Out and a better balanced multicast tree. This means there is no QoS support and the optimized Sender Driven Multicast approach is being used. We will now discuss the evaluation, based on the previously discussed properties such as Fan-Out and Hop-Count.

- **Fan-Out**

  In Fig. 5.1(a), we see the minimum, maximum and avarage Fan-Out for 100 to 2000 hosts. Due to the introduced Fan-Out limit, we do not see any host having to send a multicast message to more than 7 hosts. Even though our algorithm tries to balance the tree as much as possible without causing more communication traffic, the avarage Fan-Out is still arround 2. This is because the root and its succeding hosts have a much greater pool of hosts they can send multicast messages to, since in Chord's native Sender Driven Multicast, messages are always sent forward due to the range that can only be set correctly in this direction. The minimum Fan-Out is 0 because the leafs of the multicast tree are of course not able to send multicast messages any further. This is also a reason why the avarage is not higher.

- **Hop-Count**

  Figure. 5.1(b) shows the Hop-Count or in other words, the number of hosts that need to be transited until a multicast message reaches its destination. As we can see, the maximum is linearly increasing while the avarage is stagnating at arround 6. This means, the more hosts are in the network, the better the tree is balanced. There are always exceptions though, as the maximum shows. This is because the predecessor of the root will mostly be served by a host that already has a high Hop-Count because it is in a less balanced area of the tree. If we look at Section 4.3, it becomes clear that the selection mechanism may not cover the predecessors of the root early due to the randomness introduced to limit the Fan-Out. But, if we increase the Fan-Out limitation which could be handled differently for each host depending on its bandwith capabilities, the Hop-Count would of course decrease. The minimum is 0 because the root does indeed forward the message for any sender. But, it does not count as an additional hop because it is clear that it always receives the message directly from the sender.

- **Host to Root QoS**

  Even though we did not activate QoS support for this scenario we wanted to show how many percent of the hosts randomly fulfill the QoS criteria defined in Section 2.4 as shown in Fig. 5.1(c). The root will always fulfill the QoS criteria which is why the maximum is at 100%. The avarage is decreasing with the number of hosts, because the chance that a host gets traversed that has a lower QoS class than the receveing host increases with a higher Hop-Count. Because there are only two states (QoS fullfilled $\rightarrow$ 100% or not fullfilled $\rightarrow$ 0%), it becomes clear that if the avarage is not at 100%, there will be at least one host that does not have its QoS class fulfilled which is why the minimum is at 0%.

- **RTT to Root Satisfied**

Now, we want to have a look at the RTT to Root Satisfied property, which is being shown in Fig. 5.1(d). As we can see, there is always at least one host that does not overstep its maximum RTT of 200ms because the root's successor is always closer than that value if we look at the delays being used in Table 3.1 (which has a maximum of 91ms). The avarage begins with 100%, which is plausible if the tree is balanced well enough (avarage Hop-Count is 2). It then decreases to arround 50% because the higher the Hop-Count is, the higher the latencies are in general. This is of course not acceptable because if the Chord network were to host a video conference or online multiplayer game, 50% of the people would have trouble following the conversation or playing the game because their interactions would always come too late, making their participation very unconfortable. For the minimum, the same applies as previously described.

