QUALITY OF SERVICE FOR OVERLAY MULTICAST CONTENT ADDRESSABLE NETWORK (CAN)

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Luca Carlo Bettosini
2009

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

Acknowledgment

There are many people who helped me to realize the research I did, and that led to this Master thesis. First of all, I would like to thank my tutor Marc Brogle, who has always supported me and who has invested a lot of time in solving different kinds of problems. Secondly I would like to thank Sebastian Barthlome and Andreas Ruettimann for there creative ideas, their quick solution determination and for the many good talks we had over the P2P topic. Last I would like to thank Martin Sütterlin, my manager, for the time and support which he gave me during my studies.

Abstract

The multicast paradigm is one of the most efficient methods to distribute information to a group of receivers. The IP implementation of this paradigm, IP Multicast, is not widely spread in today's Internet, mostly due to political reasons. However, Application Level Multicast (ALM) or Overlay Multicast is an alternative solution to use multicast. One Overlay Multicast solution is the Content Addressable Network (CAN), which offers multicasting to the end-user. To provide Quality of Service (QoS) functionality to this network, we have introduced the Overlay Multicast QoS (OM-QoS) Framework, which enables QoS mechanisms for multicasting in ALMs. It is a self-managing framework, which can be used for different Overlay Multicast protocols. The evaluations for the Overlay Multicast QoS Framework with the underlying Peer-to-Peer (P2P) network CAN has shown that we can guarantee, that all paths in the multicast tree support specific QoS requirements. Furthermore, we have evaluated the Overlay Multicast QoS Framework with the NICE P2P network protocol. Both evaluations have shown a small overhead in terms of number of hops to the multicast tree's root, node to root RTT and fan-out.

Summary

Multicast is the delivery of information to a group of receivers. One of the first implementations of the multicast transport scheme in the Internet Protocol (IP) environment is IP Multicast. Instead of sending a datagram to each host individually using unicast, the datagram is sent to a specific IP Multicast address, called multicast group. However, to enable IP Multicast in a network, the infrastructure (e.g. routers) need to support the distribution of multicast packets. In today's Internet, most IP providers do not support IP Multicast due to political reasons.

Overlay Multicast, also called End System Multicast or Application Level Multicast (ALM) was proposed as an alternative implementation of the multicast paradigm to the IP Multicast implementation. This approach builds a virtual topology on top of the physical Internet to form an Overlay Network. Therefore, the IP layer provides a best-effort unicast datagram service, while the Overlay Network implements all the multicast functionality. The network topology of such an Overlay Multicast infrastructure is often built using a P2P network.

P2P networks are distributed network systems, which have no hierarchical organization and no centralized control. Therefore, the peers themselves form an Overlay Network on top of the IP network. In this thesis, we focus on structured P2P networks, which means in a technical manner that the overlay topology is tightly controlled. Content is not placed randomly but at specific locations within the network.

The Content Addressable Network (CAN) is such a structured P2P system. It is designed to be scalable, fault-tolerant and self organizing. It's architecture is based on a multi-dimensional Cartesian coordinate space on a multi-torus, which is partitioned among all peers currently in the network. Routing within CAN is performed according to neighbor tables, meaning that a message is routed towards the destination with a simple greedy algorithm, which forwards a message to the neighbor closest to the destination.

The goal of this thesis is to enable Quality of Service (QoS) for P2P protocols, which do not have a native QoS support (e.g. CAN). The term Quality of Service refers to the capability of a network to provide better service to selected traffic using various technologies. Therefore, it can be used to guarantee a certain level of performance to a data flow. We use QoS classes as a generalization for the different QoS requirements of users. These QoS classes have to fulfill certain properties in order to ensure that we can order them. With this order, we can establish a QoS tree, which has the node with the highest QoS class as root and each end-to-end path from the root to a leaf node has to have monotonically decreasing QoS requirements.

We have built a framework, which is independent from the underlying P2P network and which provides QoS support for the underlying P2P network by setting up the network to form such a

QoS aware tree. We have designed the framework in such way, that it intercepts messages from the underlying P2P network, and therefore only a few adaptions have to be made within the P2P network itself. The basic network design of the framework can be described as a chain structure. For every QoS class, a separate P2P network entity is built. These separate P2P entities are connected to each other over gateway nodes. The gateway node is a designated node, which is normally the root in such a network and which is responsible for establishing a link to the next lower QoS class represented by another P2P entity.

To further optimize our Overlay Multicast QoS Framework, we introduced backup links which accelerates the search for a takeover node within the leave process of a gateway node and which reduces packet loss. Another improvement of the framework was the hop optimization. If we have n different QoS classes, the nodes which are in the lowest QoS class would receive a multicast message after at least n hops. This leads to a large node to root RTT and could cause problems for certain multicast applications. The hop optimization establishes additional gateway links over the existing chain topology to reduce the high hop count by at least a factor of two.

Our evaluations of the native CAN framework have shown that the hop count improved and therefore also did the node to root RTT and the fan-out if the nodes rejoined the network after having stayed in the network for a certain period. Furthermore, if we increase the number of dimension for the CanKeySpaces, the fan-out rises drastically.

The evaluations of our Overlay Multicast QoS Framework have shown that we can guarantee, that all paths in the multicast tree support certain QoS requirements. We evaluated the framework with CAN and NICE as underlying P2P networks and we saw only a slight overhead in terms of node to root RTT, hop count and fan-out.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The importance of fast and efficient delivery of information has grown more and more over the past years. In particular, the efficient delivery over the Internet has gained focus, as the number of Internet users is growing every day while the network capacity is only slowly increasing. One of the most efficient delivery methods is the multicast paradigm, which is almost as old as the Internet itself. The multicast paradigm allows delivery of information to a group of interested receivers. IP Multicast is one implementation of the multicast paradigm, which specifies the delivery of packets in an Internet Protocol environment. Altough research in the area of IP Multicast has made significant progress, the protocol was never widely deployed in the Internet. There are multiple reasons why the wide deployment never took place, the most important problem was the lack of support from the Internet Service Providers (ISP) due to billing and security issues. As a result of the non-acceptance of IP Multicast, the Overlay Multicast approach was developed. The main difference between those approaches is that the Overlay Multicast handles multicast messages on the end-systems and not in the network. Multicast traffic is disseminated using unicast connections and therefore ISPs do not need to support IP Multicast.

Another approach to reduce network traffic and to distribute data in an efficient way is the Peer-to-Peer (P2P) approach. This approach tries to solve a major problem of the common client-server paradigm. To increase the number of concurrent access rates at a particular server, the network capacity at the server side needs to be increased as well, or new mirrors of the server have to be installed. Both solutions imply further investment in the infrastructure used. The P2P approach on the other hand distributes the data across the P2P network, which is build among the receivers. Peers in the P2P network act both as client and server at the same time. This eliminates the central instance (server) and therefore reduces the investment needs. P2P mechanisms are often used as a underlying infrastructure to enable Overlay Multicast.

A further problem is that current P2P systems do not take different requirements and parameters (e.g. bandwith, jitter) of individual users into account. This leads to an inefficient usage of resources. To fulfill different requirements, the underlying network structure should be reordered. Quality of Service (QoS) mechanisms provide different priorities to different entities (user, data flows, applications). In networks, QoS could be interpreted as different service levels for different data flows, which can fulfill the different requirements of the users. Today QoS is only rarely supported in current available P2P implementations or in Overlay Multicast applications [1].

The goal of this thesis is to introduce a framework, which has the ability to provide Quality of Service, for any P2P network and therefore allows the usage of QoS within certain Overlay Multicast applications, which are based on those P2P networks. We have evaluated our approach by implementing Content Addressable Network (CAN) [2] within the Omnet++ network simulator [3]. Omnet++ is a public-source, component based discrete simulator, whose main purpose is to simulate communication systems. CAN was one of the first proposals for a P2P network, but as far as we know it was never implemented. Our implementation demonstrates the high scalability and error resilience of this approach but on the other hand also shows the limits of CAN (e.g. high fan-out, high complexity, duplicates for multicast transmission). For our implementation, we have used a modular approach to enable an easy transport of the produced code into a real world application.

CAN does not natively support Quality of Service. To enable QoS for CAN, we have developed a framework, which creates a dedicated P2P network for every QoS class and links those dedicated P2P networks together. The Overlay Multicast QoS Framework is built in such way that its use is transparent to the underlying topology and only demands few requirements from the underlying network. The framework can therefore be used for almost any P2P protocol. Aside from CAN, we also tested the framework with the P2P network NICE [4]. Our evaluation will show that our framework implementation is scalable, error resistant, highly portable and only generates a small overhead.

This thesis is structured as follows. In the next Chapter we will describe the terms Native Multicast and Overlay Multicast. Furthermore we will give an overview of the different Peer-to-Peer networks as well as introducing the term Quality of Service. In Chapter 3 we will describe the implementation of the CAN P2P network and the implementation of the Overlay Multicast QoS Framework, which enables the usage of QoS. In Chapter 4 we will present and explain the various results of our simulations. The last Chapter will conclude this thesis with a summary of the results as well as a short outlook to future work.

Chapter 2

Related Work and Background Theory

2.1 Multicast

For the delivery of information, three different routing schemes are known:

- Unicast is the delivery of information from one sender to one receiver. If multiple receivers need to be addressed, the information must be sent to each of them individually. An example of the unicast scheme is shown in Fig. 2.1.

- Broadcast is the delivery of information from one sender to all receivers in a network. Therefore, the information has to be sent only once to reach all receivers. An example of the broadcast scheme is shown in Fig. 2.2.

- Multicast is the delivery of information to a group of receivers. In this scheme the information is only sent once and will reach all receivers that joined the multicast group, on which the information was sent. An example of the multicast schemes is shown in Fig. 2.3.

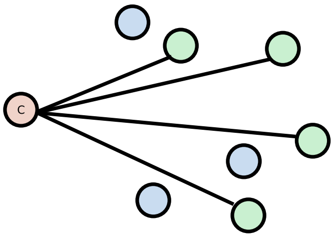


Figure 2.1: Unicast Scheme

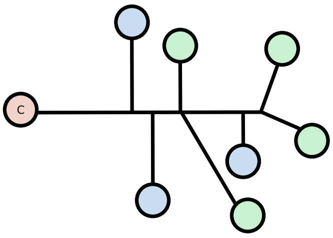


Figure 2.2: Broadcast Scheme

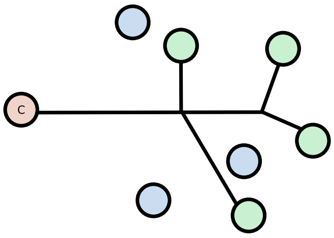


Figure 2.3: Multicast Scheme

To distribute information to multiple receivers, multicast is the most efficient scheme of those three, because the sender only needs to send information once. One of the first implementations of the multicast transport scheme in the Internet Protocol (IP) environment [5] is IP Multicast [6].

2.1.1 IP Multicast

IP Multicast is an extension of the Internet Protocol to support multicasting. Instead of sending the same datagram to each host individually using unicast, the datagram is sent to a specific IP address called multicast group. The reliability of such a multicast datagram is identical with the reliability of a normal unicast datagram, which means that no arrival of the datagram at the destination is guaranteed.

A host (sender or receiver) can join one or more multicast groups. The hosts which have joined a multicast group are called members of this group and they can leave this group at any time (dynamic membership). Furthermore, there is no restriction of how many users can join a multicast group nor where they are located in the network.

A multicast group is defined by a multicast IP address. In the Internet Protocol address space, the range from 224.0.0.0 to 239.255.255.255 is reserved by the Internet Assigned Numbers Authority (IANA) [7] for multicasting. These IP addresses are called class D IP addresses and they all have the high-order bits set to “1110“. There are two different types of multicast groups. First, there are the permanent groups, which have fixed assigned IP addresses to each group but this does not imply fixed membership. For example, the multicast group 224.0.0.1 is reserved for the permanent group of all IP hosts [6]. These permanent groups are mostly used for administrative purposes and can have no members. A list of those predefined groups can be found on the IANA website. The second group type is called transparent groups, because they only exists if they have at least one active group member. This is the normal group type for multicast sessions.

To transmit multicast information, specific routers with multicast abilities handle the forwarding of IP Multicast datagrams between different networks. Inside a local network, the sender transmits the IP Multicast datagrams directly to all members of this network. The hosts, which joined the sent multicast group will receive and process the packet while the remaining hosts in the local network will just ignore the packet. Depending on the time-to-live (TTL) [5] of the datagram, the multicast router attached to the local network will forward the datagram to other connected networks, which are interested in receiving multicast traffic from this multicast group.

In Fig. 2.4, the sender transmits a multicast datagram to a specific multicast group, where all hosts with the green color have joined this specific multicast group. The first router forwards the packet further into the network as there are multiple hosts, that want to receive multicast messages from this multicast group, attached to the network. The router does not duplicate the packet, because the red host has not joined this multicast group. The next two routers receive the packet through the network and they both duplicate the received datagram and send one to each network with joined group members.

The Internet Group Management Protocol (IGMP) [6] is used by multicast routers to communicate with attached peers. It allows the router to get an overview of which attached network is interested in receiving datagrams from which multicast group. The protocol is only used for communication between multicast routers and end-systems and not between multicast routers. At the moment three versions of IGMP exist: IGMPv1 [6], IGMPv2 [8] and IGMPv3 [9]. They are backward compatible.

In IGMPv1 [6], the router periodically sends a host membership query to all attached networks.

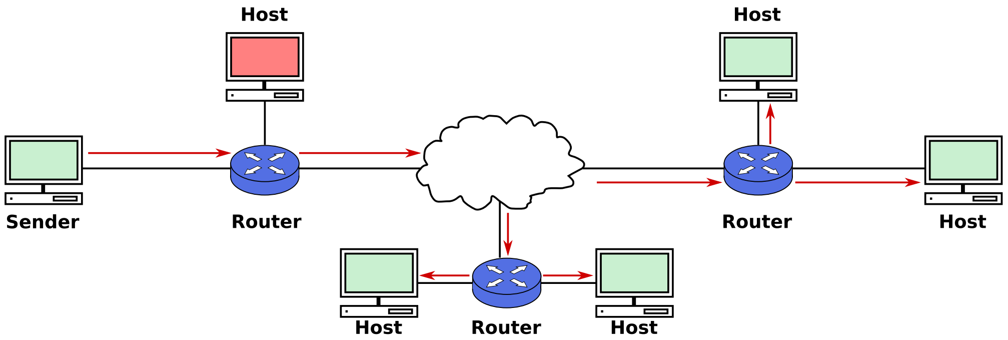


Figure 2.4: IP Multicast Example Scenario

These queries are sent to the all-host group address (224.0.0.1) with a TTL of 1. This ensures that they only reach hosts, which are directly attached to the network. All hosts, which have received this query will respond with a host membership report for every interested multicast group. After receiving a host membership report for a multicast group from a network, the router starts forwarding multicast messages from this group to that network. The router will continue to send multicast messages to that network until it receives no further host membership reports answering its host membership queries from the network for this multicast group. This is called an implicit leave.

With IGMPv2 [8], the hosts have the ability to send LEAVE messages back to the router when they want to leave a specific multicast group, which reduces administrative network traffic. To further reduce this traffic, the multicast router can send two kinds of membership queries: a general query (like in IGMPv1 [6]) to learn which groups have members in the attached networks, and a group-specific query to learn, if a group has any members left in an attached network.

In the latest version of IGMP (IGMPv3 [9]), the support for ”source filtering” was added. This allows hosts to only receive multicast messages from a specific source.

Despite the fact that IP Multicast is a good implementation of the multicast paradigm, it was never widely deployed in today’s Internet. ISPs refuse to deploy it due to security and billing reasons.

2.1.2 Overlay Multicast

Overlay Multicast [10], also called end-system multicast or Application Level Multicast (ALM), was proposed as an alternative implementation of the multicast paradigm to the IP Multicast implementation. This approach builds a virtual topology on top of the physical Internet to form an Overlay Network. Each link in the virtual topology is a unicast tunnel in the physical network. Therefore the IP layer provides a best-effort unicast datagram service, while the Overlay Network implements all the multicast functionality such as dynamic membership maintenance, packet duplication and multicast routing.

Figure 2.5 shows a scenario, where a sender transmits a multicast packet to a multicast group in the ALM (all hosts with the green color have joined this multicast group). The Overlay Multicast application at the sender site forwards the multicast packet using unicast to the next

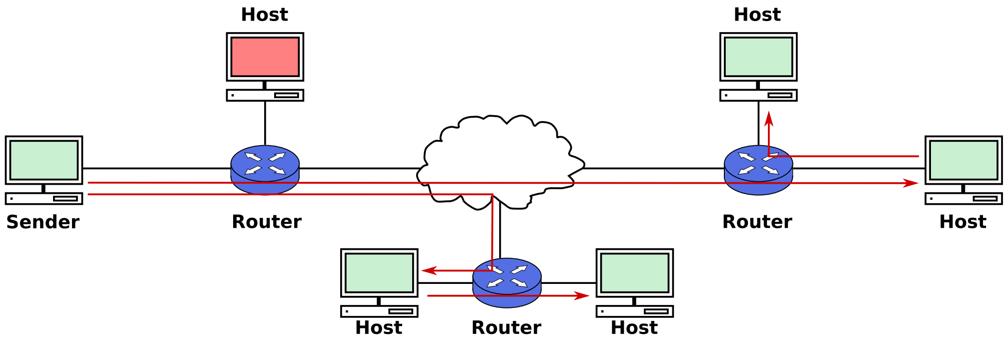


Figure 2.5: Overlay Multicast Example Scenario

hosts, which want to receive traffic from this multicast group. There, the multicast packet is duplicated and forwarded by the Overlay Multicast application to the next interested hosts. The network topology of an Overlay Multicast infrastructure is often built using a P2P network. However, the Overlay Multicast is less efficient than the IP Multicast as the multicast packets are duplicated at the end systems. Furthermore, Overlay Multicast uses the same link to the end-systems multiple times, which increases the network load. Nevertheless, it provides a solution to enable multicasting in today's Internet.

2.2 Peer-to-Peer (P2P) Networks

2.2.1 General Issues

P2P networks are distributed network systems, which have no hierarchical organization and no centralized control. Therefore, the peers themselves form an Overlay Network on top of the IP network. P2P networks offer a mix of features such as wide-area routing architecture, permanence, hierarchical naming, efficient search of data items, selection of nearby peers, redundant storage, trust and authentication, massive scalability, anonymity, and fault tolerance [11]. These networks are more than just an alternative to the common client-server model. They can offer services such as resource sharing. Furthermore, they need to manage the dynamic behavior of the peers, meaning the discovery of new peers as well as finding optimized routes through the network.

P2P networks can be divided into two kinds of networks: structured and unstructured P2P networks. The difference between these P2P networks is that in a technical meaning, the overlay topology of structured P2P networks is tightly controlled. This means that the content is not placed randomly but at specified locations within the network. This makes routing of queries through the network very efficient. Most of these networks use Distributed Hash Table (DHT) [12] as a substrate. Otherwise, unstructured P2P networks have no controlled topology. Content is placed at random locations, and therefore, queries have to be flooded through the network to find it. In this thesis, we focus on structured P2P networks such as CAN and NICE.

2.2.2 Distributed Hash Table (DHT)

In structured P2P networks, the location information of a data object is placed deterministically at a peer, whose identifier is equal with the data object's unique key. DHT based systems allow the assignment of uniform random IDs from a large identifier space to a set of peers. Therefore, DHT systems can be used to assign node-IDs (identifiers) to these peers as well as to assign unique keys to data objects. The node-IDs as well as the unique keys for the data are chosen from the same identifier space.

In a P2P network, data keys are mapped to a node, which currently is a member of the network. Data can therefore be stored in the network with the store operation *put(key,value)*, and can be promted with the retrieval operation *value = get(key)*. However, the retrieval operation requires routing the request from the requesting peer to the peer, which stores the requested key. Normally, a peer needs to maintain a small routing table where it stores the node-IDs and the IP addresses of the neighboring peers in the Overlay Network. This routing table enables forwarding of messages in the Overlay Network.

2.2.3 CAN

The Content Addressable Network CAN is one of the first structured P2P systems [2]. It is designed to be scalable, fault-tolerant and self-organizing. CAN's architecture is based on a multi-dimensional Cartesian coordinate space on a multi-torus. This d-dimensional coordinate space is partitioned among all current peers, as shown in Fig. 2.6. Therefore, every peer in the network has its own distinct zone within the overall space for which it is responsible. Furthermore, every node maintains a routing table containing the node-ID and the keyspace of its direct neighbors. In Fig. 2.7, the neighbor table of node G is shown. The nodes A, B, D and K are direct neighbors of G. Also node C is a direct neighbor of G, because of CANs architecture, which is a multi-torus coordinate space.

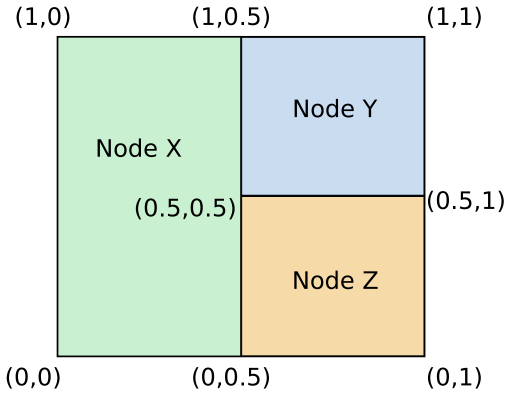


Figure 2.6: 2-Dimensional CAN Keyspace

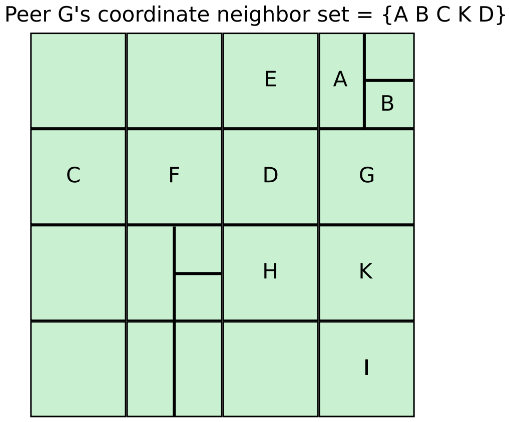


Figure 2.7: CAN Neighbor Table

Routing within CAN is performed according to the neighbor tables. A message is routed towards the destination with a simple greedy algorithm, which forwards a message to the neighbor that is closest to the destination coordinates. As shown in Fig. 2.8, which we have adapted from [2], a message is forwarded from node D to node I.

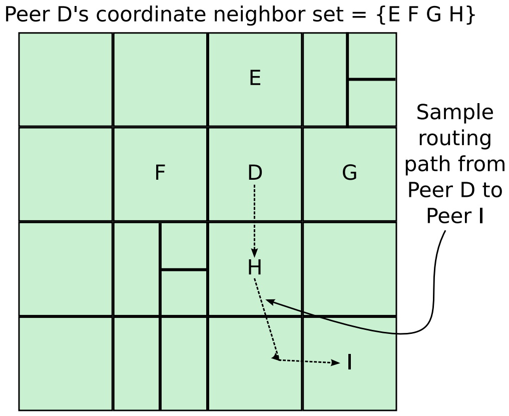

Peer D's coordinate neighbor set = {E F G H}
E
F
D
G
H
I
Sample routing path from Peer D to Peer I

Figure 2.8: CAN Routing Path

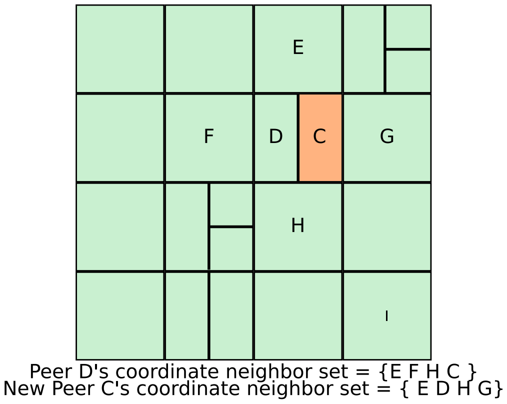

E
F
D
C
G
H
I
Peer D's coordinate neighbor set = {E F H C }
New Peer C's coordinate neighbor set = { E D H G}

Figure 2.9: CAN Join Mechanism

In CAN {key,value}, pairs are stored within the virtual coordinate space. The key is deterministically mapped using a hash-function to a point in the coordinate space, and therefore, to a node, which is responsible for this coordinate space. To lockup a value, any peer can use the same deterministic hash-function to find the point in the coordinate and must only forward a look-up message to the respective node where the key to the value is stored.

If a new node wants to join the network, it must get its own portion of the coordinate space. Therefore, it sends a JOIN message to a random node within the network. This message contains a random point within the coordinate space. The JOIN message is then forwarded through the Overlay Network to the node, which is responsible for this point. After receiving the JOIN message, this node splits its current keyspace into two halves, assigns the one half to the new joining node and informs its neighborhood about the change. In Fig. 2.9, node C has joined the network and node D has split its keyspace. The split is always performed along the longest dimension of a node. This joining mechanism allows a uniform distribution among the coordinate space. To get an even better distribution among the coordinate space, a design improvement was introduced [2]. Once a node receives a JOIN message containing a point for which it is responsible, it compares the size of its keyspace with the size of the neighboring keyspaces, and if a neighbor has a larger keyspace, it forwards the JOIN message to this neighbor, which then splits its keyspace in half.

If a node wants to leave the network, it needs to seek a takeover node in its neighborhood, which will take over its keyspace. The takeover node is chosen in such way that the keyspace of the takeover node can be merged with the keyspace of the leaving node. If no takeover node is found with which the keyspace can be merged, the takeover node is chosen randomly from the neighborhood. This node will then be temporarily responsible for two keyspaces. Once a takeover node has received all information from the leaving peer, it informs its old and new neighbors about the change.

CAN is fault-tolerant, because if a node fails within the network, messages can be routed around the failure. Every node sends a periodic UPDATE message to its neighbors. If such an UPDATE message is not received in a given period, the neighbor assumes that the node suddenly left the network and initiates a takeover timer with a random time. The first neighbor of the failing node whose timer expires will then take over the keyspace from the failed node and informs the other neighbors about the takeover. Unfortunately, such a failure causes all stored keys at the failed node to be lost. To prevent this loss, multiple realities could be introduced. This means that multiple Overlay Networks containing the same data could be established. As every node would have different keyspaces in different realities, the keys would be stored on multiple nodes, and therefore, a single node failure would not cause any data loss.

2.2.4 Chord

Chord [13] was introduced in 2003 and was proposed as a scalable P2P look-up protocol for Internet applications. Chord uses consistent hashing to assign identifiers to its peers and its data keys. Therefore, it can be classified as a structured P2P network. Chord has a ring based design, called a Chord ring, where identifiers are represented in a circle, ordered clockwise from 0 to $2^{(m-1)}$ (m defines how many bits an identifier has). However, the size of identifiers must be large enough to avoid hashing multiple keys to the same identifier. Peers can join a Chord network with little effort. This means that they only need to find a successor node in the existing Chord network in order to be considered as a joined peer in this network. They receive such a successor from a bootstrap peer, which maintains a partial list of Chord nodes. On the other hand, nodes can leave the network at any time without any interruption.

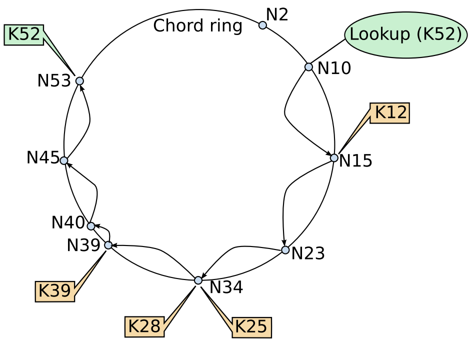


Figure 2.10: Chord Look-up Example

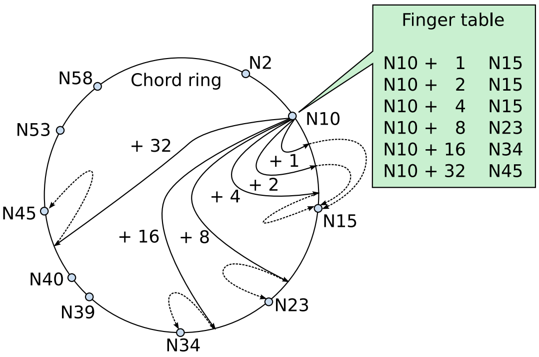


Figure 2.11: Chord Finger Table

Figure 2.10 (adapted from [13]) shows a Chord ring, which has nine active nodes and 5 keys in the network. The hash function assigned unique identifiers to the keys and to the peers, which define their positions in the chord ring. Each node is responsible for the keys with an identifier between the chord-ID of the node's predecessor and its own identifier. In our example, node 53 is responsible for key 52. Each peer in the network knows its successor peer on the Chord ring. Therefore, look-up requests can be sent around the identifier space until the node responsible for the searched key is reached. In our example, node 10 sends a look-up request for key 52. This request is forwarded through the network to node 53, which is responsible for this key. Node 53 will then return the data of this key to node 10.

Due to the fact, that passing around look-up requests is inefficient ($O(n)$), chord nodes maintain a finger table with up to m entries. The i^{th} entry of node s contains the identity of the first peer that succeeds s by at least $2^i - 1$ $(1 \leq i \leq m)$ on the chord ring. In Fig. 2.11 (adapted from [13]) we show the finger table for node 10, which consists of 6 entries. The first three entries all point to node 15, as there is no other node between node 10 and 13 $(10 + 2^2 - 1)$. The last entry in the table is 42 $(10 + 2^5)$, because 74 $(10 + 2^6)$ would point again to node 10 (the chord ring consits of a range from 0 - 64 and 74 overlaps this range by 10). Using finger tables, routing is much more efficient and has a performance of $O(log(n))$.

2.2.5 NICE

NICE [4] (Nice is the Internet Cooperative Environment) was introduced in 2002. It is specifically designed for low bandwith multicast applications with a large receiver tree. However, NICE can also be used for large bandwith multicast applications. It is designed as an efficient, scalable, and distributed tree-building protocol, which does not require any information about its underlaying topology. NICE uses hierarchical clustering to build its topology. This hierarchical clustering supports a number of different data delivery trees with different properties. Therefore, the basic operations of NICE are creating and maintaining this hierarchy, as this implicitly defines the multicast delivery path.

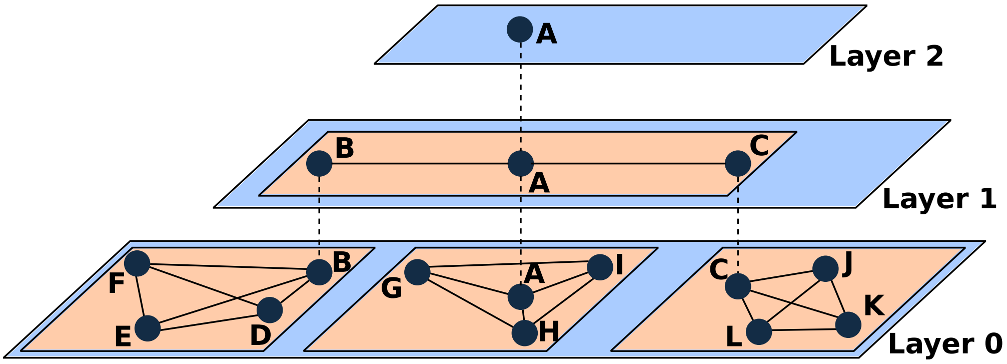


Figure 2.12: NICE Clustering Example

Figure 2.12 shows an example of the hierarchical structure of a NICE network. All nodes have joined the bottom layer 0 and have built three different topology clusters. The first one consists of members F, E, D, B and their cluster leader being B. The other two clusters on the layer 0 have the nodes A and C defined as their cluster leaders. These three cluster leaders form another cluster on layer 1. Furthermore, they define their own cluster leader A, which joins the next higher layer 2. This member hierarchy is crucial for the scalability of the NICE protocol, as most members are at the buttom of the hierarchy and only maintain a state about a constant number of other peers in the network.

As described in our example, NICE assigns nodes as members to different layers. All nodes are assigned to be members of layer 0. The nodes assigned to a layer form a cluster within this layer.

NICE defines the maximum cluster size and if a cluster expands over this size, it splits itself into two clusters. Furthermore, every cluster defines a cluster leader according to the minimum maximum distance to all other cluster peers. Therefore, the node in the graph-theoretical center of the cluster is appointed as cluster leader. All cluster leaders are then members of the next higher layer, where they join new clusters and define new cluster leaders. This process continues until only one node is remaining at the highest layer.

2.3 Quality of Service (QoS)

The term Quality of Service (QoS) refers to the capability of a network to provide better service to selected traffic over various technologies. The primary goal of QoS is to prioritize network traffic. Therefore it can be used to guarantee a certain level of performace to a data flow. For example, parameters such as bandwith, jitter, or maximum packet loss could be guaranteed. A best effort network or service does normally not support QoS [14].

In this thesis we introduce the concept of QoS classes as a generalization for the different requirements of the users. A QoS class is therefore a combination of different QoS parameters. As an example, a QoS class could be a combination of bandwidth and jitter. Whatever the combination of these parameters is, the QoS classes must have the following properties:

- A total order relation for all QoS classes must exist.
- The different paremeters of the QoS classes must be independent of link length and number of hops in the network.
- There is only a finite number of QoS classes.

With these properties, we can ensure that we can order the different QoS classes independent of their path length. This is a necessary prerequisite if we want to guarantee QoS requirements for all users in a multicast tree.

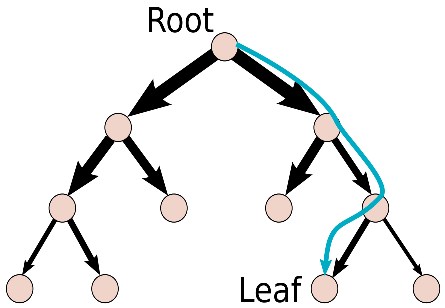


Figure 2.13: Overlay Multicast Quality of Service Tree Example

To ensure that the network provides QoS guarantees in a multicast tree, the tree has to be build according to the following rules:

- The node with the highest QoS class is the root of the multicast tree
- A child has to have a smaller or equal QoS class than its parent node

As shown in Fig. 2.13, each end-to-end path from the root to a leaf node in the multicast tree has to have monotonically decreasing QoS requirements [1].

The goal of this thesis is to build a framework, which is independent from the underlying P2P network and provides the QoS support for the underlying P2P network by building such a QoS aware tree.

Chapter 3

Quality of Service for Application Layer Multicast

3.1 Basic Omnet++ Implementation

3.1.1 Network Design

For our implementation of the P2P protocols and the Overlay Multicast QoS Framework, we used the Omnet++ simulator [3]. Omnet++ is a public-source, component based discrete simulator, whose main purpose is to simulate communication systems.

Omnet++ allows us to create a network topology by connecting instances of peers to each other. The topology is normally specified in a ".ned" file and looks as follows:

```
connections:
        node[0].out++ --> node[1].in++;
        node[0].in++ <-- node[1].out++;
```

In this example, node 0 can send messages to the node 1 and vice versa.

In addition to the topology file, Omnet++ requires modules, which will represent the logic of the different emulated peers within the network. All these modules must inherit from "cSimpleModule", which is provided by the Omnet++ framework. There are four virtual member functions, which need to be redefined in order to add functionality to the nodes.

- **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 an alternative to the handleMessage(cMessage *msg) method but is rarely used. We didn't used this method in our implementation.

- **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 having specified the logic of the different peers and their connections, the messages which are sent between the nodes must be declared. This can be done either by creating a class, which inherits from the Omnet++ class cMessage or, in a more convenient way, by declaring the messages in a ".msg" file. Omnet++ will then translate the ".msg" files automatically into C++ source code files. In our implementation, we have declared a basic OverlayMessage with the parameters source and destination.

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

All other messages in our implementation which are sent through the network inherit from this message.

3.1.2 Filter Chain Concept

To generate a modular implementation in the simulator, which could be later transferred into a real-world application, we have used a filter chain concept, which is shown in Fig. 3.1.

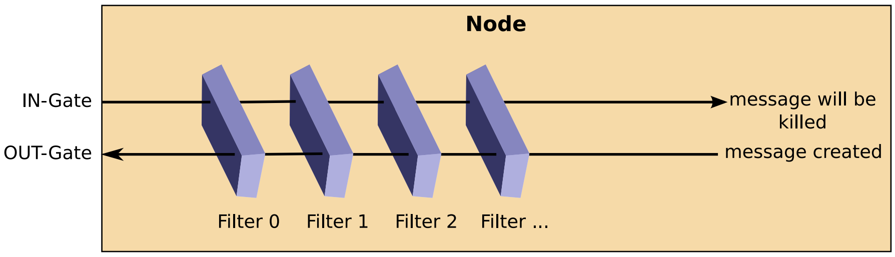


Figure 3.1: Filter Chain Example

The filter-chain concept is based on the mechanism, that every incoming message is piped through a chain of different filters. These filters provide different functionality to the node. Each filter will receive the incoming message, will eventually process this message and pass it on to the next filter. Once the message has reached the last filter, the message is considered as received and will be therefore destroyed. As shown later, we have implemented the P2P functionality as well as the QoS overlay framework as a dedicated filter.

3.1.3 Network Matrices

For our implementation, we need a more flexible network design approach than presented in 3.1.1. We want the topology file and the connections to be created automatically based on our inputs. Therefore, we have implemented the class distanet.cc, which creates a star topology with n nodes with a entity in the center called the distanet. Fig. 3.2 shows such a topology.

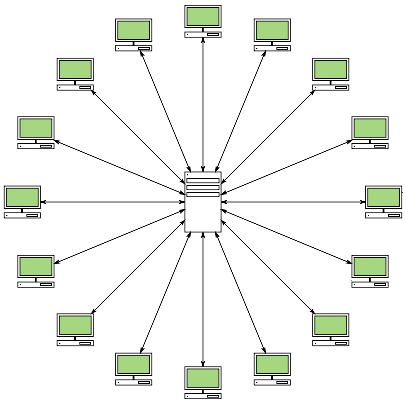


Figure 3.2: Distanet

The main purpose of the center entity was to simulate a real network based on distance matrices. A distance matrix is a file, which contains a Round-Trip-Time (RTT) matrix. With this RTT matrix, all messages, which are sent to a destination, are first routed to the distanet entity, where they are delayed using the appropriate RTT value from the distance matrix, and are then sent to their destination. This design has the advantage that we can simulate different network types, by just adapting the RTT matrix.

In our implementation, we used the universal topology generator ”BRITE” [15] to generate different real-world network topologies. For our evaluations, we have generated 13 different topology files with BRITE using the following parameters:

```
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
```

These topology files were used to generate the RTT matrices by calculating the shortest path for all nodes to each other in terms of delay. For each of these 13 topology files, we created the corresponding RTT matrice. In Table 3.1, we show the minimum, maximum and mean RTT values for those matrices.

Table 3.1: RTT 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 CAN Implementation

3.2.1 Bootstrapping

An introduction to CAN was already given in Chapter 2.2.3. In this Chapter we will explain in more detail our implementation of CAN, the problems encountered and further improvements of the CAN protocol. We will start with an entity, which is called the Bootstrap node.

The Bootstrap node simulates the entry point to a P2P network. For our implementation, we have defined a dedicated node as our Bootstrap node. However this functionality could also be decentralized and fully distributed. If a node wants to join an existing P2P network, it must at least know a rendezvous point in that network. Such a point provides initial configuration information to a joining peer, which allows successfully joining the network.

We have implemented this Bootstrap functionality as an independent filter in our filter-chain. This filter is only active on a predefined node in the network, which remains on-line during the entire simulation. For all other nodes, this filter remains inactive and it hands all messages directly on to the next filter in the filter chain.

In our implementation of CAN, the Bootsrap node maintains a list of up to n active node-IDs in the network. Using this list, the Bootstrap node has only a partial view of the network. A node, which wants to join the network, sends a ”CanBootstrapQueryMessage” to the Bootstrap node. This node replies to the message with a ”CanBootstrapResponseMessage”, containing m ($m \leq n$) IDs of its active node-ID list, which will be used for the joining process of the new node.

A node is considered as active, after it has successfully joined the network. Afterwards it sends a ”CanBootstrapInformReadyMessage” to the Bootstrap node to confirm the successful join. The Bootstrap node will then add this node-ID to the list of active node-IDs in the network according to the FIFO (First In, First Out) principle. An example of the active node list can be found in Fig. 3.3, which shows the active node-ID list with $n = 5$ before node J joins the network and Fig. 3.4 shows the list after node J has entered the network. The FIFO algorithm ensures that the node-ID list of the Bootstrap node is as accurate as possible. Furthermore, if a node leaves the network faultlessly, it informs the Bootstrap node about its leave, to have its ID removed from the active node-ID list.

If a Bootstrap node receives a ”CanBootstrapQueryMessage” and it has no members in the active node-ID list, it checks if another node has recently (within a specified time period) joined the network, but has not confirmed its active state. If this condition is true, it delays the incoming query message for a specified time, to wait for the ”CanBootstrapInformReadyMessage” of the other node. If the condition is false, the Bootstrap node considers the requesting node as the first one in the network and returns the ”CanBootstrapResponseMessage” with the requester node-ID. Therefore, the new node knows that it is the first node in the network. This adaptation was necessary to avoid timing problems, which we encountered during our simulations.

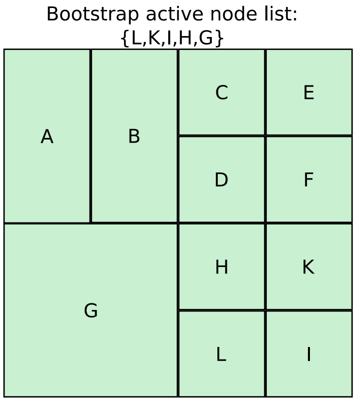


Figure 3.3: CAN - Bootstrap Node Network View before the Joining of Node J

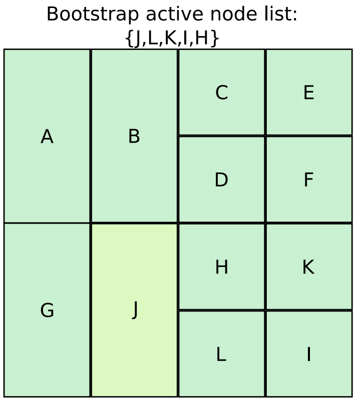


Figure 3.4: CAN - Bootstrap Node Network View after the Joining of Node J

3.2.2 Join a CAN Network

As explained in Section 3.2.1, a node which wants to join a specific CAN, contacts the Bootstrap node to receive a list of node-IDs, which are currently active in the network.