- **Host to Root RTT**

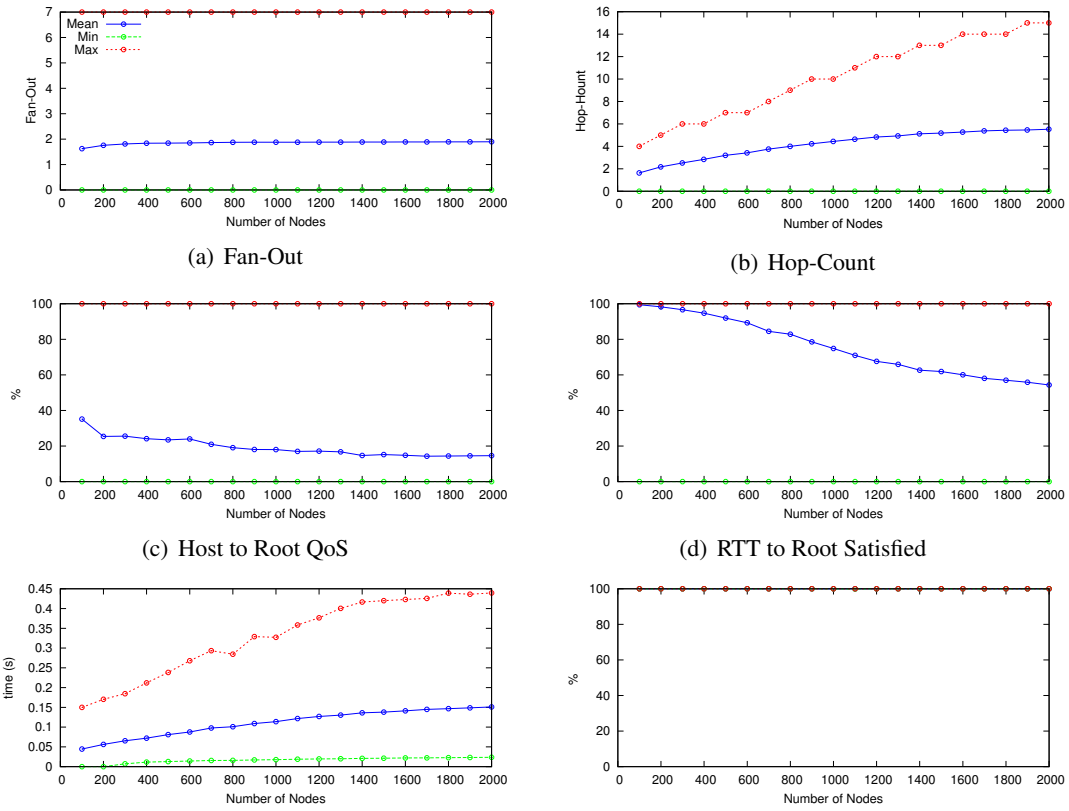
  If we have a look at Fig. 5.1(e) we can see that even though the Fan-Out has been limited and du to an increased Hop-Count, we still have fairly acceptable delays to the root. The maximum is a bit off but still always under half a second without any RTT to Root optimizations. If we compare the shape of the curve with the one of the Hop-Count, we can see that the same argumentation applies here as well. The avarage begins with 50ms at a 100 hosts which is also consitant with the Hop-Count, and ends with 150 ms. This could be confusing because one would think in this case the avarage of the RTT to Root satisfied would have to be higher as well since we set a maximum of 200ms. But, it seems that many hosts are only a little bit over this threshold, while many other hosts are way below it. The minimum is of course very low because it is given by the lower values in our RTT matrix (Table 3.1 for a Hop-Count of 1).

- **Total percentage of received multicast messages**

  Figure. 5.1(f) shows how many of the sent multicast messages were actually received by the hosts. As we can see, due to our optimizations (finger notification, optimized robustness), we could actually manage not to lose any message. We cannot state that we could guarantee this because there could be very special situations where pointers are wrong. This is due to delays that are naturally a problem in asynchronus networks. But, this is still very acceptable because even the native IP-Multicast is not 100% reliable.

5.3 Optimized Chord using Sender Driven Multicast with integrated QoS support

Figure. 5.2 shows the results of Chord using the same parameters as used in Section 5.2 but with the QoS module enabled. This means that unless the QoS support interferes with the network, the statistics should more or less be the same except for the QoS evaluations. And, if we compare the two scenarios we indeed have no significant divergences. Though with QoS support, the root has as previously mentioned to be the host with the lowest ID leading to a regular root transfer, expecially in the beginning.



(a) Fan-Out

(b) Hop-Count

(c) Host to Root QoS

(d) RTT to Root Satisfied

(e) Host to Root RTT

(f) Total percentage of received multicast messages

Figure 5.1: Chord - No QoS - Sender Driven Multicast

So, we will only go into the details of Fig. 5.2(c). As expected, the QoS property is fulfilled in any case, even if the pointers are not accurate, because all the hosts are sorted clockwise and the multicast messages are also sent in this direction. This might be different in a real network scenario because there, we expect QoS classes to change. Also, the host may not even recognize that before it is too late. Or at least, some messages will not be sent because its bandwith limit is reached, or they will be delayed or jitter would be generated depending on the QoS property that has been violated. But, also in a real scenario, the host could quickly rejoin at a position that matches its changed capability. There would still be a chance that while changing its position and notifying its neibours, the host would not be able to handle the traffic as it should. So, of course in a real network, it would not look as perfect as it does here. But, it is still a very reliable QoS machanism assumed that the QoS class can be measured properly and does not change all the time. We did tests where hosts got a message that their QoS class is not right anymore, forcing them to rejoin. But, the stats looked more or less the same because hosts could immediatly react on the change and did not have to wait until they detected themself that they could not handle their QoS class anymore. During this time, many if not all the children of the host would have been affected having a greater influence on the statistics. We are very satisfied with the results since they show that QoS can be achieved using Chord without having a determinable disadvantage to Chord without QoS support.

5.4 Optimized Chord using Receiver Driven Multicast with integrated QoS support

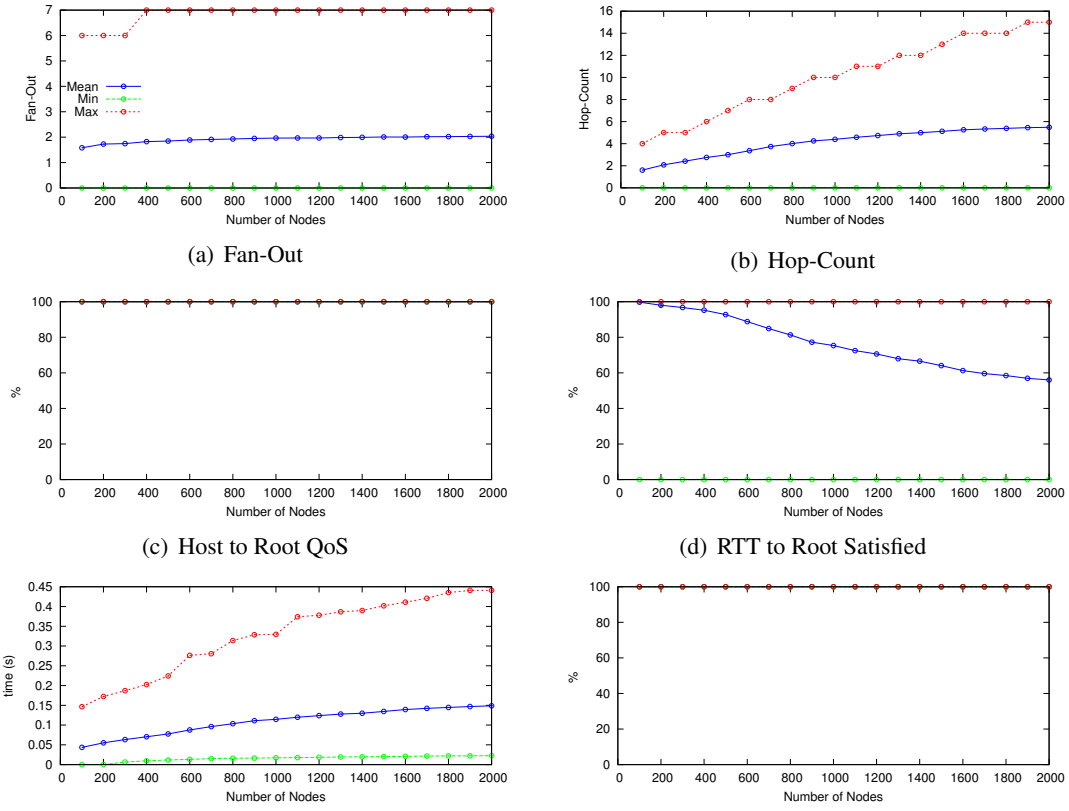
Figure 5.3 shows the results of Chord using Receiver Driven Multicast, which has been described in Section 3.3. The parameters again are the same as in the two other scenarios. But of course, the results will not be as similar because of the completly different approach to maintain a multicast tree, which is being built upon the receiver's needs and not upon "random" fingers.

So, what can we expect from this approach? Of course, we would like to see a higher percentage of hosts that have their RTT to Root requirements fullfilled. We would also suspect the total percentage of received multicast messages to drop since there is a more complex structured to be maintained. And we would suspect that the avarage RTT to Root drops as well since the multicast tree is now organized by the hosts Rtt to Root requirements.

1. **Fan-Out**

   If we have a look at Fig 5.3(a) we can see that the limited Fan-Out in our Receiver Driven Multicast distribution is more or less equivilant to what we have seen in the Sender Driven aproach. This is also what we would have expected since first, the Fan-Out is limited to the same number, and second, the search mechanism for the children is similiar to the one of fingers but in an inverse direction. Fingers are being found 2^i IDs away, wheras children search a parent at $theirChordID/2$. If this does not fullfill their requirements, they search at $theirChordID/2/2$ resulting in a logarithmic abbreviation of Chord IDs compared to an exponential extention in the reverse direction.