If the list contains only the node-ID of the joining node, it considers itself as the first node in the network. Therefore, it assigns the whole CAN keyspace as its responsible keyspace. The whole CAN keyspace is the space spanned over d axes (d is the number of dimensions). On every axis, the whole CAN keyspace is bounded between the value 0 and the value 1. Given that no other node is in the network, the joining node considers itself as joined and sends a ”CanBootstrapInformReadyMessage” back to the Bootstrap node.

If the list received from the Bootstrap node contains one or more node-IDs, which are not equal to its own ID, the normal join process is initiated. The joining node selects randomly a point P within the whole CAN keyspace and sends a ”CanJoinQueryMessage” to a random selected node from the list received from the Bootstrap node.

Once a node has received such a ”CanJoinQueryMessage”, it checks if the point P, which is stored in the message, belongs to its keyspace or to one of its temporary keyspaces. Temporary administrated keyspaces can occur after some nodes have left the network and will be explained in Section 3.2.4. In case this is true, the node starts with the ”split”-process. In case the check returns false, the node searches in its neighborhood for another node, which is closer (euclidean distance) to the point P than every other node in its neighborhood.

The neighborhood of a node consists of nodes that keyspaces are adjoining to its own keyspace. A node learns and maintains its own set of neighbors, the IP address of those nodes as well as their keyspace-coordinates in a neighbor-list.

If a node starts the split process, it asks its neighbors for permission to split. Once it has received all acknowledgments from the neighborhood, it can pursue with the split process. In case one or more neighbors do not allow the split, because they are also currently performing a split or they are currently in the ”leave” process, the declining node sends a ”CanJoinDeclinedMessage” back to the node, which wants to split itself. Therefore, the joining process is aborted and will restart, once the joining node has received the decline message. All nodes, which have positively acknowledged the split process are going into a freeze mode, where they cannot leave or process any incoming join queries until they receive the information that the split is over. This handshake algorithm is necessary to avoid timing conflicts.

In case the node has received all acknowledgments to perform a split, it checks if it currently has temporary keyspaces. If this check returns true, all information about the temporary keyspace, its coordinate space and its neighbors are sent with a "CanJoinResponseMessage" back to the newly joining node. The temporary keyspace sent as well as its neighbors are removed from the node, which has performed the split.

If the check returns false, the node starts to split its keyspace into two equal halves according to the following rule: The node is following a well-known ordering of the dimensions in order to decide along which dimension it has to split its keyspace. For example, for a three dimensional keyspace, it would first split along the X dimension, then along the Y dimension and then along the Z dimension before it would restart to split along the X dimension. This algorithm helps to better re-merge the keyspaces once a node decides to leave the network.

If a node has split its keyspace into two equal halves, it assigns one half as its own keyspace and sends a "CanJoinResponseMessage" back to the joining node containing the other half of its previous keyspace as well as information about its new keyspace and its old neighbors. Afterwards, it informs all its old neighbors with a "CanNeighborUpdateMessage" about the split and removes nodes form its neighbor list, which are no longer in its neighborhood. All nodes receiving this "CanNeighborUpdateMessage" will then update their neighbor list according to the new information from the message.

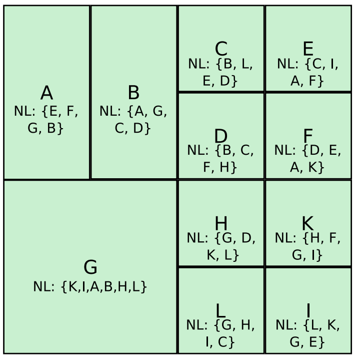


Figure 3.5: CAN - Neighbor List before the Joining of Node J

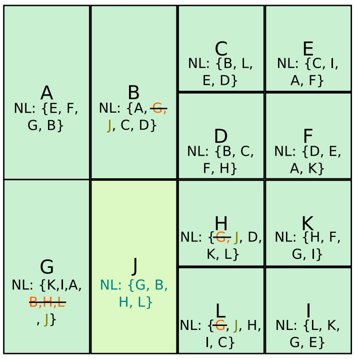


Figure 3.6: CAN - Neighbor List after the Joining of Node J

Once the joining node receives the "CanJoinResponseMessage", it assigns its new keyspace from the message. Furthermore, it updates its neighbor list with the neighbor information from the message but only takes over nodes, which are in its neighborhood according to its new keyspace. Afterwards it informs all nodes from the neighbor list with a "CanNeighborUpdateMessage" about its presence. Finally, it informs the Bootstrap node with a "CanBootstrapInformReadyMessage" that this node has successfully joined the network. Fig. 3.5 and Fig. 3.6 show the neighbor lists of the different nodes before and after node J has joined the network.

3.2.3 Multicast

CAN does not natively support multicasting. However, it can support multicast by building group specific mini CANs [16] and [17]. For each multicast group, a mini CAN is built on top of an existing CAN structure. Only nodes, which are interested in receiving messages from this multicast group will join this mini CAN, which has the same structure as a normal CAN.

As only nodes, which are interested in the topic join such a mini CAN, multicast messages can be forwarded using flooding algorithms. The simplest algorithm forwards multicast messages to all its neighbor, if it hasn't already received the message. However, such a flooding algorithm is very inefficient, tends to result in a lot of duplicates and does not make special use of the CAN structure. Therefore we have implemented the following forwarding algorithm [2]:

1. The source node forwards the messages to all its neighbors.

2. If i is the first dimension along which it does not overlap with the source. The message is then forwarded from the node to all its neighbors with which the node abuts along $1 - (i - 1)$ dimensions and to all its neighbors with which it abuts along dimension i, but in the direction going away from the source node.

3. The node does not forward a message along a particular dimension if that message has already traversed at least half-way across the space from the source coordinate along this dimension.

4. A node caches the sequence number of the already received multicast messages and does not forward a message containing a sequence number it has already received.

With this algorithm, CAN can support multicasting without generating duplicates in a perfectly partitioned coordinate space. However, this is almost never the case and to support ”temporary keyspaces”, which can appear if a node leaves the network (as explained in Section 3.2.4), we have added the following rule to the already proposed rules:

5. If a node has ”temporary keyspaces”, which it has to administrate, it forwards to its neighbors according to rule 1-4. Furthermore, it forwards the message to all neighbors of the ”temporary keyspaces” without sending the message multiple times to the same destination (if a node is in more than one neighborhood of the ”temporary keyspaces”) or to itself again.

With this additional rule, we can also distribute multicast messages to all listening nodes if we have ”temporary keyspaces”. Unfortunately the additional rule leads to more duplicates as we simply flood the message to all neighbors if it reaches a ”temporary keyspace”. Figure 3.7 shows an example of a multicast distribution from one sender to all nodes in a not perfectly partitioned 2-dimensional coordinate space. The red keyspaces are nodes, which receive duplicates.

3.2.4 Leave a CAN Network

Until now, we have seen how nodes can enter a CAN network and how multicast messages can be distributed in such a network. Now, we look at the leaving process. In our implementation, we

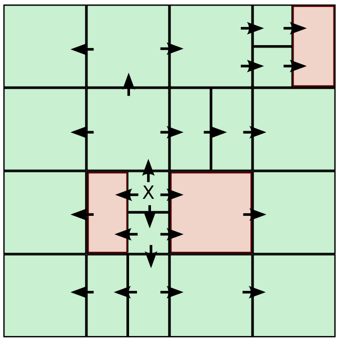


Figure 3.7: CAN - Multicast Distribution

have only considered cases where nodes leave the network faultlessly (graceful leave). We had to implement a handshake algorithm to avoid failures in the network. However, if a node would suddenly leave the network due to connectivity problems (non-graceful leaves), the network would be able to repair itself. The nodes are pinging their neighbors periodically to detect non-graceful leaves and perform a takeover if a non-graceful leave was detected. But this was not the topic investigated in this thesis.

Before a node can leave the network faultlessly, it performs a handshake algorithm with its neighborhood. Furthermore, it evaluates the ideal takeover node from its neighbor list. The ideal takeover node is the node, which could be easily merged with the current keyspace. More precise, it is the node which fulfills the following criteria:

- The takeover node is in the neighbor list.
- The takeover node's keyspace adjacent edge to the leaving node has the same coordinates as the adjacent edge of the leaving node's keyspace.

With these criteria, we can ensure that the takeover node can merge its own keyspace with the keyspace of the leaving node. An example can be found in Figures 3.8 and 3.9. In this example, node J wants to leave the network and has defined node G as takeover node. The neighbors from node J (nodes G, B, H and L) are in a frozen status. During the leave process, node G has merged its existing keyspace with the old keyspace from node J, as shown in Fig. 3.9.

However, if the leaving node does not find a takeover node, which fulfills the requirements above, it defines randomly a takeover node from its neighbor list. This takeover node cannot merge its own keyspace with the leaving node's keyspace, therefore it will manage this keyspace as an additional keyspace, called a temporary keyspace. A node, which manages in addition to its own also temporary keyspaces, has a neighbor list for every temporary keyspace. An example of a merge with temporary keyspaces can be found in Figures 3.10 and 3.11, where node G wants to leave and its neighbors A, B, H, L, K and I are in a frozen state. It chooses A as its takeover node. In Fig. 3.11, node A with its temporary keyspace is shown, as it was not able to merge the two keyspaces.

After the leaving node has received all acknowledgments from its neighbor list, and therefore also from the takeover node, it sends a "CanLeaveTransferMessage" to the takeover node con-

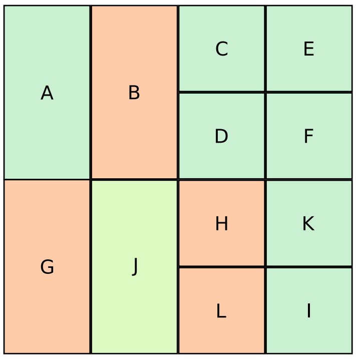


Figure 3.8: CAN - Keyspace before Leaving of Node J with Takeover Node G (Merge Possible)

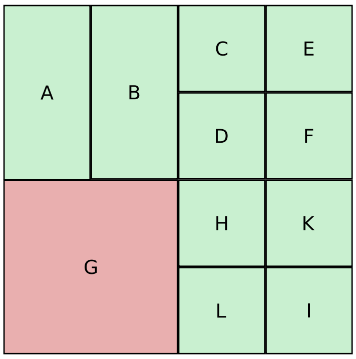


Figure 3.9: CAN - Keyspace after Merge from Leaving Node J to G

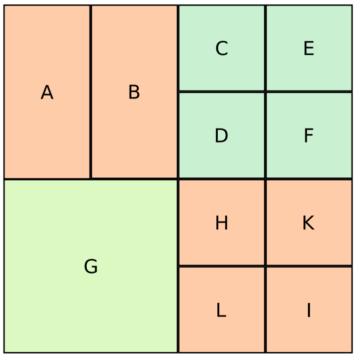


Figure 3.10: CAN - Keyspace before Leaving of Node G with Random Takeover Node A (Merge not Possible)

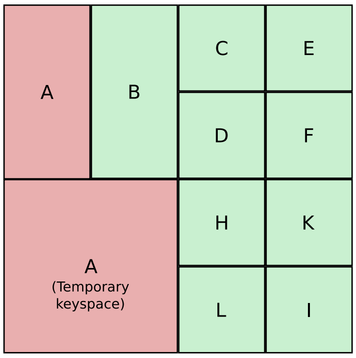


Figure 3.11: CAN - Keyspace after Leaving of Node G to Node A with Temporary Keyspace

taining its coordinate space as well as its neighbor list. Once the takeover node has received this message, it will contact all new neighbors and inform them about the change in the network. During this leave process, all nodes which have previously accepted the handshake are in a freeze mode, meaning that they cannot accept a join or have to wait until they can leave. Once they receive the new network information from the takeover node, the freeze is revoked.

Once a leaving node has sent its ”CanLeaveTransferMessage” to the takeover node, it sends a final ”CanBootstrapLeaveMessage” to the Bootstrap node. The Bootstrap node then removes the leaving node from its active node list to avoid that a newly joining node receives the contact information about a node that already left the CAN.

3.3 Overlay Multicast QoS Framework

3.3.1 Basic Network

To provide QoS guarantees to a P2P network, which has no native Quality of Service (QoS) support, we have designed an Overlay Multicast QoS (OM-QoS) Framework [18], which enables QoS functionality in those networks. The main goal was to design a framework, which intercepts messages from the underlying P2P network and acts accordingly. Therefore, only a few adaptations have to be made within the P2P network.

In our implementation, we added a new filter, the OM-QoS Framework filter, to our existing filter chain (Bootstrap filter - P2P network filter). This filter has to be inserted before the P2P network filter, as it needs to intercept packets from it. Therefore, the filter chain looks now as follows: Bootstrap filter - OM-QoS Framework filter - P2P network filter. Furthermore, the implementation of the Bootstrap filter, as described in Section 3.2.1, had to be extended.

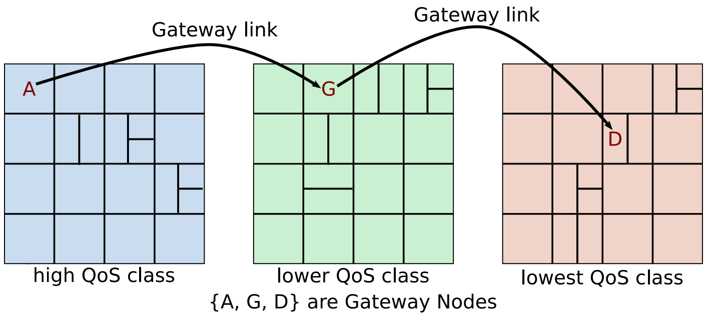


Figure 3.12: Overlay Multicast QoS Framework Basic Design with P2P Network CAN

The basic network design can be described as a chain structure. For every QoS class, a separate P2P network is built. These separate P2P networks are connected to each other over gateway links. Each of these P2P networks have a designated node, which is normally the root in those networks, and which is responsible for establishing a link to the next lower QoS class.

Such a node is called a gateway node and is a normal node in the network with additional responsibilities. In Fig. 3.12, an example is given with 3 different QoS classes. For each QoS class, a separate network was established and the networks are linked with each other over the gateway nodes (A, G, D).

3.3.2 Bootstrapping

To enable the Overlay Multicast QoS (OM-QoS) Framework in a P2P network, the Bootstrap filter of the existing P2P network needs to inherit from the OM-QoS Framework Bootstrap filter. For our implementation, we have used one designated node as Bootstrap node for all P2P network entities. However, this functionality could also be decentralized and fully distributed. If a node wants to join the network, it sends a ”Query Message” (P2P network dependent) to the Bootstrap, containing the QoS class of the joining node. Instead of sending back a set of random active nodes, the Bootstrap filter asks the OM-QoS Framework Bootstrap filter for a set of random active nodes for this specific QoS class. The OM-QoS Framework Bootstrap filter maintains not only a list of random active nodes for the network, but also maintains such a list for every QoS class. Then the Bootstrap node replies to this ”Query Message” with a ”Response Message”(P2P network dependent), which contains the random active nodes received from the OM-QoS Framework Bootstrap filter.

Once a node has successfully joined the P2P network, it sends an ”OvFrBootstrapInformReadyMessage” message back to the Bootstrap. The Bootstrap will then add this node to the active node list of the joined node’s QoS class. It will add this node to the list according to the FIFO principle to ensure that the active node-ID list of the Bootstrap for each QoS class is as accurate as possible.

Furthermore, the Bootstrap checks if the joined node is the first one for its specific QoS class. If this is the case, the Bootstrap considers this node as a gateway node for this QoS class and sends an additional ”OvFrInformGatewayMessage” to the joined node. This message will enable the OM-QoS Framework on the joined node. In addition, the Bootstrap sends an ”OvFrInformGatewayMessage” to the gateway node of the next higher QoS class to inform the node that a new gateway node in the next lower QoS class has appeared.

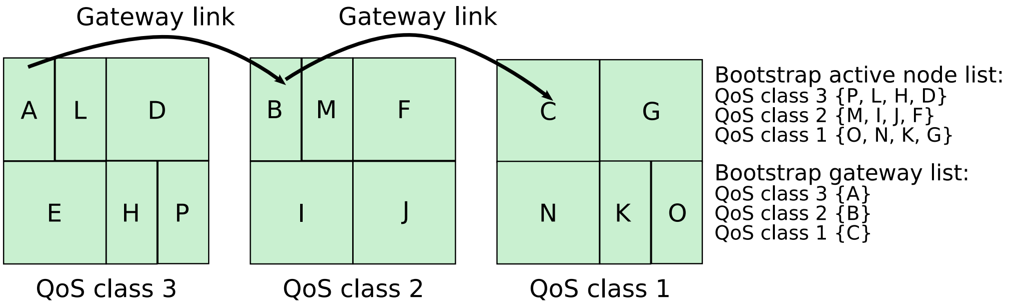


Figure 3.13: Overlay Multicast QoS Framework Bootstrap View

Moreover, the Bootstrap maintains an overview of all gateway nodes from the different QoS networks. This ensures that if multiple nodes in the network are failing, the network can still restore the gateway links. Therefore, the Bootstrap reacts to an ”OvFrRequestForGatewayMessage” with an ”OvFrInformGatewayMessage” to inform a gateway node about the gateway node in the next lower QoS class. An example of the Bootstrap’s active node list and its gateway list can be found in Fig. 3.13.

If the Bootstrap is informed about a leave of a node (faultless leave), it removes this node from the active list for the respective QoS class. Furthermore, it checks if that node was a gateway node. If this check returns true and the gateway node was not in the highest QoS class, it removes the node from the gateway node list and replaces it with a temporary placeholder. This is needed, as the next higher gateway node will determine the successor of the leaving gateway node, which will then inform the Bootstrap about its new presence.

If the check returns true and the leaving gateway node was in the highest QoS class, the Bootstrap checks if the leaving node had a successor in the P2P network. If this is the case, the successor will be informed by the Bootstrap that it is the new gateway node for this QoS class. If the leaving node had no successor in the P2P network, the Bootstrap will determine a node from the active node list as the new gateway node for that QoS class. If a leaving node has a successor or not depends on the underlying P2P network topology.

3.3.3 Join the Overlay Multicast QoS Framework

The gateway node’s functionality is normally deactivated. Once a node receives an ”OvFrInformGatewayMessage” from the Bootstrap or an ”OvFrInitializeNewGatewayMessage” from its parent gateway node, the gateway node’s functionality is initialized. If the gateway node has received an ”OvFrInformGatewayMessage”, it will extract the next lower gateway node-ID from the message and adds it as its new child. This means that every incoming multicast message will be forwarded to those children.

If the node was initialized by an ”OvFrInitializeNewGatewayMessage” from its parent gateway, it sends an ”OvFrRequestForGatewayMessage” to the Bootstrap to request information about the gateway node in the next lower QoS class. The Bootstrap will respond to this message with an ”OvFrInformGatewayMessage”, which includes the requested information.

Once the gateway node has received the information about the next lower gateway node, it successfully joined the OM-QoS network.

3.3.4 Root Transfer

Some P2P networks (e.g. NICE, CHORD) have a predefined root in their networks. For the Overlay Multicast QoS Framework, this root should also be the gateway node. The first node which joins such a network, is defined as the root and regarding the OM-QoS Framework’s view it is also a gateway node. Over time, more and more nodes join the network and the root is changing. Therefore, the OM-QoS Framework filter must also react to these changes.

An active gateway node will intercept the ”OvFrRootTransferMessage” from the underlying P2P network and will then send an ”OvFrGatewayNodeLeavingMessage” to its parent gateway and to the gateway node in the next lower QoS class. This message contains the new root as a parameter. The parent gateway node will react to this message by sending an ”OvFrInitializeNewGatewayMessage” to the new root to activate its OM-QoS Framework filter.

After sending these two messages, the old root will then deactivate its OM-QoS Framework filter and consider itself as a normal node and not as a gateway node anymore.

3.3.5 Leave the Overlay Multicast QoS Framework

If a node in a P2P network is an active gateway node, meaning that the OM-QoS Framework filter is activated, it will intercept the ”OvFrNodeLeaveMessage” from the underlying P2P network. Once the filter receives such a message, it sends an ”OvFrGatewayNodeLeavingMessage” to its parent and the gateway node in the next lower QoS class. This message can contain a node, which will be the takeover node of the leaving node. However, not every P2P network defines such a parameter, and therefore this is not mandatory.

After the parent gateway node has received the ”OvFrGatewayNodeLeavingMessage”, it is responsible for defining a new gateway node in the next lower QoS class. If it has received a takeover node in the ”OvFrGatewayNodeLeavingMessage”, it considers this node as the new gateway node and sends an ”OvFrInitializeNewGatewayMessage” to this node-ID. If no takeover node was defined, it enables a backup link gateway (described in the next Section) and sends an ”OvFrInitializeNewGatewayMessage” to this node-ID.

If neither a takeover node was defined nor a possible backup link exists, the node sends an ”OvFrRequestForGatewayMessage” to the Bootstrap. The Bootstrap will then try to find another node in the network for the requested QoS class in its active-node list. If the Bootstrap finds another node, it will send an ”OvFrInformGatewayMessage” to the newly found node and to its parent node, which was the source of the request. If the Bootstrap does not find another node for the requested QoS class, it will state that no other active nodes with the requested QoS are available and will therefore answer to the request with an ”OvFrInformGatewayMessage” containing the gateway node-ID of the next lower QoS class.

3.3.6 Backup Links

For P2P networks, which have no designated takeover node for a leaving node, the parent gateway must ask the Bootstrap for searching the new gateway node, as described in the previous Section. This method is not very efficient and can lead to a certain delay and therefore to a loss of multicast messages. To reduce this delay, we have introduced Backup Links.

Once a node is a gateway node, meaning that its OM-QoS Framework filter is active, it sends an ”OvFrRequestNeighborMessage” to the gateway node in the next lower QoS class. This message is received by the underlying P2P network, which responds to it with an ”OvFrReturnNeighborMessage” containing node-IDs from the neighborhood of this node.

After receiving such an ”OvFrReturnNeighborMessage”, the gateway node will store these node-IDs as backup links. It will periodically ping these nodes to ensure that they are still in the network and will update the list by periodically sending an ”OvFrReturnNeighborMessage” to the gateway node in the lower QoS class. An example of the OM-QoS Framework with backup links is shown in Fig. 3.14. There, the backup links are represented by dashed lines.

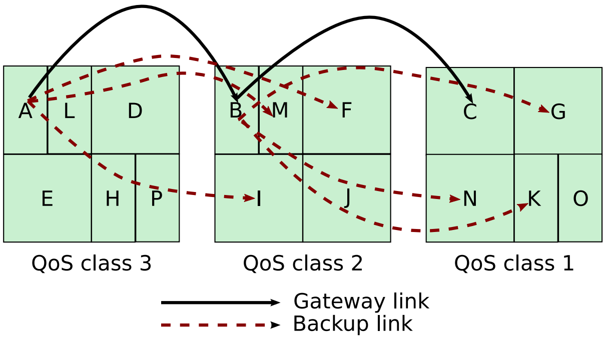


Figure 3.14: Overlay Multicast QoS Framework with Backup Links

3.3.7 Hop Optimization

The implemented chain design is a solid framework, however it has a major drawback. If we have n different QoS classes, the nodes which are in the lowest QoS class, will receive the message after at least n hops. This leads to a large RTT to root, which can be a problem for certain multicast applications.

To solve this problem, we have implemented a hop optimization algorithm, which establishes additional gateway links over the existing chain topology. These additional gateway links jump over multiple QoS classes and solve the the above described issue.

Once a gateway node is successfully activated, it sends periodically ”OvFrSearchMessage” to its parent gateway node. A parent node, which receives such a message checks if the following condition is met: $currentQoSClass \geq n * senderQoSClass, n \in (2, 3, ..., g)$. The parameter g is the maximum number of additional gateway links for each gateway node. If the condition is met, the parent node establishes an additional gateway to the sender of the ”OvFrSearchMessage”. If the condition is not met, the node forwards the message to its own parent and so on until the message receives a gateway node with a QoS class, which is high enough to meet the condition mentioned above. In Fig. 3.15, an OM-QoS Framework with additional gateway links is shown.

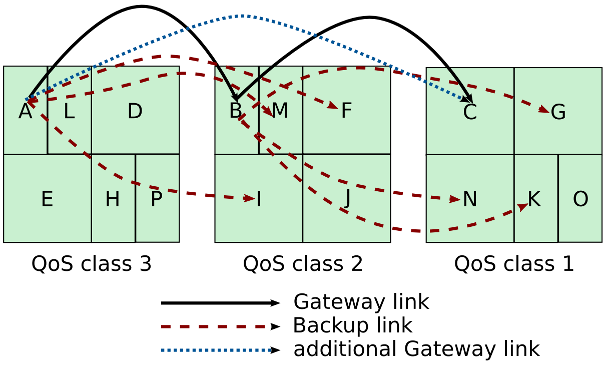


Figure 3.15: Overlay Multicast QoS Framework with Additional Gateway Links

Chapter 4

Evaluation and Results

4.1 Evaluation Scenarios

In this Chapter, we explain the specifications and scenarios for the evaluations of our implementation. Furthermore, we present the results conducted from these scenarios.

For the evaluations, we have used 13 different RTT matrices as described in Chapter 3.1.3. The characteristics of these RTT matrices can be found in Table 3.1. For each of those matrices, we have performed evaluations with 20 different random seeds. The seeds are used by the random number generators (RNG) as input variables. Within our implementation, we used the Mersenne Twister RNG [19]. This RNG has a period of $2^{19937} - 1$, and 623-dimensional equidistribution property is assured. In our implementation, the RNG is used for the uniform distribution of message delays. This feature is mostly used for periodically sent messages or for defining the leave time of a node. The random seed values used for our evaluations can be found in Table 4.1.

In this Chapter, we will describe the evaluation of different scenarios with our implementation. A scenario is a set of evaluations calculated with the same network protocol specific parameters, but with different number of nodes, different RTT matrices, and different random seeds. An overview of the evaluated scenarios can be found in Table 4.2.

Every performed scenario was evaluated with 20 different network ranges. The ranges have a node count from 100 to 2000 in steps of 100. Therefore we have performed 5200 simulations for every scenario: 20 (different number of nodes) * 20 (different RNG seeds) * 13 (different RTT matrices).

The min. and max. values, which are shown in the following Sections, are the maximum / minimum values calculated from all evaluations, for the respective number of nodes for a certain scenario.

For all results presented in this Chapter, we have removed 1% of the outliers (0.5% of the minimum and maximum values each).

In the next Sections we will discuss the following network metrics for each evaluation scenario:

- **Hop count**
  The number of nodes a multicast message passes until it reaches its destination.

- **Fan-out**
  The number of children of a node in the multicast tree

- **Node to root RTT**
  The RTT between the node and the root of the multicast tree

- **Average duplicates per multicast message**
  How many duplicates a node receives for one multicast message

- **Received total multicast messages**
  The percentage of received multicast messages

- **Join time**
  How long a node takes from starting the join procedure until it has received all necessary information from the P2P network. For CAN, this means that the join time is the time it takes until a node receives its new CanKeySpace and has received all information about its neighbors.

- **Leave time**
  The time a node takes to leave the network from the moment it decides to leave the network until it has completed the leaving handshake algorithm.

- **Rejoin time**
  How long a node takes to leave the network completely and then reconnects again.

4.2 CAN without QOS

4.2.1 Normal Mode with 2-Dimensional CanKeySpace

Figure 4.1 shows the evaluation results from our native CAN P2P implementation (Section 3.2) with a 2-dimensional CanKeySpace scenario.

The hop count, which is the path length of a node to the root, can be found in Fig. 4.1(a). The average is rising logarithmically with an upper bound around 16. The maximum is steadily rising from 10 nodes to 40 nodes, and the minimum value stays at 0 as the root has always a routing path length of 0. However in the Figure, the min. value rises from 0 to 1 due to the removal of the outliers. This Figure shows that the multicast tree built in the CAN network is equally distributed, as we have a logarithmically rising average curve.

Figure 4.1(b) shows the fan-out of the evaluation runs. The average stays constant around 2.5. The max. value stays also constant around 19, which proves that the CAN network scales well. The min. value stays at 0 as there are always nodes, which do not have any children.

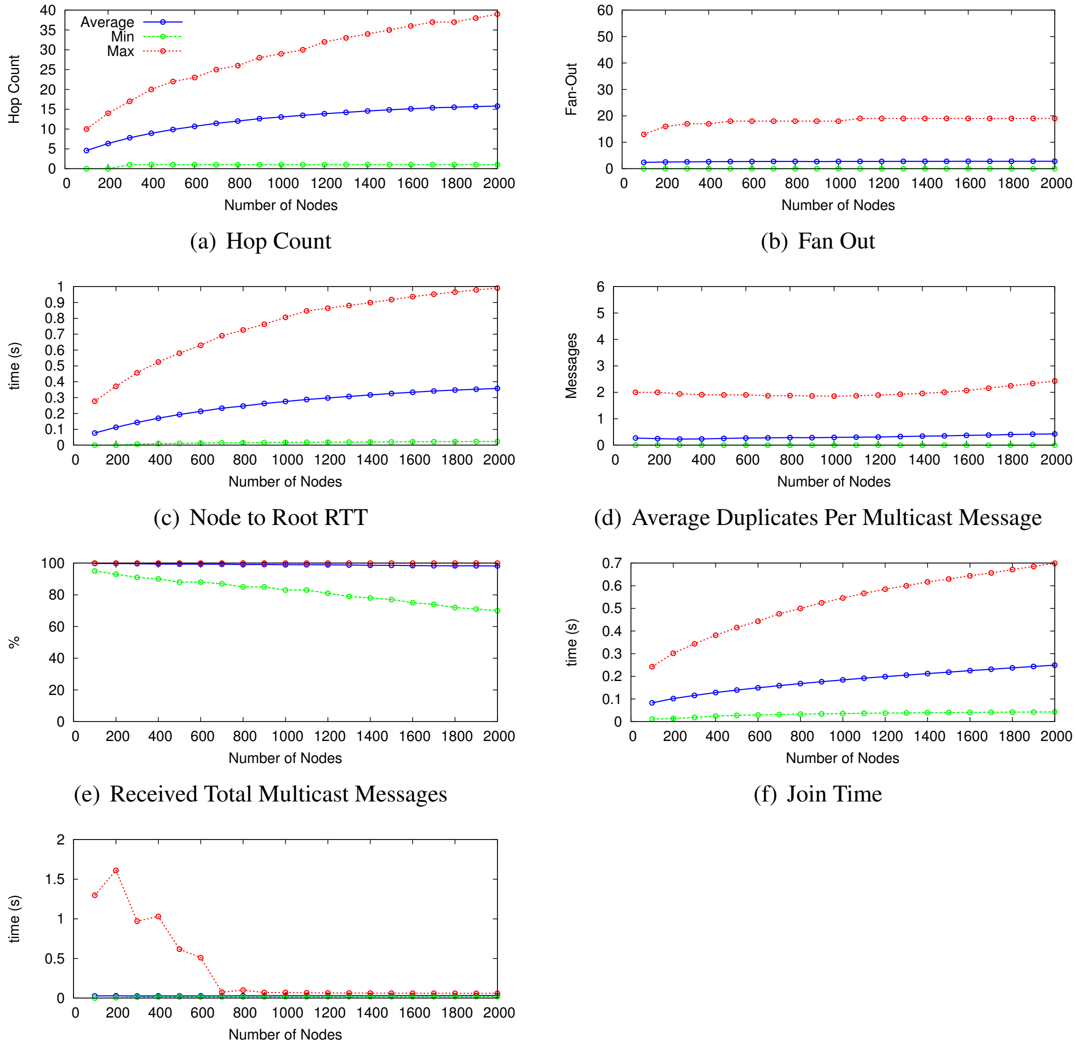
In Fig. 4.1(c), the node to root RTT is presented. Like the hop count, the average node to root RTT is rising logarithmically with an upper bound around $0.4s$. The min. value should stay at 0 as the root has always a node to root RTT of 0. In the Figure, the min. value is though

Table 4.1: Random Input Seeds

1768507984
33648008
1082809519
703931312
1856610745
784675296
426676692
1100642647
1359921031
1209575029
640572720
1569615780
1142429693
307193866
34708029
97450298
743126457
593716555
910097052
449294716

Table 4.2: Evaluation Scenarios

CAN - Normal Mode with 2-Dimensional CanKeySpace
CAN - Normal Mode with 3-Dimensional CanKeySpace
CAN - Normal Mode (Small Network) with 2-Dimensional CanKeySpace
CAN - Stressed Mode with 2-Dimensional CanKeySpace
Overlay Multicast QoS Framework with CAN - Hard QoS with 32 QoS Classes
Overlay Multicast QoS Framework with CAN - Hard QoS with 64 QoS Classes
Overlay Multicast QoS Framework with CAN - Hard QoS with 128 QoS Classes
Overlay Multicast QoS Framework with CAN - Hard QoS with 256 QoS Classes
Overlay Multicast QoS Framework with CAN - Soft QoS with 32 QoS Classes
Overlay Multicast QoS Framework with CAN - Soft QoS with 64 QoS Classes
Overlay Multicast QoS Framework with CAN - Soft QoS with 128 QoS Classes
Overlay Multicast QoS Framework with CAN - Soft QoS with 256 QoS Classes
Overlay Multicast QoS Framework with NICE - Hard QoS with 32 QoS Classes
Overlay Multicast QoS Framework with NICE - Hard QoS with 256 QoS Classes



(a) Hop Count

(b) Fan Out

(c) Node to Root RTT

(d) Average Duplicates Per Multicast Message

(e) Received Total Multicast Messages

(f) Join Time

(g) Leave Time

Figure 4.1: CAN - Normal Mode with 2-Dimensional CanKeySpace

slightly higher as we have removed the outliers. This Figure shows again the equally distributed multicast tree of the CAN network, as we have a logarithmically rising average curve.

The average duplicates per multicast message is shown in Fig. 4.1(d). This parameter shows how many duplicates in average per multicast message are still generated within CAN, despite the improvements (as explained in Section 3.2.3). The average is around 0.25 duplicates per multicast message, which is due to the nature of CAN. However, the minimum stays at 0, which proves that duplicates can often be avoided. In the worst case, there can be a maximum of 2 to 2.5 duplicates for a multicast message.

In Fig. 4.1(e), the percentage of received multicast messages is shown. The average is around 99% received multicast message, which is an excellent value. However, the min. value is slowly decreasing to 70%. This can be explained through timing problems or due to the duplicate suppression algorithm, which still could be improved. The best solution for a node, which does not receive multicast messages anymore, is to rejoin the network again. The node will then receive a new CanKeySpace with a new parent node, which will provide again multicast messages to the node.

The last two Figures show the join time (4.1(f)) and the leave time (4.1(g)). The average join time is slowly increasing starting from $0.09s$ for 100 nodes to $0.28s$ for 2000 nodes. The increase is explainable due to the joining algorithm, which routes the "CanJoinQueryMessage" through the network (explained in Section 3.2.2). If the network increases, the join message has to be routed over more hops to reach its destination and therefore, the join time increases as well.

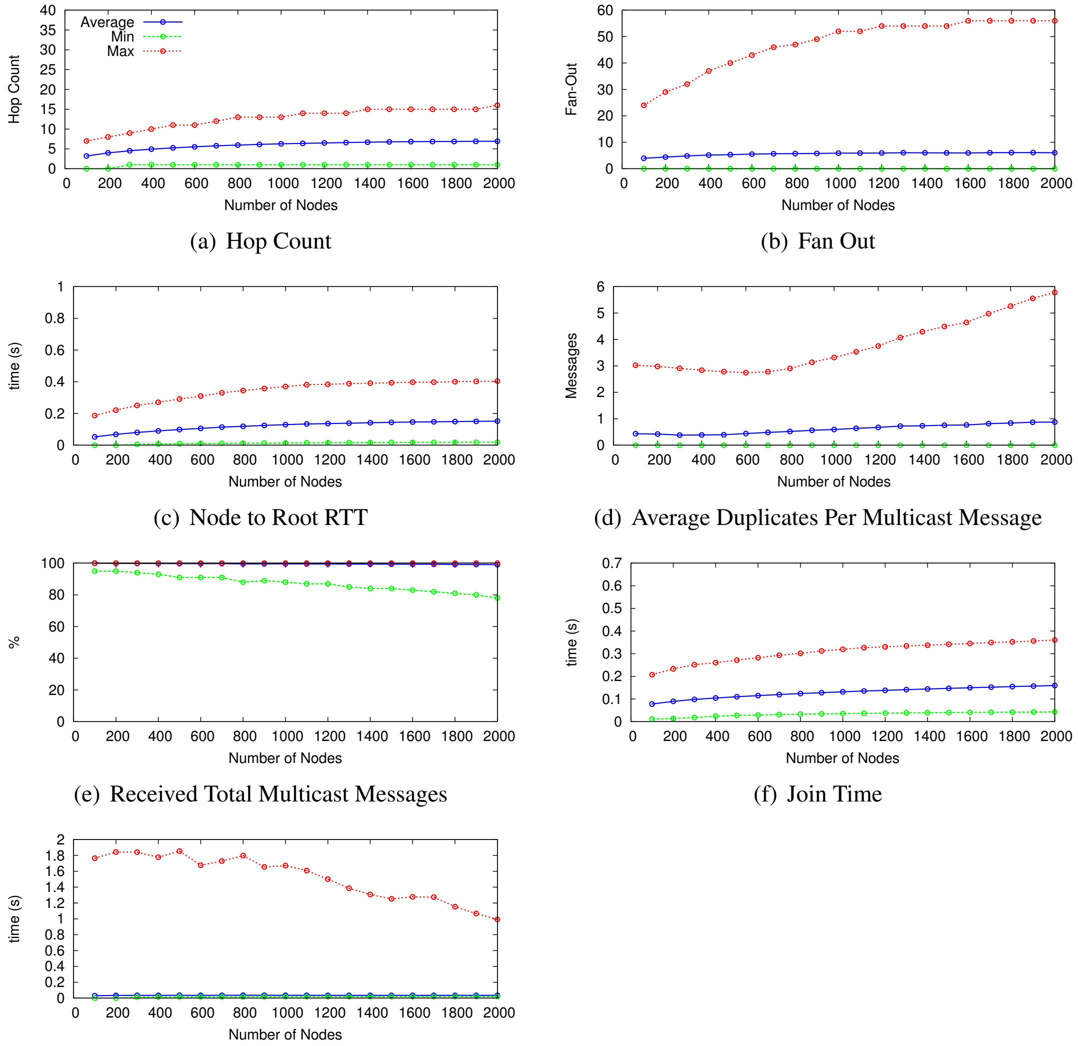
The average leave time of the node stays constant around $0.03s$. But the maximum value starts with high peaks around $1 - 1.5s$ for 100 to 400 nodes and is then rapidly decreasing to $0.06s$ where it stays constant with increasing number of nodes. The peaks can be explained through timing problems within the leave process itself. If a node wants to leave, all nodes in the neighborhood are going into a freeze status, during which they cannot leave until the other node has finished its leave process (explained in Section 3.2.4). As the nodes in our implementation start to leave randomly within a certain time frame, the possibility that they interfere with each other decreases with the number of nodes in the system and therefore, the max. leave time is decreasing with an increasing number of nodes in the system.

To sum up, we have shown in this Section that our CAN implementation is working and that CAN scales well but that the issue with the duplicate multicast messages could still be improved.

4.2.2 Normal Mode with 3-Dimensional CanKeySpace

Figure 4.2 shows the evaluation results from our CAN P2P implementation (Section 3.2) with a three-Dimensional CanKeySpace scenario.

The hop count is shown in Fig. 4.2(a) and has an average around 5 and a maximum value, which is slowly increasing from 6 to 16. Compared to the results with a two-dimensional CanKeySpace, the average and maximum hop count are significantly lower and more flattened. This behavior can be explained with the presence of the third dimension, which provides CanKeySpaces with more neighbors, and therefore more routing possibilities. This leads to a reduction of the routing path length.



(a) Hop Count

(b) Fan Out

(c) Node to Root RTT

(d) Average Duplicates Per Multicast Message

(e) Received Total Multicast Messages

(f) Join Time

(g) Leave Time

Figure 4.2: CAN - Normal Mode with 3-Dimensional CanKeySpace

This is also the explanation for the higher fan-out in Fig. 4.2(b) compared to the fan-out with the two-dimensional CanKeySpace in Fig. 4.1(b), as the nodes have more neighbors. With a three-dimensional CanKeySpace, the average fan-out stays constant around 5.

The node to root RTT in Fig. 4.2(c) has an average value, which slowly increases from $0.08s$ to $0.18s$. Compared to the average value with a two-dimensional CanKeySpace in Fig. 4.1(c), the value has been significantly decreased. As we have a significant decrease of the path length, the root to RTT had to decrease as well.

In Fig. 4.2(d), the average duplicates per multicast message is shown. The average increases from 0.4 to 0.98 duplicates per multicast message. Compared to Fig. 4.1(d) with a 2 dimensional CanKeySpace, the average increased by 0.5. This effect can be explained again with the higher number of neighbors per node.

Figure 4.2(e) shows the percentage of received multicast message. There the average stays almost constant at 99.8% received multicast messages. The additional dimension has almost no influence to this parameter, if we compare it with Fig. 4.1(e) with a two-dimensional CanKeySpace.

The last two Figures show the leave and join time of the nodes in the simulation. As expected, the average and maximum join time are lower as in Fig. 4.1(f) with only two-dimensional CanKeySpaces. The reason for that is that the average path length in a three-dimensional CanKeySpace is shorter, and therefore the time until the join message force a node to split is shorter.

However, the max. leave time in Fig. 4.2(g) is only slowly decreasing from $1.8s$ to $1s$. Compared to Fig. 4.1(g), there is no peak for small number of nodes. Again, this can be explained with the higher number of neighbors, and therefore fewer nodes interfere when they leave the network.