2. **Hop-Count**



(a) Fan-Out

(b) Hop-Count

(c) Host to Root QoS

(d) RTT to Root Satisfied

(e) Host to Root RTT

(f) Total percentage of received multicast messages

Figure 5.2: Chord - QoS Support - Sender Driven Multicast

Looking at the Hop-Count in Fig. 5.3(b), it becomes clear that this multicast distribution must use a different approach. Especially in the beginning, where the Hop-Count is arround 4, it is much higher than the one of the Sender Driven approach. This can be explained due to the fact that a host tries to find a parent that is not too close to the root so that its RTT requirements are only fullfilld tightly. This is done because the root of course is the best parent for every host. So, for once it is better to have the longest distance from the root as possible in respect to the host's RTT requirements. This gives other hosts, which have even tighter requirements or that are physically far away from the root a chance to find a parent as well. This will lead to a higher Hop-Count at the beginning but as we can see, it stops at arround the same amount for 2000 hosts. Meaning, that with 2000 hosts, the trees early positions are very well saturated resulting in a pretty well balanced tree.

The maximum could also be reduced from 15.5 hops to 11. This is because having over 10 hops, the RTT to Root becomes too high for the upper limit of 200 ms since the mean RTT between two hosts is arround 24 ms.

3. **Node to Root QoS**

   Figure. 5.3(c) shows that the QoS property is always fulfilled. This is due to the fact that the hosts are always searching their parents backwards, meaning that they receive their messages always from a host with a lower ID (higher QoS class). This is exactly what our QoS definition dictates.

4. **RTT to Root Satisfied**

   Figure 5.3(d) shows if each multicast message has been received in time or not. This is measured at the maximum RTT to Root a host would like to have. As we can see, until 400 hosts, the messages always arrive in time. So, in all the cases, the host's requirements are fulfilled. After that, the average begins to drop slowly until 10% of the messages do not arrive in time with 2000 hosts. This is fairly good considering that the Fan-Out is limited and that the requirements are, with a maximum RTT to Root range from 100ms to 200ms, pretty tight.

   Also, what we have to take into account is that our intention was to develop a mechanism that could be used by almost any P2P network. So, we used a pretty simple approach that does not need a very complex structure to be maintained but unfortunately also does not clusterize hosts that are physically close before it optimizes their RTT to Root.

   With other words, if we look at a path from the root to a leaf, it may very well be that between two hosts are located in Europe there is a hop containing a host from the USA. This would happen if:

   (a) the root is from Europe.

   (b) a joining American host does not find a parent under the root's children that would satisfy its requirements so it joins the root directly, and

   (c) a european host joins that does not have a very tight RTT to Root requirement hence, it can use the American host as its parent.

This is a worst case scenario that does not happen very often. But it shows that a path from the root to the leaf is not built upon the physical proximity of all the hosts in that branch but only for one host's RTT to Root requirements. It would be better if the tree was clustered: e.g., a branch for each continent then subbranches for each region in that continent and so on. But, this would of course need a much more complex structure like NICE (see Section 2.3.4, which:

(a) may have more problems working with different P2P networks because its complex structure would require the network to provide very specific information (e.g., provide a selection of hosts in a certain QoS class / in a certain region).

(b) requires a lot of communiation traffic.

(c) has no Fan-Out limit and would become even more complex if, e.g., delegated cluster leaders would be used.

(d) has a multicast tree that needs to be rearranged more often.

Mainly because of the latter, this would also lead to a lower percentage of received multicast messages, since messages can get lost during rearrangements. Furthermore, we can also say that it would be hard to find an application where 2000 hosts are in need of a RTT between 100 to 200 ms.

A video conference with so many people does not really make sense, leaving only a scenario like live broadcast stream, which is not very delay critical or a sensor network that relies on accurate data, or very large massive multiplayer games. Considering games, it would actually be possible that 2000 players are managed by the same P2P network. Hence, it would make sense to perform a hierarchy for different RTT requirements. For example, the communication between players that are in the same area would be done by a network that has a maximum RTT of 100 ms, wheras the communication between the different areas would be done by an additional network but with a maximum of 200ms or even higher. So, the setting of the maximum RTT to Root parameter really depends on the application. Of course, if it is set too low with too many hosts, then there is simply no possibility that all hosts can be served in time. This clarifies that the choice of this parameter and the organization of the application are the main factors in order to succeed in building a network that supports RTT guarantees.

If we have a look at the minimum of the graph in Fig. 5.3(d), we can see that it is dropping to 0 for 500 hosts. This means that at least one host has its RTT requirements never fullfilled. This is the case if:

(a) there are already many hosts in the network.

(b) the joining host has a very high QoS class.

(c) the root and its children already have their Fan-Out limit reached.

In such a scenario, the joining host simply has no other host to attach itself to. In case of direct successors of the root, the root actually deletes other children in order to make place for them because the root is the only possible parent they can use if QoS is enabled. But,

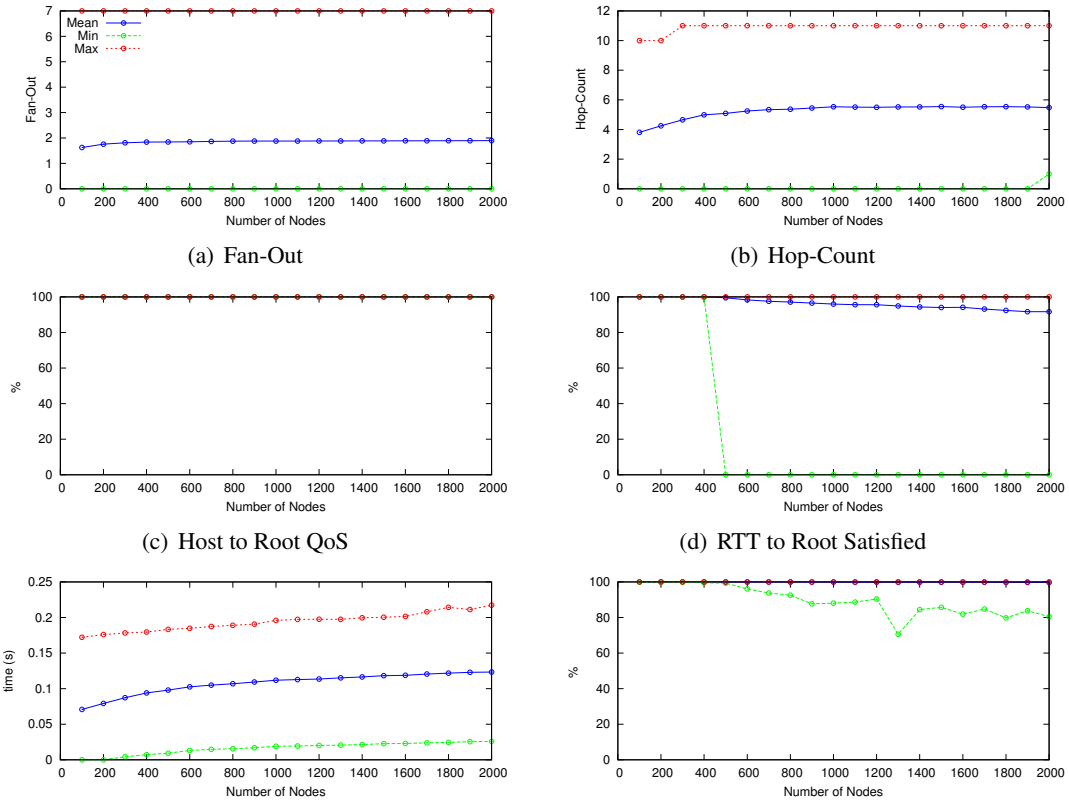
for hosts that are further away, this has not been implemented, since the potential parent can not determine if it is the only possible anchor. Again, a more complex structure that is aware of the timinigs between the children and is more involved with the underliying network could handle those hosts as well. But, as said before, we want to keep it as simple as possible, hence this is not an option.

5. **Node to Root RTT**

   The curves in Fig. 5.3(e) compared to the previous statistics about the RTT to Root shown in Figures 5.1(e) and 5.2(e) are not much better. This is because of the same facts, we already mentioned before. The hosts try to find a parent that can barely fulfill their RTT to Root requirements. For the maximum though, we can see that it only exceeds the 200ms mark by 20ms compared to the 250ms in the optimzed Sender Driven case. What is important is that the curves stagnate with the amount of hosts being in the network, meaning its scalable.