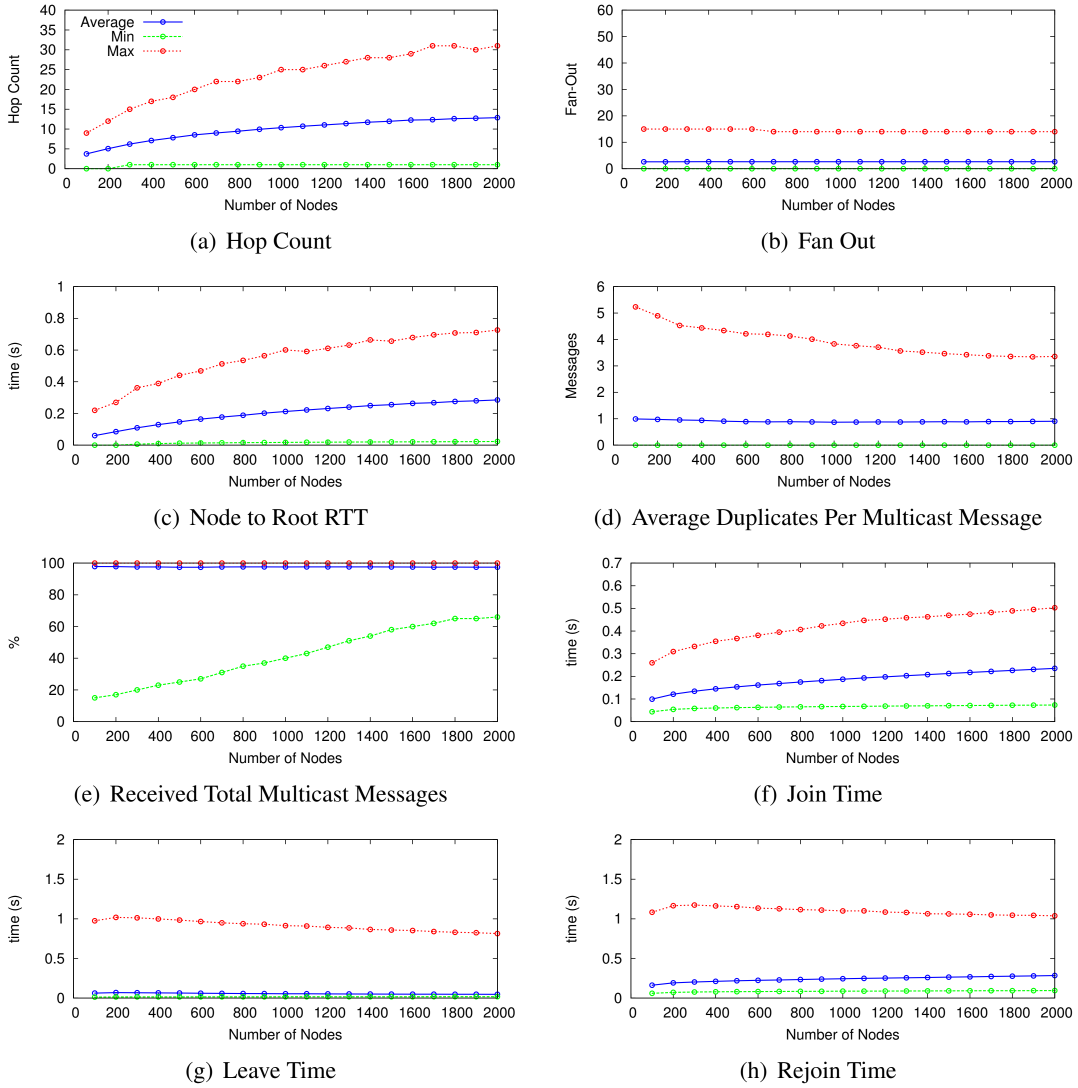
To sum up, we have shown that our CAN implementation is also working with higher CanKeySpace dimensions. With a higher number of dimensions, a better node to root RTT can be achieved, but on the cost that the fan-out of the nodes increases. As such a high fan-out is often not suitable for applications in the network, we have tested further simulations only with two-dimensional CanKeySpaces.

4.2.3 Stressed Mode with 2-Dimensional CanKeySpace

In this Section, we present the results for a two-dimensional CanKeySpace scenario in a stressed network environment. This means that every node joins and leaves the network up to 5 times (uniformly distributed from 1 to 5) with an interval of 2 seconds starting 10 seconds after the node has successfully joined the network. In Fig. 4.3, the results of this scenario are shown.

Compared to the normal mode as shown in Fig. 4.1(a), the hop count in Fig. 4.3(a) has a 17% lower average, which slowly increases from 3.7 to 12.87. The maximum value in the stressed mode is also slightly lower than in the normal mode. This behavior can be explained with the better distribution of the CanKeySpaces and also with the appearance of temporary CanKeySpaces (explained in Section 3.2.3), which have a influence on the message distribution algorithm.

The fan-out, which is shown in Fig. 4.3(b), has a constant average around 2.6 and a constant maximum around 14. Compared to the normal mode, as shown in Fig. 4.1(b), the maximum



(a) Hop Count

(b) Fan Out

(c) Node to Root RTT

(d) Average Duplicates Per Multicast Message

(e) Received Total Multicast Messages

(f) Join Time

(g) Leave Time

(h) Rejoin Time

Figure 4.3: CAN - Stressed Mode with 2-Dimensional CanKeySpace

decreases slowly, but despite that, the stressed mode has no influence on the fan-out.

Figure 4.3(c) shows the node to root RTT. The average is slowly rising from $0.06s$ to $0.28s$. Compared to the normal mode in Fig. 4.1(c), the average and the maximum values are significantly lower. This behavior can be explained with the lower hop count of the stressed mode, which leads to the lower node to root RTT.

Looking at Fig. 4.3(d), which shows the average duplicates per multicast message, the average stays around 1 and the maximum is slowly decreasing from 5 to 3.2. Compared to the normal mode as shown in Fig. 4.1(d), the average and the maximum values are both up to two or three times higher. This behavior can again be explained with the appearance of temporary CanKeySpaces, which inhibit an efficient duplicate suppression algorithm.

The percentage of received multicast messages can be found in Fig. 4.3(e). Compared to the normal mode, the average decreased by 1% to a constant value around 98%, which is an excellent performance. However, the minimum value slowly increases from 19% to 65%. This behavior can be explained with the nature of the stressed mode. If nodes join and leave the network quite rapidly, multicast messages can get lost. However, this is only the case for a very few number of nodes which the average value confirms.

The join time in Fig. 4.3(f) has an average, which rises slowly from $0.1s$ to $0.21s$. The maximum value is also slowly rising from $0.28s$ to $0.5s$. Comparing with the normal mode as shown in Fig. 4.1(f), the average behaves equally. However, the maximum value increases slower than in the normal mode, due to the existence of temporary CanKeySpaces and better distribution of the CanKeySpace, which leads to a faster routing of the ”CanJoinQueryMessage”, and therefore to a faster join time.

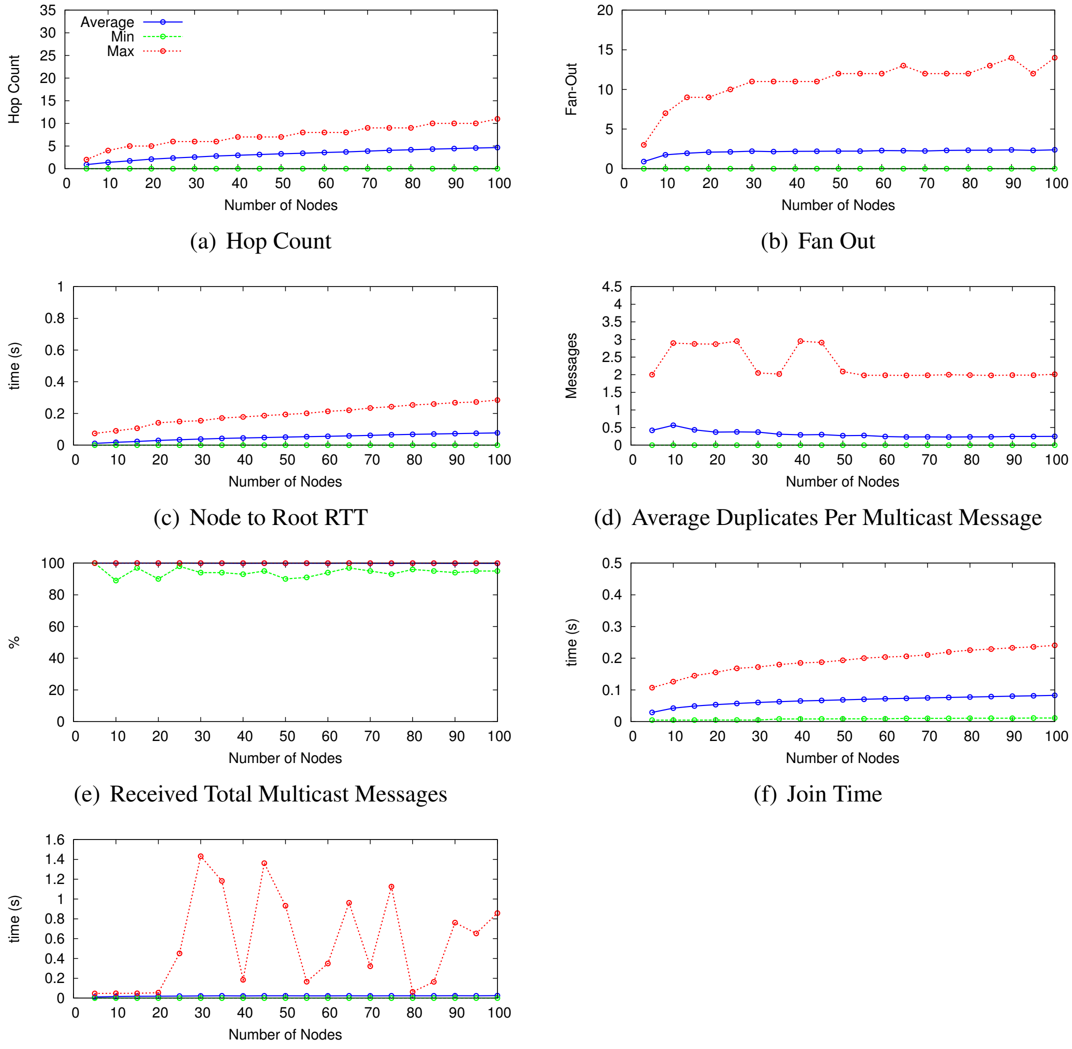
The leave time is presented in Fig. 4.3(g). The average value is comparable with the average value from the normal mode as shown in Fig. 4.1(g), which is around $0.03s$. But the maximum value is slowly decreasing from around $1s$ to $0.8s$, which is different from the normal mode. It can be explained again with the appearance of temporary CanKeySpaces, which leads to more neighbors for a leaving node, and therefore to more interference between multiple leaving nodes. That is why the peak behavior of the normal mode became almost a high constant value in the stressed mode.

The rejoin time in Fig. 4.3(h) shows how long a node takes to rejoin the network. It consits of the leave time and the join time. The average rejoin time stays constant around $0.16s$, which is acceptable.

To sum up, we have shown that our CAN implementation performs well in a stressed environment. Some metrics showed even better results as in the normal mode due to the more balanced distribution of the CanKeySpaces and the therefore more balanced multicast tree. To improve the normal network, a rejoin counter could be implemented, so that every node rejoins the network after a certain timeperiod to get a better CanKeySpace distribution.

4.2.4 Normal Mode (Small Network) with 2-Dimensional CanKeySpace

Figure 4.4 is showing the evaluation results from our native CAN P2P implementation (Section 3.2) for small networks. We have conducted the evaluation with different small network



(a) Hop Count

(b) Fan Out

(c) Node to Root RTT

(d) Average Duplicates Per Multicast Message

(e) Received Total Multicast Messages

(f) Join Time

(g) Leave Time

Figure 4.4: CAN - Normal Mode (Small Network) with 2-Dimensional CanKeySpace

sizes, from 10 to 100 nodes in 10 nodes steps. In Section 4.3 the evaluations with the Overlay Multicast QoS Framework are shown, where we have small networks for each QoS class. To better compare these results with the native CAN implementation we present here the results for the native CAN implementation with small networks.

The hop count in Fig. 4.4(a) and the fan-out in Fig. 4.4(b) behave both as expected. The average hop count rises slowly from 1 to 4.5 and the fan-out stays constant around 2.1.

The node to root RTT in Fig. 4.4(c) increases respectively to the hop count with an average from 0.02s to $0.08s$.

The average duplicates per multicast message, which is shown in Fig. 4.4(d) has an average, which is slightly decreasing from 0.5 to 0.25. This shows that also for small networks duplicates are generated, due to the nature of CAN. The percentage of received multicast messages has an average of 99.9% received multicast messages and the minimum has a lower bound at 90%.

The nodes have almost no delay to join the network as shown in Fig. 4.4(f), where the average join time is slowly increasing from $0.02s$ to $0.08s$. This behavior can be explained with the join algorithm and the fact that the less nodes are in a CAN network, the lower the travel time of the JOIN message through the network is, as explained in Section 3.2.2.

The average leave time shown in Fig. 4.4(g) is around $0.02s$. The maximum shows multiple peaks up to $1.4s$ for the leave time. Depending on the internal allocation of the CanKeySpaces, nodes can interfere with each other when they are leaving, and therefore can delay each other. This interference of the leaving nodes generates the multiple peaks shown in Figure 4.4(g).

4.2.5 Quality of Service Analysis of CAN

The main goal of this thesis was to enable Quality of Service for P2P networks, which do not have a native QoS Support. As CAN is a P2P network, which does not natively support Quality of Service, we wanted to know how many paths in a multicast tree do randomly support QoS, if we assume that every node has an allocated QoS class. A path in the multicast tree fulfills the QoS requirements if every condition described in Section 2.3 is meet.

Figure 4.5 shows the percentages of QoS paths fulfilled in a native CAN implementation for different number of QoS classes. For every number of QoS classes, the average decreased from 22% to 9%.

As shown in the next Section, we have managed to improve this value to 100% QoS paths fulfilled for different numbers of QoS classes with our Overlay QoS framework.

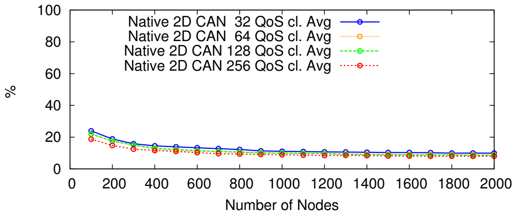


Figure 4.5: CAN - QoS Paths Fulfilled

4.3 Overlay Multicast QoS Framework with CAN

In the next Sections, we will introduce the results of our Overlay Multicast QoS Framework implementation with an underlying CAN P2P network. We have evaluated two different modes:

- **Overlay Multicast QoS Framework with hard QoS**
  In the hard QoS mode, every node has a random QoS class allocated. This QoS class does not change during the entire simulation as the requirements will be fulfilled during the whole simulation. This mode simulates the behavior of the network provider guaranteeing the QoS class requirements to the peers.

- **Overlay Multicast QoS Framework with soft QoS**
  In the soft QoS mode, every node can periodically check if its QoS class requirements fulfilled. If its QoS class requirements are not fulfilled anymore, it needs to change its parent in order to find another parent, which fulfills its QoS class requirements. We have simulated the detection of QoS not fulfilled situations using a timer for each node. This timer then induces the node to search for a new parent by reconnecting to the network. It starts 10 seconds after the node has successfully joined the network and reappears up to 5 times (uniformly distributed from 1 to 5) with an interval of 2 seconds. Therefore, the soft QoS mode can be compared with the evaluation of the stressed mode with CAN.

We have tested both modes with 4 different amounts of QoS classes (32, 64 128 and 256). For all evaluations, we observed that 100% of the multicast paths fulfilled the QoS class requirements of the peers as shown in Fig. 4.6.

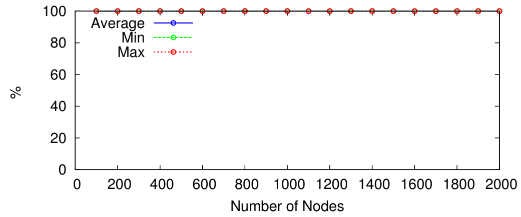
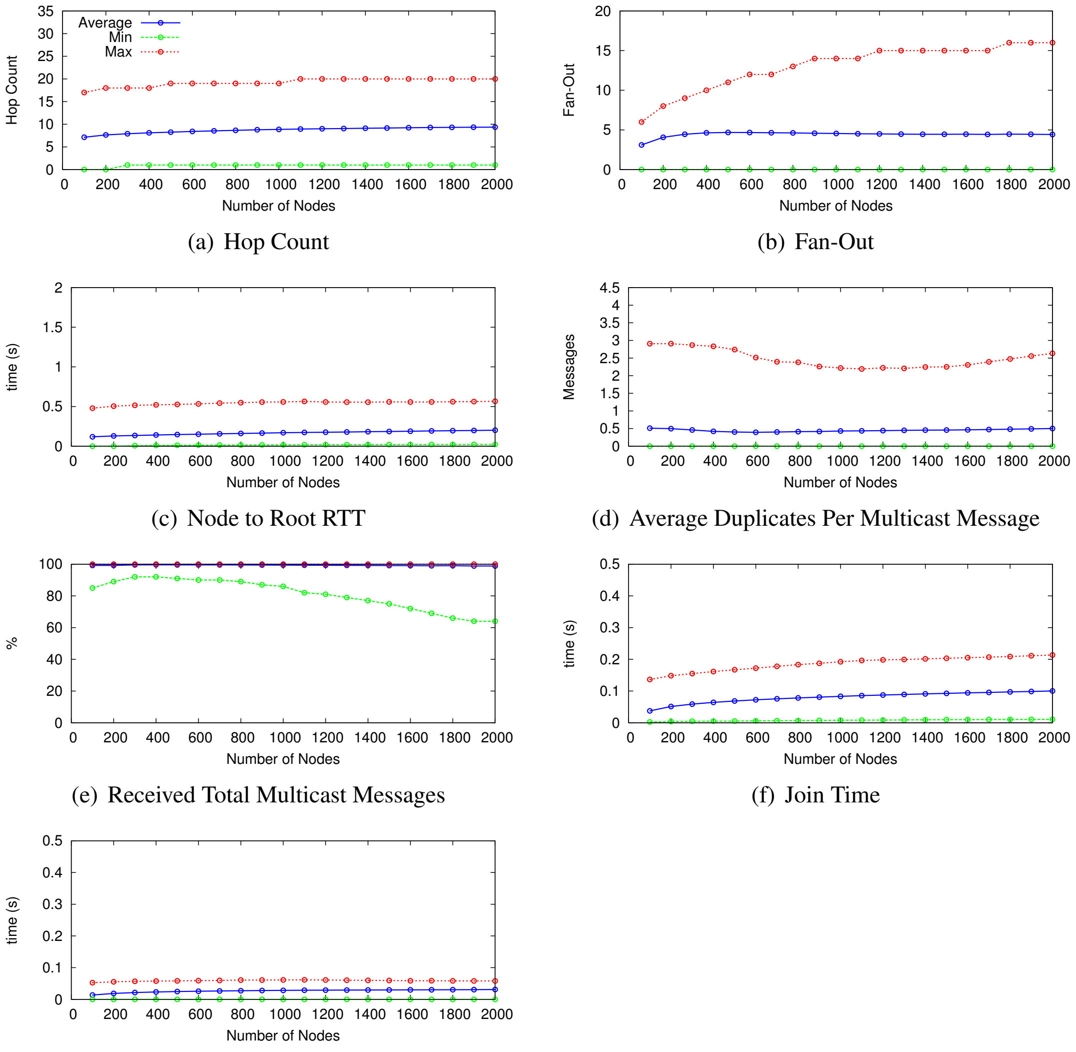


Figure 4.6: Overlay Multicast QoS Framework - QoS Requirement Fulfilling Paths for 32, 64, 128 and 256 QoS Classes

Furthermore, we compare the evaluation metrics of the Overlay Multicast QoS (OM-QoS) Framework applied to CAN with the evaluation results of the native CAN network to determine the overhead created by the OM-QoS Framework. As the OM-QoS Framework creates multiple mini CANs, we need to compare the evaluation results from the OM-QoS Framework with the small network scenario presented in Fig. 4.4 and not with the normal CAN scenario shown in Fig. 4.1. For example, if we have the evaluation results of a OM-QoS Framework with 32 QoS classes and 1500 nodes, we need to compare it with the evaluation results of the native CAN network with 47 ($100/32 = 47$) nodes, as we have on average 47 nodes in each of the 32 mini CANs.

We will only perform hard OM-QoS evaluations with the normal CAN mode and not with the stressed mode. We will compare the soft OM-QoS evaluations with the stressed mode as the nodes rejoin the CAN network, once their QoS is not satisfied anymore, which can be compared with the stressed mode, where the nodes are forced to leave and join.