6. **Total percentage of received multicast messages**

   In Fig. 5.3(f), we see that although usually all hosts receive all messages, there is at least one host from a network with at least 400 hosts that does not receive every message. This can also be explained as we did previously. If a host is unable to find a parent that satisfies its RTT requirements in time (we used 6 tries), it searches itself another parent to at least be able to receive multicast traffic even though it will not arrive on time. But, depending on the application, the host may be able to use the incoming data anyway. After a short time period (10s), it will try to find a better parent again. During the time a newly joined host searches for a parent, supporting its maximum RTT or not, it will of course not receive any multicast message which is the reason why the minimum drops to 80%.



(a) Fan-Out

(b) Hop-Count

(c) Host to Root QoS

(d) RTT to Root Satisfied

(e) Host to Root RTT

(f) Total percentage of received multicast messages

Figure 5.3: Chord - QoS Support - Receiver Driven Multicast

Chapter 6

Conclusion and Outlook

As we mentioned in the introduction, the goal of this masterthesis is to provide an application that is able to support multicasting, QoS and RTT to root guarantees independently of the underlying physical network. The motivation for doing this is that the requirements for networks, especially the Internet, in terms of bandwidth, delays and other QoS properties are constantly growing mainly because of all the new real time media. One way to cope with that is to construct faster Internet lines, introduce more complex routers, etc. But, this is very expensive and also is constraint by contracts between providers, billing issues and other political reasons. This makes it obvious that it is necessary to look for technologies and structures that can use the existent network infrastructure more efficiently.

It has been shown in the introduction and Chapter 2 that we cannot use IP Multicast because it is not supported Internet wide. But, we can achieve some sort of multicasting using an application to maintain connections to other hosts and to distribute multicast messages. We have also seen that even though ALM is not as efficient as native IP Multicast with the client-server model it is still much more efficient than using simple unicast connections.

As we have mentioned previously, P2P protocols are a very good choice to provide such an application based multicast service. They are also less error prone because of their decentralized structure in contrast to the classical client-server model.

Furthermore, we have seen that it is a necessity for modern networks to support QoS, since todays multimedia applications demand hosts to have reliable connections to the sender in order to get a good quality of experience. Unfortunately, QoS is not supported Internet wide due to similar reasons as for IP multicasting apply. But fortunately, there are P2P networks like Chord that can support QoS on an application layer as we have shown in Section 3.2.5.

We have also shown that even though a P2P network can support QoS, it mostly still lacks support for all the properties that rely on a multi hop path from the sender to the receiver because P2P networks are often not aware of the physical proximity between their peers. To realize a service that could provide RTT to root guarantees, we would either need to change our P2P network in a way that allows each peer to search its parent itself, depending on its requirements. Or we can add an additional overlay that could provide this service to our P2P network. We decided to use the latter because it is the more modular approach since it can be used with many different P2P networks.

We found a completely decentralized, structured, P2P network called Chord that supports most of the demanded properties. Chord is organized in a ring structure and uses Distributed Hash Tables (DHT) to place new hosts in this ring. The way this works is that a joining host will build a hash value over its unique identifier (e.g., its IP address), which will give it its position on the ring. Each data will also get a hash value based on its unique identifier (e.g., its file name) and will then be placed on the peer with the same hash value or the peer next to it if no such peer exists. This structure enables Chord to root messages directly to its destination instead of having to use some sort of flooding mechanism as it would be the case in an unstructured network like Kazaa [19]. This makes Chord very scalable, which is also something that will be very important in future networks, since the number of network devices attached to the Internet is growing fast. We also had to introduce optimizations that make Chord faster and more robust in order to get it to work with all our requirements.

In addition to using Chord, we developed a custom P2P network on top of Chord that uses a receiver driven multicast mechanism to provide RTT to root guarantees to our Chord network and potentially also to other P2P networks. This mechanism was not easy to develop because we wanted it to be as simple as possible. Since the more complex a network algorithm gets (also on an application level), the more error prone it is and also the more communication traffic is caused, restricting the flexibility of a network.

Now, looking at the results, we can say that we succeeded in implementing Chord in the Omnet++ simulator, supporting multicast, QoS and RTT to root guarantees.

Furthermore, thanks to our optimizations, we are able to provide a Sender Driven Multicast mechanism that does not produce duplicates nor misses any receivers. We were also able to implement QoS as proposed by the inventors of Chord, having also no trouble in dealing with many different QoS classes.