4.3.1 Hard QoS with 32 QoS Classes



(a) Hop Count

(b) Fan-Out

(c) Node to Root RTT

(d) Average Duplicates Per Multicast Message

(e) Received Total Multicast Messages

(f) Join Time

(g) Leave Time

Figure 4.7: CAN with Overlay Multicast QoS Framework - hard QoS with 32 Classes

In Fig. 4.7, the results for our Overlay Multicast QoS (OM-QoS) Framework with 32 QoS classes with the hard QoS mode are shown.

The hop count in Fig. 4.7(a) has a an average value, which slowly increases from 7 to 9.5. The maximum is also slowly increasing from 17 to 20 hops. The minimum stays again at 0 as the root has 0 hops to itself. In the Figure, this value is rising to 1 as we have removed 0.5% outliers. If we compare these results with the small CAN network scenario shown in Fig. 4.4(a) for 0 to 62.5 number of nodes, we have constantly 5.7 more hops, which is the overhead generated by the OM-QoS Framework. We compare the overlay results with the Figure 4.4(a) for small networks as we have 32 different P2P networks and therefore we have a maximum of 62.5 $(2000/32)$ nodes per CAN network.

In Fig. 4.7(b), the fan-out remains constant around 4.9 and has a maximum value, which rises constantly from 5.9 to 15.5. Comparing these results with the native small CAN network in Fig. 4.4(b), the Overlay Multicast QoS Framework has constantly a higher fan-out of 2.4. Again, this is the overhead generated by the OM-QoS Framework.

Looking at the node to Root RTT in Fig. 4.7(c), it has a slowly rising value from $0.14s$ to $0.20s$. The maximum value is also slowly increasing from $0.49s$ to $0.51s$. Therefore, the RTT behaved as expected as the Hop Count increased also slowly. Comparing this to the native CAN evaluation in Fig. 4.4(c), we can see that the average and the max. value are both higher. This behavior was expected, as we had a higher hop count in Fig. 4.7(a).

The average duplicates per multicast message shown in Fig. 4.7(d) has an average around 0.5 and a maximum value around 2.5. In the native CAN network in Fig. 4.4(d), the average duplicates per multicast message is 0.25 lower. Due to the additional gateway links in the OM-QoS Framework, we have a slightly higher average, as these gateway nodes generate further duplicates in the framework itself.

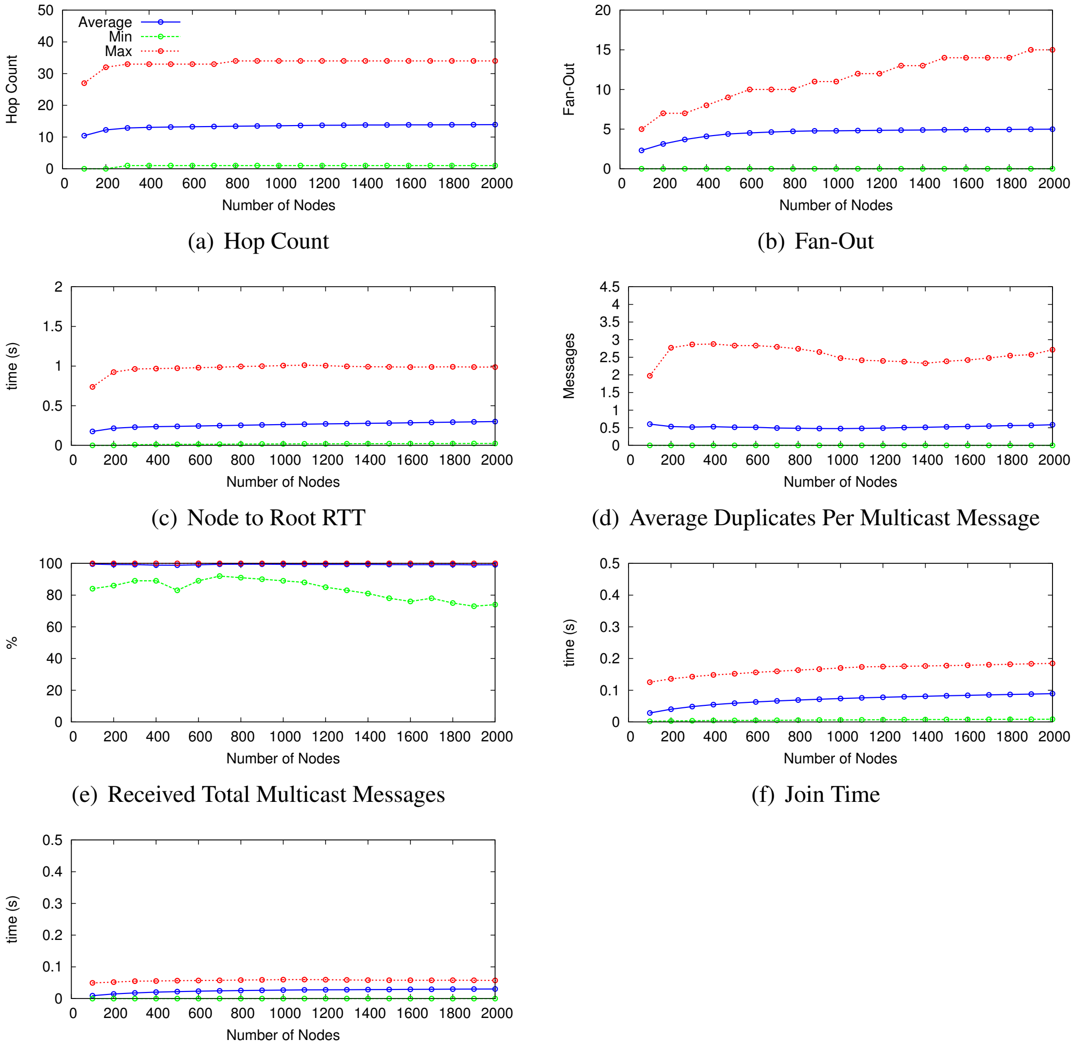
The average percentage of received multicast messages in Fig. 4.7(e) stays constant at 99%, which is equal to the native CAN evaluation in Fig. 4.4(e). Therefore, we have no additional loss caused by the OM-QoS Framework.

The join time is shown in Fig. 4.7(f). The average is slightly increasing from $0.04s$ to $0.1s$ and the maximum value also increases from $0.13s$ to $0.21s$ Comparing these results with the native CAN results shown in Fig. 4.4(f), we can see that the average and the maximum values are almost identical. This behavior was expected, as the OM-QoS Framework does not interfere with the Join process.

The leave time is shown in Fig. 4.7(g) with an average leave time around $0.02s$. Again, this result is equal to the average leave time in the native CAN results in Fig. 4.4(g). However, the maximum value is also constant at $0.05s$ for the OM-QoS Framework but has several peaks up to $1.4s$ for native CAN. This behavior can be explained with the existence of multiple CAN network entities for the OM-QoS Framework. As the nodes leave the network randomly, they interfere with each other if they are in small networks. However, the probability that multiple nodes are leaving at the same time in the same QoS class, and therefore in the same CAN network, is lower with the OM-QoS Framework as we have a dedicated CAN network for each QoS class instead of one network for all nodes in a native CAN.

To sum up, the Overlay Multicast QoS Framework affects the hop count, and therefore the node to root RTT as well. This behavior can be explained with the topology of the OM-QoS Framework, which generates for every QoS class a mini CAN, and which increases the average hop count compared to the native CAN. The fan-out and the average duplicates per multicast message increased only slightly compared to the native CAN, and it has almost no effect on the join and leave time of a node.

4.3.2 Hard QoS with 64 QoS Classes



(a) Hop Count

(b) Fan-Out

(c) Node to Root RTT

(d) Average Duplicates Per Multicast Message

(e) Received Total Multicast Messages

(f) Join Time

(g) Leave Time

Figure 4.8: CAN with Overlay Multicast QoS Framework - hard QoS with 64 Classes

In this Section we present the results for our Overlay Multicast QoS Framework with 64 QoS classes with the hard QoS mode.

Figure 4.8(a) shows the hop count, where the average is slowly increasing from 10.4 to 13.8. If we compare this value with native CAN in Fig. 4.4(a), we have constantly 10.9 more hops, which are generated from the OM-QoS Framework. Compared to the evaluation with 32 QoS classes with the OM-QoS Framework where we had constantly 5.7 more hops than in the native mode, the hop count generated by the OM-QoS Framework has been doubled for twice as much QoS classes.

The node to root RTT in Fig. 4.8(c) has a slightly rising average from $0.17s$ to $0.26s$. Compared to the evaluation with 32 QoS classes presented in Fig. 4.7(c), it is slightly higher, but behaves as expected if we take the higher hop count into account.

The fan-out 4.8(b), the average duplicates per multicast message 4.8(d), the percentage of received multicast messages̃reffig:can2D64QoSOF0R0.5O-ReceivedTotalMulticastMessagesP, the join time 4.8(f), as well as the leave time 4.8(g) did not show a significant change compared to the evaluation with 32 QoS classes.

To sum up, the framework behaved as expected. With twice as much QoS classes, the hop count ratio was doubled, and therefore the node to root RTT increased.

4.3.3 Hard QoS with 128 QoS Classes

In this Section we present the results for our Overlay Multicast QoS Framework with 128 QoS classes with the hard QoS mode. As in the previous Section, only the hop count and the node to root RTT will be discussed in details, as the other metrics were equal to the evaluations with 32 QoS classes (presented in Section 4.3.1).

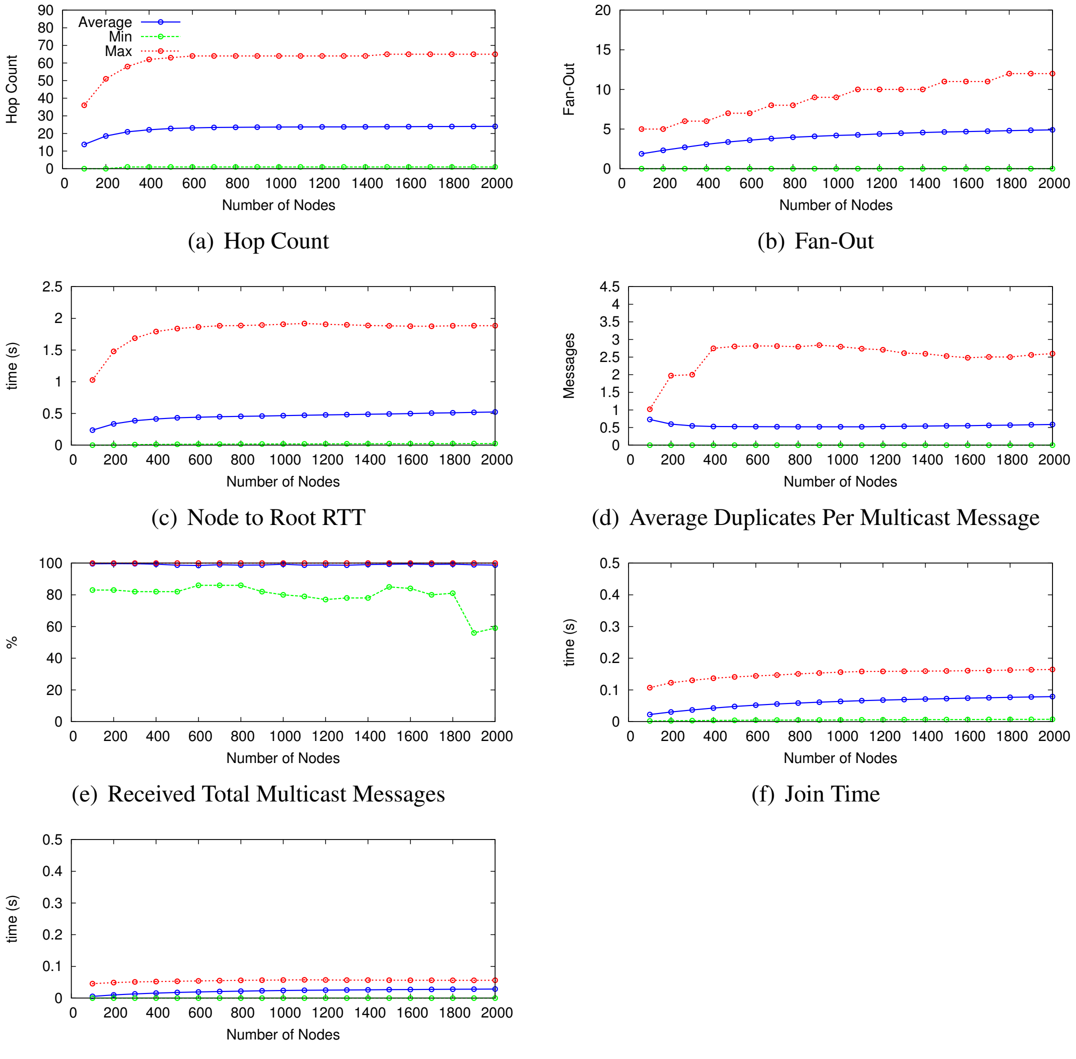
The hop count shown in Fig. 4.9(a) rises in average from 13.7 to 24.0 and the maximum value rises from 36 to 65. If we compare the average with native CAN in Fig. 4.4(a), the average hop count value for the Overlay Multicast QoS Framework is between 12.8 and 21.2.

The node to root RTT shown in Fig. 4.9(c) rises according to the hop count in average from $0.24s$ to $0.52s$ and the maximum value rises from $1.02s$ to $1.80s$. The average RTT is still acceptable and proves that the implementation is still scalable. However, looking at the maximum value, some problems might occur with time critical applications. A solution to solve this problem, would be to add more additional path optimized gateway links to the Overlay Multicast QoS Framework to get a lower hop count and therefore a lower node to root RTT.

4.3.4 Hard QoS with 256 QoS Classes

In this Section we present the results for or our Overlay Multicast QoS Framework with 256 QoS classes with the hard QoS mode. The main goal was to test how resilient the implementation is to a high number of QoS classes. Especially we wanted to analyze how the hop count and the node to root RTT behave in such a scenario. All other metrics behaved as expected, equal to the evaluations with 32 QoS classes (presented in Section 4.3.1).

The hop count is presented in Fig. 4.10(a) and has an average value increasing from 15.7 to 43.60 and a maximum value increasing from 45 to 127. If we compare the average values with the native CAN values in Fig. 4.4(a), the Overlay Multicast QoS hop count value is between 14.8 and 40.7.



(a) Hop Count

(b) Fan-Out

(c) Node to Root RTT

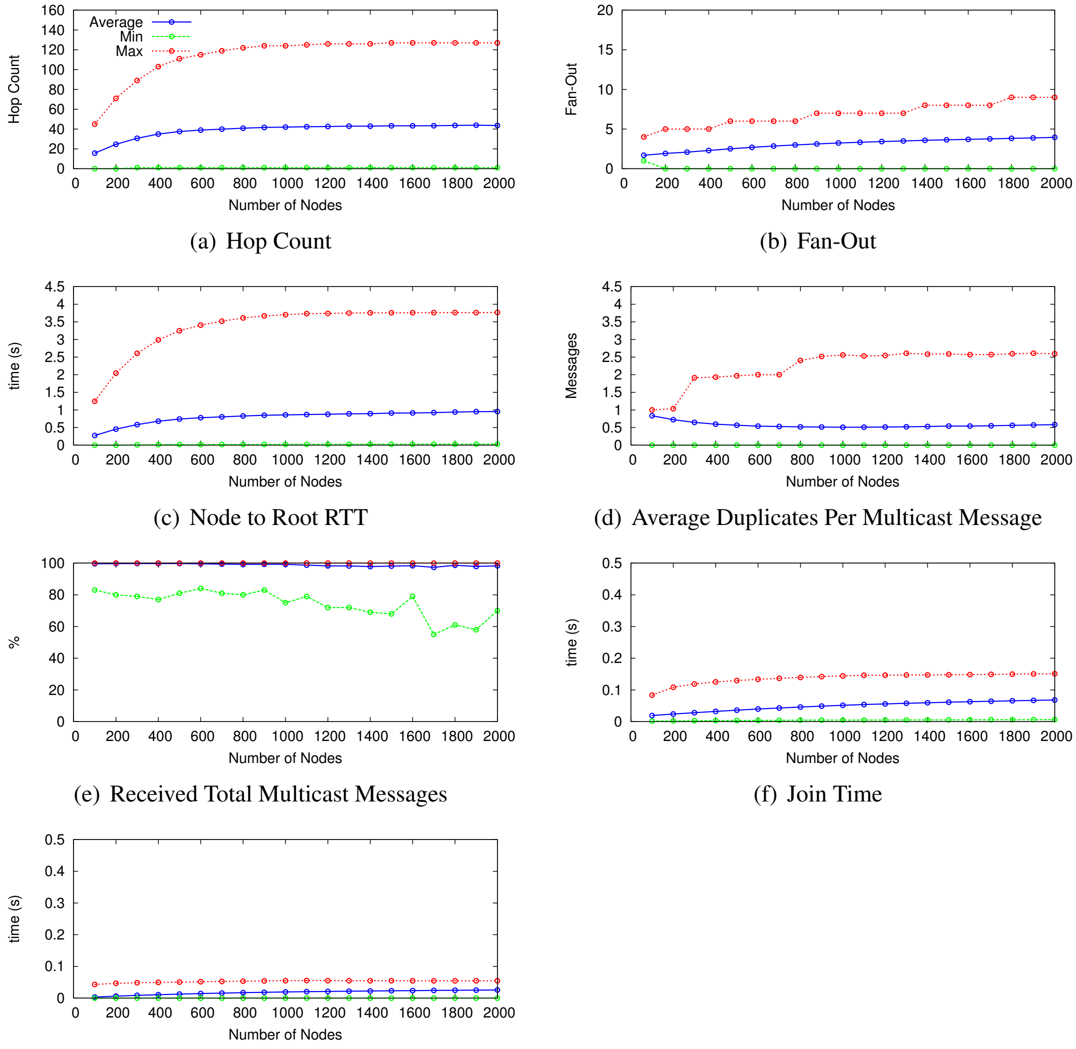
(d) Average Duplicates Per Multicast Message

(e) Received Total Multicast Messages

(f) Join Time

(g) Leave Time

Figure 4.9: CAN with Overlay Multicast QoS Framework - hard QoS with 128 Classes



(a) Hop Count

(b) Fan-Out

(c) Node to Root RTT

(d) Average Duplicates Per Multicast Message

(e) Received Total Multicast Messages

(f) Join Time

(g) Leave Time

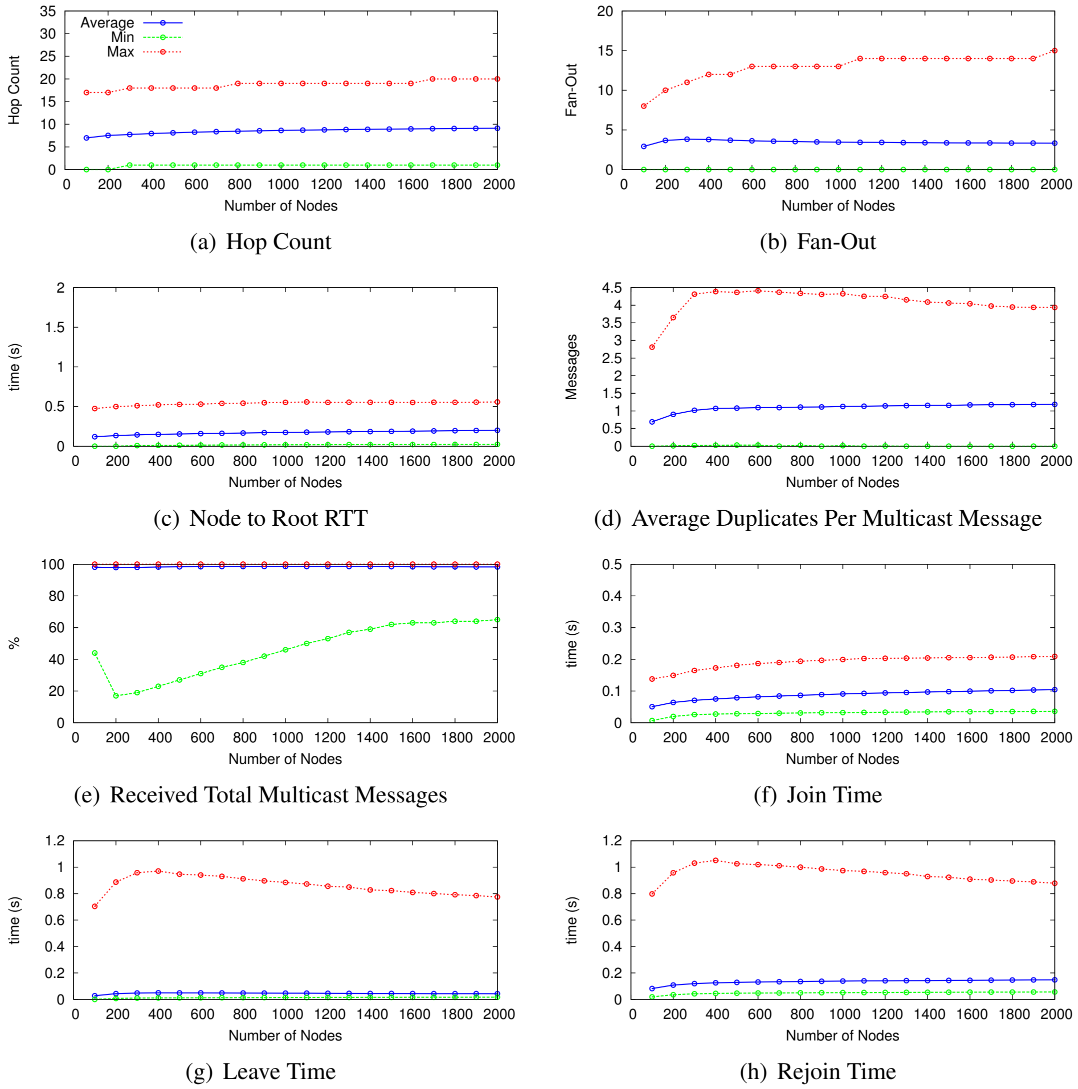
Figure 4.10: CAN with Overlay Multicast QoS Framework - hard QoS with 256 Classes

The node to root RTT is presented in Fig. 4.10(c) and has an average value increasing from $0.27s$ to $0.95s$ and a maximum value rising from $1.2s$ to $3.76s$. Both values rise according to the hop count.