Last, we succeeded in implementing an additional overlay that is able to provide RTT to root guarantees. There are situations though where this framework is not able to guarantee every host's RTT to Root requirements. This happens if the requirements for the RTT to the root are too tight compared to the size of the network. This means that the network may have to be split in some sort of hierarchy with consecutively decreasing RTT to root requirements from the root to the leafs. This would allow the hosts that are in close proximity to each other to exchange important information very quickly whereas hosts far away from each other would share less important information that may not be so delay critical. We have seen that if we for example would use NICE for this framework, we might get better results this is due to the fact that the multicast tree is better balanced since the min. and max. cluster size constrains forces it to restructure and rebalance the tree. Also NICE is proximity aware allowing a host to find its parent faster. But, as said previously NICE is also very complex resulting in less error resilience. Also, because of this complexity it would not be optimal for a framework implementation since it may not be as flexible as required.

What has taken a lot of time while realizing this masterthesis were the optimizations. These were not planned from the beginning. But, as we implemented more and more functionality, it became clear that we had to do some sort of improvements in order to have stable simulations. The main problem in the beginning was that the hosts joined / left too fast after each other so

that the stabilization protocol did often not run as quickly as it should have, leaving gaps in the Chord ring. Of course, we could have decreased the period until the next stabilization cycle runs, but this would lead to a lot of unnecessary communication traffic. Furthermore, we could have increased the number of successor, which would also lead to an increase in communication traffic and also to an increase in memory consumption. Because more communication traffic and more memory consumption means more time to be spent on simulations, we decided to do optimizations whenever we encountered an unfinished simulation. The time spent on these improvements is worth it though since they produce a much better percentage of received multicast messages. They also reduce the overall communication overhead and reduce the number of successors needed. In most cases, they even make the stabilization protocol obsolete or at least decrease the stabilization period saving a lot of communication traffic.

To summarize, we are content with our results even though we see room for improvements as will be discussed shortly. We achieved an RTT to root guarantee service that is very reliable for a low number of nodes (up to 400 hosts: 100% received multicast messages, 100% RTT to Root requirements fulfilled). Even with 2000 hosts, there are still 90% of the hosts that receive their messages in time. For the other hosts, it is not as bad as it used to be without such a framwork, because the maximum RTT decreased from 450ms, which would not allow any real time application, to 200ms, which would still allow many real time applications with some limitations.

Now at last, we will show what could be even further improved and what aspects have not been tested yet. One of the main concerns we have is that a test on a real network (Internet) may show very different results. This is because our Omnet++ implementation does not model congestion, no human errors, no electromagnetic radiation and also no background traffic flowing over the same connections that may influence the simulation. A test like this is out of the scope of this masterthesis and also a test in real network would be different each time it runs since a real network's load is varying all the time. This would lead to unreproducible results, which in this case also would not result in much significance.

Also, what we have to keep in mind is that even though we can try to reorganize our multicast tree, it is still depending on the underlying network. Especially for RTT to root guarantees, this means that we cannot actually guarantee them because the underlying network may be too slow or used to capacity making guarantees simply impossible. In situations like this, we could reduce the hosts QoS class / RTT to root requirements to a certain point where its connection is simply too bad to maintain any proper real time service. So, in the end even though we try as hard as possible to overcome the limitations of the physical network, we still need to rely on it. The only thing we can do is to use its capabilities as best as we can (best effort) or we could use QoS reservation mechanisms i8f the underlaying network would provide those.

For future work, we could think of supplying our RTT to root framework with a hierarchy mechanism as has been stated previously. This hierarchy would be useful for very large and delay critical applications. For example this could be used for massive online multi player games where many players are in the same world but in different regions. Here, it would make sense that players near to each other have a low delay in order for example to see an animation in time whereas players being further away have a higher delay because they cannot directly interact with each other (maybe only with chat messages, which are not delay critical). Also, there may

be other P2P networks that are better suited for this task because they are more proximity aware. But, as mentioned previously, this would add more complexity and lower its flexibility which might prevent it from being usable by many P2P networks without having to modify them too much.

Glossary

ALM	Application Layer Multicast
BRITE	Boston university Representative Internet Topology gEnerator
CAN	Content Addressable Network
DHT	Distributed Hash Table
FIFO	First In, First Out
IANA	Internet Assigned Numbers Authority
IGMP	Internet Group Management Protocol
IP	Internet Protocol
NICE	Nice is the Internet Cooperative Environment
NL	Neighbor list
OM	Overlay Multicast
P2P	Peer-to-Peer
QoS	Quality of Service
RNG	Random Number Generator
RTT	Round Trip Time
TTL	Time-to-live

Bibliography

[1] S. Deering, "Host extensions for IP multicasting," RFC 1112 (Standard), Aug. 1989, updated by RFC 2236. [Online]. Available: http://www.ietf.org/rfc/rfc1112.txt

[2] M. Brogle, D. Milic, and T. Braun, "Quality of service for peer-to-peer based networked virtual environments," *Parallel and Distributed Systems, International Conference on*, vol. 0, pp. 847–852, Dec. 2008.

[3] M. Hosseini, D. T. Ahmed, S. Shirmohammadi, and N. D. Georganas, "A survey of application-layer multicast protocols," *Communications Surveys & Tutorials, IEEE*, vol. 9, no. 3, pp. 58–74, Sep. 2007. [Online]. Available: http://dx.doi.org/10.1109/COMST.2007.4317616

[4] R. Braden, D. Clark, and S. Shenker, "Integrated Services in the Internet Architecture: an Overview," RFC 1633 (Informational), Jun. 1994. [Online]. Available: http://www.ietf.org/rfc/rfc1633.txt

[5] M. Brogle, D. Milic, and T. Braun, "Qos enabled multicast for structured p2p networks," in *Consumer Communications and Networking Conference, 2007. CCNC 2007. 4th IEEE*, Las Vegas, NV, USA, Jan. 2007, pp. 991–995.

[6] M. Brogle, L. Bettosini, and T. Braun, "Quality of service for multicasting in content addressable networks," in *12th IFIP/IEEE International Conference on Management of Multimedia and Mobile Networks and Services (MMNS'09)*, Telecom Italia Future Centre, Venice, Italy, Aug. 2009.

[7] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, Aug. 2003.

[8] A. Varga, "The omnet++ discrete event simulation system," in *Proceedings of the European Simulation Multiconference*. Prague, Czech Republic: SCS – European Publishing House, Jun. 2001, pp. 319–324. [Online]. Available: http://www.omnetpp.org/

[9] Z. Albanna, K. Almeroth, D. Meyer, and M. Schipper, "IANA Guidelines for IPv4 Multicast Address Assignments," RFC 3171 (Best Current Practice), Aug. 2001. [Online]. Available: http://www.ietf.org/rfc/rfc3171.txt

[10] J. Postel, “Internet Protocol,” RFC 791 (Standard), Sep. 1981, updated by RFC 1349. [Online]. Available: http://www.ietf.org/rfc/rfc791.txt

[11] W. Fenner, “Internet Group Management Protocol, Version 2,” RFC 2236 (Proposed Standard), Nov. 1997, obsoleted by RFC 3376. [Online]. Available: http://www.ietf.org/rfc/rfc2236.txt

[12] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan, “Internet Group Management Protocol, Version 3,” RFC 3376 (Proposed Standard), Oct. 2002, updated by RFC 4604. [Online]. Available: http://www.ietf.org/rfc/rfc3376.txt

[13] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy, “Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web,” in *ACM Symposium on Theory of Computing*, El Paso, Texas, United States, May 1997, pp. 654–663. [Online]. Available: http://doi.acm.org/10.1145/258533.258660

[14] D. Eastlake 3rd and P. Jones, “US Secure Hash Algorithm 1 (SHA1),” RFC 3174 (Informational), Sep. 2001, updated by RFC 4634. [Online]. Available: http://www.ietf.org/rfc/rfc3174.txt

[15] 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*, vol. 32, no. 4. New York, NY, USA: ACM Press, Oct. 2002, pp. 205–217.

[16] S. Barthlomé, “Bachelor thesis: Quality of service for overlay multicast applied to the nice protocol,” University of Bern, Switzerland, Sep. 2009.

[17] A. Medina, A. Lakhina, I. Matta, and J. Byers, “Brite: An approach to universal topology generation,” in *MASCOTS ’01: Proceedings of the Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. Washington, DC, USA: IEEE Computer Society, Aug. 2001, p. 346.

[18] M. Matsumoto and T. Nishimura, “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator,” *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, Jan. 1998.

[19] N. Zennstroem and J. Friis, “Kazaa,” Amsterdam, Netherland, Mar. 2001.