To sum up, we have shown that even with a high number of QoS classes, the OM-QoS Framework is still working. However, the hop count, and therefore the node to root RTT will rise accordingly if no additional gateways are introduced. Depending on the application, a high hop count may be even irrelevant, but QoS requirements split in 256 differentiated QoS classes is also not very likely to happen.

4.3.5 Soft QoS with 32 QoS Classes

In Fig. 4.11, the results for our Overlay Multicast QoS Framework with 32 QoS classes with the soft QoS mode are shown. As we use CAN as our underlying P2P network, and therefore can only rejoin to the network if a node's QoS is not fulfilled, we can compare the results below also with the native CAN stressed mode.



(a) Hop Count

(b) Fan-Out

(c) Node to Root RTT

(d) Average Duplicates Per Multicast Message

(e) Received Total Multicast Messages

(f) Join Time

(g) Leave Time

(h) Rejoin Time

Figure 4.11: CAN with Overlay Multicast QoS Framework - soft QoS with 32 Classes

In Fig. 4.11(a), the hop count of the evaluation is shown. The average is slowly increasing from 6.9 to 9.1 and the maximum is also slowly incrementally increasing from 17 to 20. If we compare these results with the hard QoS results in Fig. 4.7(a), they are almost equal. In Section 4.2.3 we compared the stressed mode of the native CAN with the native CAN normal mode and discovered that the hop count in the stressed mode has a 17% lower average. However, we have not seen the same effect with the comparison of hard and soft QoS mode with the OM-QoS Framework. Altough the average hop count for the soft QoS mode and hard QoS mode seems almost equal, the average hop count for soft QoS mode is somewhat smaller than for the hard QoS mode. The reason that the average hop count is not significantly lower in the soft QoS mode is that the OM-QoS Framework part of the hop count is between 60% and 81.5%, and therefore the hop count reduction effect with rejoining nodes within CAN is very small.

The fan-out is shown in Fig. 4.11(b) and has a constant average value around 3.1 and a maximum value increasing from 8 to 15. Comparing these results with the hard QoS mode in Fig. 4.7(b), we have a lower average value but the maximum value behaved the same. The lower average value can be explained with the rejoin process and the smaller CanKeySpaces, therefore the effect was not seen at the comparison of native CAN normal mode and native CAN stressed mode in Section 4.2.3.

In Fig. 4.11(c), the node to root RTT is presented. The average value rises from $0.08s$ to $0.2s$ and the maximum value rises slowly from $0.47s$ to $0.55s$. Both values behaved as expected when looking at the hop count.

The average duplicates per multicast message is shown in Fig. 4.11(d), and has an average, which is constant around 1.18. Comparing this to the results for hard QoS as shown in Fig. 4.11(d), the average is 0.5 higher. This is due to the same effect that was shown at the comparison of native CAN normal mode with native CAN stressed mode in Section 4.2.3. It can be again explained with the nature of CAN and the appearance of temporary CanKeySpaces, which inhibit an efficient duplicate suppression algorithm. The maximum value of average duplicates per multicast message rises from 2.8 duplicates for 100 nodes to 4.4 duplicates for 300 nodes and is then slowly decreasing to 4 duplicates for 2000 nodes. The strong rise in the beginning can be explained with the low number of nodes per QoS mini CAN entity. Therefore, the number of duplicates is rising until the number of nodes in a mini CAN has reached 10 nodes ($10 \cong 300$ nodes / 32 mini QoS CAN entities) and is then slowly decreasing as the multicast tree in each CAN can be better distributed.

Figure 4.11(e) shows the percentage of received multicast messages. The average remains constantly around 98.1%. The minimum decreases to 19% for 200 nodes and increases then constantly back to 65%. Compared to the hard QoS results presented in Fig. 4.7(e), the average is slightly lower but still good. If we look at the minimum value, the behavior is similar as in the native CAN stressed mode and can be explained with the rejoin process. If nodes join and leave the network due to non satisfied QoS, multicast messages can get lost. However, this happens only to a few number of nodes as the average confirms. The drop of the minimum value from 100 to 200 nodes can be again explained with the low number of nodes in each QoS mini CAN and the appearance of temporary CanKeySpaces. The temporary CanKeySpaces do normally not appear if only a few nodes are connected to the network, since the CanKeySpaces can easily be re-merged as explained in Section 3.2.4.

The join time is shown in Fig. 4.11(f) and has an average slowly increasing from $0.05s$ to $0.10s$ and a maximum equally increasing from $0.13s$ to $0.21s$. Compared to the hard QoS mode in Fig. 4.7(f), both values behave equally.

In Fig. 4.11(g), the leave time is shown. The average rises constantly from $0.02s$ to $0.04s$ and the maximum value increases from $0.7s$ to $0.98s$ and then slowly decreases to $0.77s$. Compared to the hard QoS mode presented in Fig. 4.7(g), the average behaves equally, however the maximum value has increased significantly. This is the same behavior we detected when comparing native CAN normal mode with native CAN stressed mode and can be explained with the appearance of temporary CanKeySpaces, which leads to more neighbors for a leaving node, and therefore to more interference between multiple leaving nodes. The increase of the maximum leave time from 100 nodes to 300 nodes can be explained with the interference of nodes when they decide to leave. If we have only a few nodes distributed over every mini QoS CAN, the probability that two nodes in the same mini CAN leave at the same time is low, and therefore they cannot interfere with each other. If more nodes join the network, this probability increases. Once there are more than 10 nodes in once mini CAN, the probability that two nodes leave at the same time in the same QoS mini CAN and interfere with each other is starting to decrease, and therefore the maximum leave time is also decreasing.

In the last Figure 4.11(h), the rejoin time is shown. It describes how long it takes for a node to get a new parent, which fulfills its QoS class. It the old parent does not satisfy its QoS class anymore. The average value is slowly rising from $0.08s$ to $0.147s$ and the maximum value behaves almost equally as the leave time because of the same reasons as described above.

To sum up, we have shown that our Overlay Multicast QoS Framework does also work with the soft QoS approach. All observed values behaved as expected and we especially had an excellent average percentage of received multicast messages. Furthermore, we have seen that the CAN rejoin effect described in Section 4.2.3 does not have a major impact on hop count and node to root RTT with the soft QoS approach, as the major influence on these values come from the OM-QoS Framework.

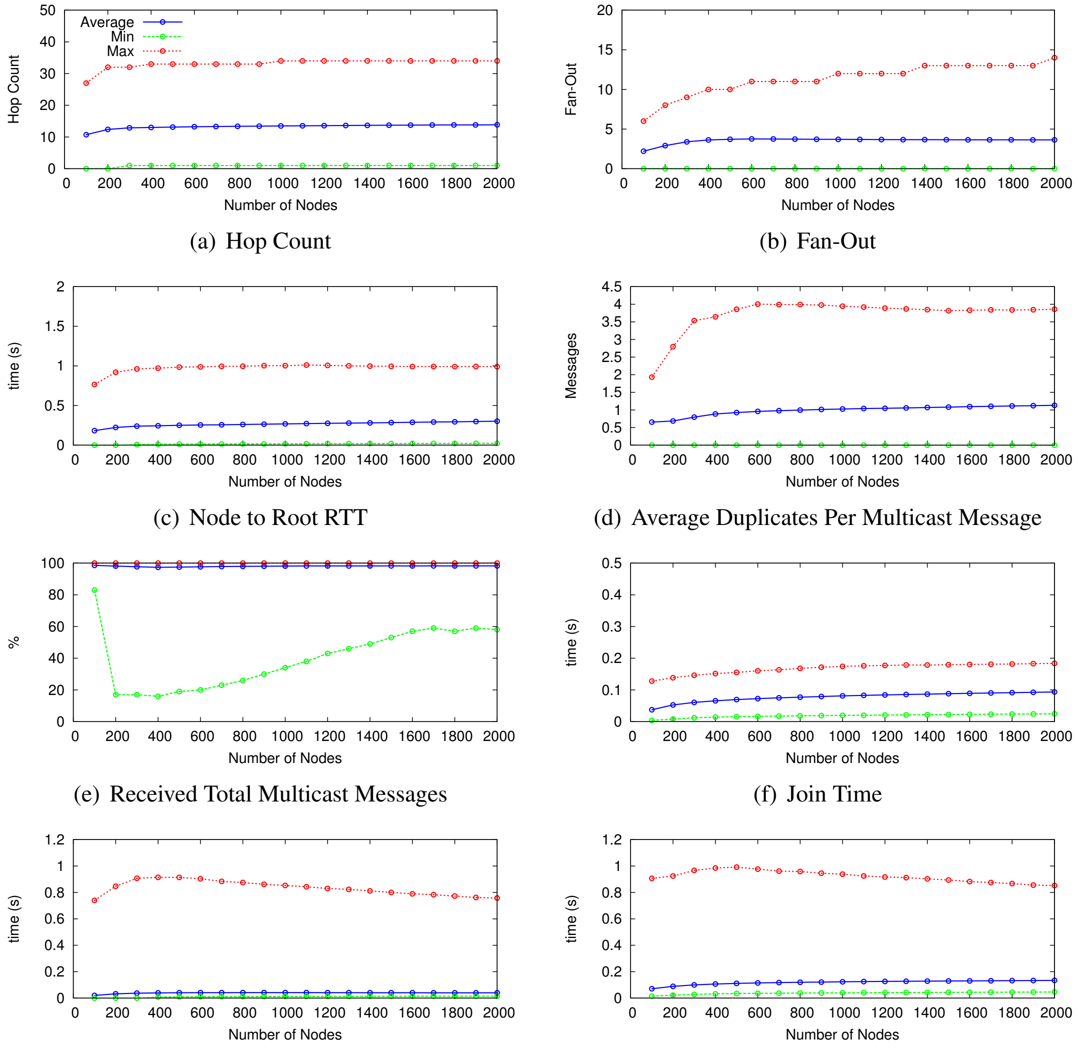
4.3.6 Soft QoS with 64 QoS Classes

In this Section, we present the results for our Overlay Multicast QoS Framework with 64 QoS classes with the soft QoS mode.

The hop count in Fig. 4.12(a) has an average value rising slowly from 10.7 7o 13.8. The maximum value rises stronger, from 27 to 34 hops. Compared to the values from the soft QoS mode in Fig. 4.8(a), the values are almost even. The hop count reduction effect, which we have described in Section 4.2.3 cannot be seen. As explained in the previous Section, the OM-QoS Framework part of the hop count value is high and increases with the number of QoS classes, and therefore the hop count reduction effect gets smaller.

Figure 4.12(c) shows the node to root RTT. It has an average value, which is slowly increasing from $0.19s$ to $0.30s$ and a maximum value, which increases from $0.76s$ to $0.99s$. Both values increased according to the hop count, and therefore as expected.

The fan-out 4.12(b), the average duplicates per multicast message 4.12(d), the percentage of received multicast messages 4.8(e), the join time 4.12(f), as well as the leave time 4.12(g) did not show a significant change compared to the evaluation with 32 QoS classes.



(a) Hop Count

(b) Fan-Out

(c) Node to Root RTT

(d) Average Duplicates Per Multicast Message

(e) Received Total Multicast Messages

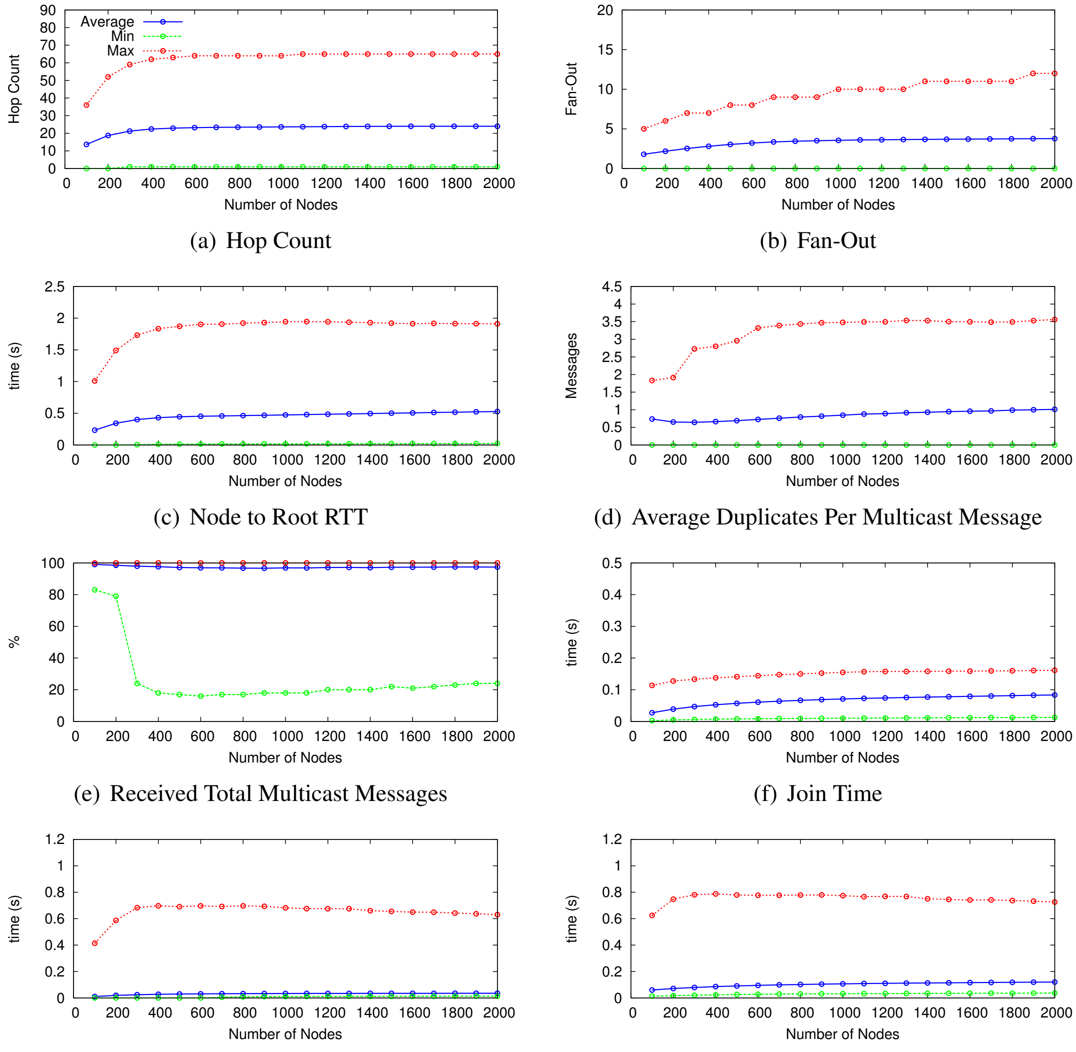
(f) Join Time

(g) Leave Time

(h) Rejoin Time

Figure 4.12: CAN with Overlay Multicast QoS Framework - soft QoS with 64 Classes

4.3.7 Soft QoS with 128 QoS Classes



(a) Hop Count

(b) Fan-Out

(c) Node to Root RTT

(d) Average Duplicates Per Multicast Message

(e) Received Total Multicast Messages

(f) Join Time

(g) Leave Time

(h) Rejoin Time

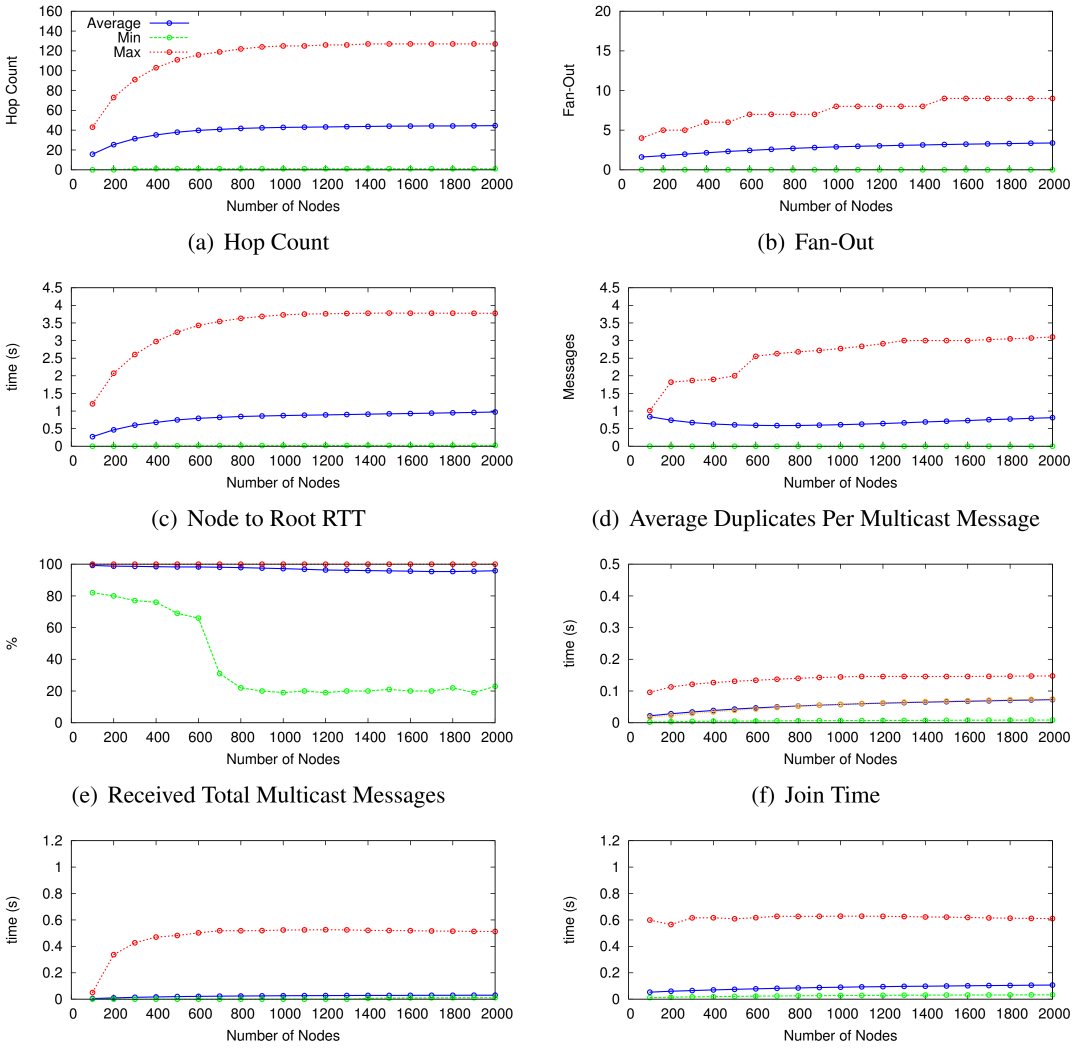
Figure 4.13: CAN with Overlay Multicast QoS Framework - soft QoS with 128 Classes

In this Section we present the results for our Overlay Multicast QoS Framework with 128 QoS classes with the soft QoS mode. As we have seen in the previous Section, only the hop count and the node to root RTT are influenced by the number of QoS classes and all other metrics are equal to the evaluations with 32 QoS classes in Fig. 4.11.

The hop count is presented in Fig. 4.13(a) and has an average value rising from 13.7 to 21.4 and an maximum value rising from 36 to 65. Compared to the hard QoS mode presented in Fig. 4.9(a), the values are the same. Again, the hop count reduction effect as explained Section 4.2.3 cannot be seen, as the OM-QoS Framework part of the hop count value is high and increases with the number of QoS classes, and therefore the cop count reduction effect gets smaller.

In Fig. 4.13(c), the node to root RTT is shown. It has an average value, which rises from $0.23s$ to $0.52s$ and a maximum value, which rises from $1.0s$ to $1.9s$. Both values grow according to the hop count increase.

4.3.8 Soft QoS with 256 QoS Classes



(a) Hop Count

(b) Fan-Out

(c) Node to Root RTT

(d) Average Duplicates Per Multicast Message

(e) Received Total Multicast Messages

(f) Join Time

(g) Leave Time

(h) Rejoin Time

Figure 4.14: CAN with Overlay Multicast QoS Framework - soft QoS with 256 Classes

In the following Section, we will present the results for the Overlay Multicast QoS (OM-QoS) Framework with 256 QoS classes using the soft QoS mode. Like in the previous Sections, only the hop count and the node to root RTT values are shown, as these values are influenced by the number of QoS classes and all other metrics are equal to the evaluations with 32 QoS classes as presented in Fig. 4.11.

The hop count is presented in Fig. 4.14(a) and has an average value, which rises logarithmically with an upper bound of 44.5. The maximum value rises equally with an upper bound of 127. Comparing these values with the hard QoS mode presented in Fig. 4.10(a), the values increased identically. This behavior is in line with our previous conclusion in Section 4.3.5.

The node to root RTT rises, as expected, according to the hop count with an average value, which rises logarithmically with an upper bound of $0.97s$, and also the maximum value rises logarithmically with an upper bound of $3.7s$.

To sum up, we can see that the OM-QoS Framework also works with soft QoS for large number of QoS classes. However, the high hop count, and therefore the high node to root RTT can cause problems for applications, which are RTT sensitive, but as explained before, QoS requirements split in 256 differentiated classes are not very likely to happen.

4.4 Overlay Multicast QoS Framework with NICE

In the next two Sections, we present the results for our Overlay Multicast QoS (OM-QoS) Framework with the underlying P2P network NICE (described in Section 2.2.5). We have implemented our OM-QoS Framework using a NICE implementation [20], as we wanted to show that the framework cannot only be used with CAN, but also with other P2P networks.

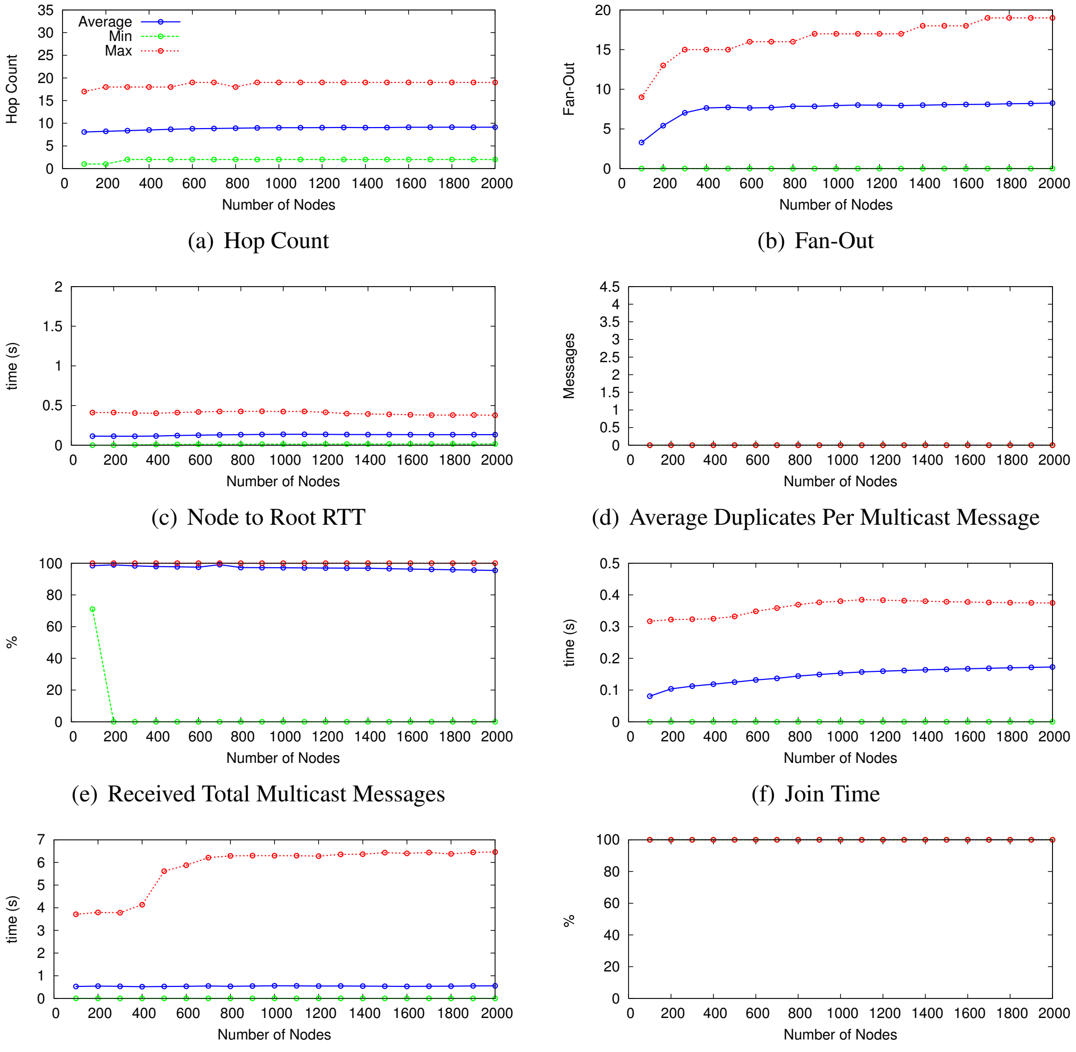
4.4.1 Hard QoS with 32 QoS Classes

Figure 4.15 shows the evaluation results from our OM-QoS Framework with NICE as underlying P2P network and with 32 QoS classes.

The hop count or path length of a node to the root is presented in Fig. 4.15(a) and shows an average value slowly rising from 8.0 to 9.2. The maximum value is rising equally from 17 to 19. The minimum value should stay at 0 as the root has always 0 as path length to itself, however the Figure shows a higher minimum value due to the outlier removal. If we compare these results with the OM-QoS Framework with CAN in Fig. 4.7(a), the results are comparable although the results with CAN are slightly higher.

In Figure 4.15(b), the fan-out is presented and shows an average value, which rises logarithmically with an upper bound of 8.2 and a maximum value, which rises from 9 to 19. If we compare these values with the results from the QM-QoS Framework with CAN in Fig. 4.15(b), we notice that the average fan-out with NICE is significantly higher as with the CAN network. This can be explained with the nature of the NICE framework, where every root has a significantly higher fan-out as the rest of the nodes in the framework. With the OM-QoS Framework, we have now such a root node in every QoS class, which increases the average fan-out.

The node to root RTT is shown in Fig. 4.15(c), and has an average value, which stays constant around $0.12s$ and a maximum value, which also stays constant around $0.43s$. Compared to the results with the OM-QoS Framework with CAN, the values are significantly lower. This



(a) Hop Count

(b) Fan-Out

(c) Node to Root RTT

(d) Average Duplicates Per Multicast Message

(e) Received Total Multicast Messages

(f) Join Time

(g) Leave Time

(h) QoS Requirements Fulfilled Paths for 32 QoS Classes

Figure 4.15: NICE with Overlay Multicast QoS Framework - hard QoS with 32 Classes

behavior was expected as NICE is optimizing its node to root RTT within its network.

Figure 4.15(d) shows that the NICE framework and the OM-QoS Framework do not have any duplicates per multicat message.

In Figure 4.15(d) the percentage of received multicast messages is shown. The average value is slowly decreasing from 98.4% to 95.0%. Comparing this value with the OM-QoS Framework results presented in Fig. 4.7(d), the average values with NICE are lower as with the CAN network. Again, this can be explained with the nature of NICE, where clusters need to be reorganized once they exceed a certain cluster size. Due to this reorganization process, some messages can get lost. This is also the explanation for the low minimum value.

The join time is shown in Fig. 4.15(f) and has an average value, which slowly rises from $0.08s$ to $0.17s$, and a maximum, which slowly rises from $0.32s$ to $0.37s$. Compared to the OM-QoS Framework with CAN results in Fig. 4.7(f), the values are lower. This is no surprise and can be explained with the simpler join process of the NICE network compared to the CAN network.

In Figure 4.15(g) the leave time is shown, where the average value stays at $0.05s$. Its maximum value is constant at $3.7s$ until 300 nodes and is then rising up to $6.45s$ for 600 nodes, where it stays constantly. This high leave time can be explained with the cluster leader handover at times where many nodes leave and with the increased complexity of this handover if the number of NICE layers increases.

The last Figure 4.15(h) shows that 100% of the multicast paths fulfilled the QoS class requirements of the peers.

To sum up, we have shown that our Overlay QoS Framework works not only with CAN but also with NICE as underlying P2P network as we also have 100% multicast paths fulfilling the QoS requirements with NICE. The hop count and the fan-out behaved similar in both networks, although NICE has slightly better results.

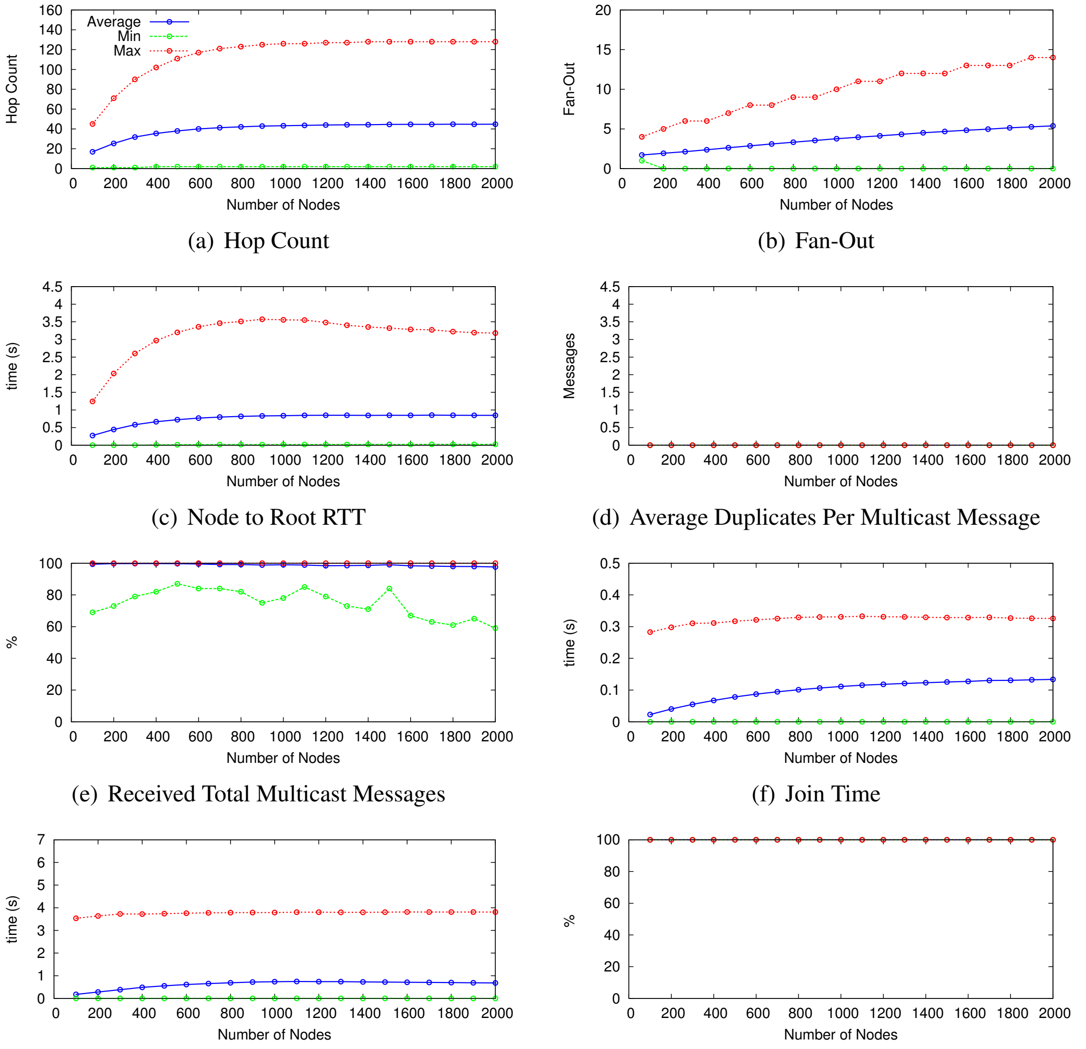
4.4.2 Hard QoS with 256 QoS Classes

In Figure 4.16 the evaluation results from our OM-QoS Framework with NICE as underlying P2P network and with 256 QoS classes are presented. Compared to the previous Section, the influence of the OM-QoS Framework on the evaluation results has increased as there are only a few nodes in each QoS NICE entity.

The hop count is presented in Fig. 4.16(a) and shows an average value, which rises logarithmically with an upper bound of 44.7 and the maximum rises equally with an upper bound of 128. If we compare these values with the results from OM-QoS Framework with CAN in Fig. 4.10(a), we have similar results although the values from NICE are slightly higher as the CAN results.

In Fig. 4.16(b), the fan-out is shown. The average value is steadily rising from 1.72 to 5.3 and the maximum is rising stronger from 4 to 14. Compared to the results with the OM-QoS Framework with CAN in Fig. 4.10(b), we can see that the CAN fan-out is slightly lower as the NICE one. As in the previous Section, this behavior can be explained with the high fan-out of the roots in each QoS NICE entity.

The node to root RTT is shown in Fig. 4.16(c), and shows an average value, which slowly rises from $0.27s$ to $0.86s$, and maximum value, which rises accordingly from $1.25s$ to $3.26s$.



(a) Hop Count

(b) Fan-Out

(c) Node to Root RTT

(d) Average Duplicates Per Multicast Message

(e) Received Total Multicast Messages

(f) Join Time

(g) Leave Time

(h) QoS Requirements Fulfilled Paths for 256 QoS Classes

Figure 4.16: NICE with Overlay Multicast QoS Framework - hard QoS with 256 Classes

Comparing these results with the values from the OM-QoS Framework evaluations with CAN in Fig. 4.10(c), we have again a better RTT with the NICE network, as NICE is optimizing the node to root RTT within its network.

Figure 4.16(d) shows again that the NICE framework and the OM-QoS Framework do not have any duplicates per multicat message.

In Figure 4.16(e), the percentage of received multicast messages is presented. The average value is slowly decreasing from 99.3% to 97.6%. Compared to the results with OM-QoS Framework with CAN in Fig. 4.10(e), we have similar results, and compared to the results with OM-QoS Framework with NICE and 32 QoS classes in Fig. 4.15(e), the results are significantly better. This can be explained with the distribution of the joining and leaving peers over the individual QoS NICE networks. As we have now 256 QoS classes compared to 32 QoS classes, we have less interference from the leaving or joining nodes with each other and therefore less messages, which get lost. The minimum value of the percentage of received multicast messages shows some serrated behavior. This can be explained with the uniform distribution of nodes and the size of clusters. Depending on the cluster size, nodes interfere less with each other if they are in the process of leaving, and therefore less multicast messages get lost.

The join time is shown in Fig. 4.16(f) and has an average value, which is increasing from $0.02s$ to $0.13s$ and a maximum value, which increases equally from $0.28s$ to $0.32s$. If we compare these values with the OM-QoS Framework CAN results in Fig. 4.10(f), they are almost identical.

In Fig. 4.16(g), the leave time is shown with an average value rising from $0.17s$ to $0.68s$ and a maximum value, which is rising equally from $3.5s$ to $3.8s$. Comparing these values with the results from OM-QoS Framework with CAN in Fig. 4.10(g), the values are significantly higher than with the CAN network. This can be explained with the leave process of the NICE network. If we compare the values with the results from the OM-QoS Framework with NICE and 32 QoS classes in Fig. 4.15(g), we have significantly lower values. This can be be again explained with the distribution of the leaving peers over the individual QoS NICE networks, and as we have now 256 QoS classes compared to 32 QoS classes, we have less interference from the leaving nodes with each other and therefore a lower leave time.

The last Figure 4.16(h) shows again that 100% of the multicast paths fulfilled the QoS class requirements of the peers.

To sum up, we have again shown that the Overlay Multicast QoS Framework works not only with CAN as an underlying P2P network, but with different P2P networks as well, and we have again 100% of the multicast paths fulfilling the QoS class requirements.

Chapter 5

Conclusion

The multicast paradigm is one of the most efficient methods to distribute data to a group of receivers. Unfortunately, IP Multicast, the implementation of the multicast paradigm for IP networks, is not widely spread in today's Internet, due to several reasons. Overlay Multicast or Application Layer Multicast eludes this problem and allows the usage of multicast in the Internet. Most of the Overlay Multicast implementations have a peer-to-peer structure as an underlying topology. An example of such P2P networks are CAN, NICE or CHORD.

We implemented the Content Addressable Network (CAN) within Omnet++. During the implementation, we discovered that the multicast implementation of CAN needed to be slightly adapted. To enable multicast within CAN, a group specific mini CAN is build, where the multicast messages are flooded through the network. To reduce the duplicates generated by the flooding, multiple duplicate suppression functions were introduced. However, the handling of multicast messages with temporary CanKeySpaces was not specified and we had to extend the suppression functionality. Furthermore, we introduced several handshake algorithms for the join and leave functionality of the nodes to get a better and more stable performance.

The evaluation results from the implemented native CAN network, presented in Section 4.2, showed that our implementation is working as expected. The hop count and the node to root RTT had acceptable results, although the maximum hop count rises quite strong for large number of nodes in the network. A solution to reduce this large hop count would be to increase the number of dimension for the CanKeySpaces (e.g. three dimensional CanKeySpace results in Section4.2.2), however this leads to an unacceptable high fan-out. As we valued the importance of a low fan-out higher than the maximum hop count, we evaluated all further evaluations only with a two-dimensional CanKeySpace. A further improvement for the CAN network could be to reduce the average duplicates per multicast message, by finding a duplicate suppression algorithm, which could further reduce this value.

An interesting result for the native CAN network was that if we force the nodes to rejoin after a certain amount of time, the network gets more equally distributed, and therefore the hop count, the node to root RTT, and the fan-out showed better results.

The CAN network has no native Quality of Service functionality. We introduced a Overlay Multicast QoS (OM-QoS) Framework solution, which should enable this functionality within CAN and for almost all P2P networks. The OM-QoS Framework builds for each QoS class a separate P2P network and connects them over gateway nodes with each other. The purpose of this chain design is to build a QoS tree, which has from the root to the different leafs a monotonically decreasing QoS path. Furthermore, the OM-QoS Framework is independent, which means that it intercepts the messages from the underlying P2P network to establish its gateway links. Only a few adaptions have to be made within the underlying P2P network.

The evaluation results for our OM-QoS Framework, presented in Section 4.3 (CAN) and 4.4 (NICE) showed that our implementation behaved as expected. We were able to achieve that 100% of the multicast paths fulfilled the QoS class requirements of the peers. We achieved this result with hard QoS, where the QoS class does not change during the entire simulation as the network provider guarantees the QoS class requirements to the peer, and with soft QoS, where the nodes can periodically check if their QoS class requirements are still fulfilled and can react depending on the result by finding a new parent to support the QoS requirements if the old parent fails to do so.

The hop count, the fan-out and the node to root RTT behaved all as expected, although the OM-QoS Framework has only an impact on the hop count and the node to root RTT. This impact can be explained with the chain design of the OM-QoS Framework, which we tried to reduce by introducing additional gateway links. To further reduce the OM-QoS Framework impact on the hop count, the additional gateway link algorithm could be improved.

For future work, our implementation could be moved into PlanetLab to test the CAN and the OM-QoS Framework in a real-time environment. As we have used a modular implementation approach, the transformation from Omnet++ to a real-world implementation should be feasible.

To sum up, we have presented a valid and stable solution to provide QoS guarantees to different kinds of P2P networks by using the Overlay Multicast QoS Framework by implementing only a few adaptions to the underlying P2P network.

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] 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.

[2] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, Aug. 2001, pp. 161–172.

[3] 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/

[4] 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.

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

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes," *Communications Surveys & Tutorials, IEEE*, pp. 72–93, Jun. 2005. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1528337

[12] 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

[13] 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.

[14] M. Brogle, D. Milic, and T. Braun, "Quality of service for peer-to-peer based networked virtual environments," in *ICPADS '08: Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*. Washington, DC, USA: IEEE Computer Society, Dec. 2008, pp. 847–852.

[15] 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.

[16] S. Ratnasamy, M. Handley, R. M. Karp, and S. Shenker, "Application-level multicast using content-addressable networks," in *NGC '01: Proceedings of the Third International COST264 Workshop on Networked Group Communication*. London, UK: Springer-Verlag, Nov. 2001, pp. 14–29. [Online]. Available: http://portal.acm.org/citation.cfm?id=648089.747491

[17] M. Castro, M. B. Jones, A. M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman, "An evaluation of scalable application-level multicast built using peer-to-peer overlays," in *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*, vol. 2, Apr. 2003, pp. 1510–1520 vol.2. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1208986

[18] 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.

[19] 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.

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

Erklärung

gemäss Art. 28 Abs. 2 RSL 05

Name/Vorname: Bettosini Luca Carlo

Matrikelnummer: 02-112-100

Studiengang: Informatik

Bachelor □ Master ☒ Dissertation □

Titel der Arbeit: Quality of Service for Overlay Multicast Content Addressable Network (CAN)

LeiterIn der Arbeit: Prof. Dr. Torsten Braun

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe o des Gesetztes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Bern, 29.06.2009
Ort/Datum

L. Bettosini
Unterschrift