OM-QoS: Quality of Service for Overlay Multicast applied to the NICE Protocol

Bachelorarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Sebastian Barthlomé
2009

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

Contents

List of Figures

List of Tables

Acknowledgment

I would like to thank my supervising tutor, Marc Brogle for being very helpful and patient during my work. A further thank is dedicated to the members of the RVS group, who supported me with helpful advices and provided a comfortable environment to me. Especially Dragan Milic who provided with Marc the P2P Overlay Framework, which simplified the implementation of the NICE protocol and the QoS mechanisms. I would also like to mention the Master students Luca Bettosini and Andreas Rüttimann who helped me anytime I had problems. At last, a special thank goes to my girlfriend, Chantal Pfaffen for attentively reading through my Bachelor thesis.

Chapter 1

Summary

The goal of this Bachelor thesis is to implement Overlay Multicast Quality of Service (OM-QoS) mechanisms for the NICE protocol. NICE is a structured Peer-to-Peer (P2P) network using a tree structure running as an Overlay Network protocol on top of a physical network. NICE provides Application Layer Multicast (ALM) and can therefore be used to introduce multicast functionality in network infrastructures where native IP-Multicast is not deployed. Additionally, the implementation of QoS support in NICE is also done. There are two approaches of QoS support: The protocol-dependent approach and the protocol-independent approach. In the first approach, the cluster leader selection in the NICE protocol is modified to support QoS. In contrast to the first approach, the second approach does not include a modification of the NICE protocol but uses dedicated NICE networks to provide QoS support. These dedicated NICE networks are arranged in layers, which are interconnected. Therefore, the protocol-independent approach is called "Layered OM-QoS Framework". To be able to support QoS, we introduce the term "QoS class", which is an abstract model of QoS requirements like bandwith, hardware resources, uptime, etc., or a combination of them. Such a QoS class is represented by an integer, which allows to order the QoS classes. This is important since both of the QoS approaches use that QoS classes to build the tree structure according to it.

Another extension that we implemented is the End-to-End (E2E)-Delay Optimization. By the term E2E-Delay we mean the Round Trip Time (RTT) from any node within the NICE network to the root node. Such an optimization is interesting regarding real-time multicast applications such as audio- / video multicasting.

As a simulation environment, we used the OMNeT++ v3.3 network simulator, which is written in C++. It is component- and message based, modular and open-architecture. OMNeT++ provides strong GUI support as well as statistical output and analyzing tools. The NICE protocol implementation runs on top of a P2P Overlay Framework for OMNeT++, which provides the underlying network topology for NICE. This framework establishes connections between nodes over a central Overlay Module, which simulates a real-world network topology by delaying the information exchanged by nodes according to distance matrices containing the delay between every two nodes. These distance matrices were extracted from the topology generator Brite. We implemented the NICE protocol in the OMNeT++ network simulator and extended it with the protocol-dependent and the protocol-independent QoS approaches as well as the E2E-Delay Optimization mode. The logic of the protocol and the extensions are implemented in so-called

filters, which build a filter chain. Every node in the network uses the same filter chain. The object, that is used to exchange information is the message object provided by OMNeT++. Each message has to pass the senders and the receivers filter chain, in which appropriate reactions are invoked.

Using this simulation environment, we evaluated certain scenarios and compared them together. Per evaluation we performed 780 simulation runs. An evaluation consists of 13 different distance matrices, 3 random seeds as input for the random number generator and each of these combinations were used to simulate networks containing 100 to 2000 nodes with an increase of 100 nodes per step.

Our results show that we successfully implemented the NICE protocol and that the QoS approaches really support QoS. We also see that QoS is an improvement for the NICE protocol. Additionally, we show the difference between the two QoS approaches. On top of that, the E2E-Delay Optimization can be used to guarantee RTT constraint satisfaction during the join of nodes.

Chapter 2

Introduction

The goal of this Bachelor thesis is to show that Quality of Service (QoS) can be enabled to Application Layer Multicast (ALM) - based Overlay Peer-to-Peer (P2P) networks and works efficiently. Considering today's Internet applications, the use of Quality of Service (QoS) - enabled Overlay Multicast (OM-QoS) networks is highly demanded. As an example, multimedia streaming applications can be mentioned. Imagine if the streaming server does not need to be better equipped than the clients, which are interested in the data stream. With an OM-QoS network it is possible to build such streaming networks with much less expensive infrastructure and is also usable through the whole Internet. In this Bachelor thesis, the NICE [1] protocol is used to show two QoS enabling approaches. One of the two approaches to enable QoS in NICE is protocol-dependent, whereas the other is protocol-independent and uses multiple NICE-based networks arranged in layers with interconnected gateway nodes. The protocol itself and the QoS approaches are implemented within an Overlay Framework, which runs inside the OMNeT++ network simulator allowing to simulate large sets of peers in an Internet-like environment. For the evaluation of our simulation runs, another framework was used, which allows to compare each run with different configurations to each other.

2.1 P2P Networks

Peer-to-Peer (P2P) [2] networks gained more and more interest in the last few years. P2P networks are used to share resources, such as distributing data among thousands or millions of computers, offering computation power to other peers, or providing distributed storage space. In pure P2P networks, each peer acts as server and client. This means that the network does not consist of servers and clients but of peers, which serve and request data from other peers. This is in contrast to the classical Internet structure, the client-server architecture, where data is provided by high-performance servers and consumed by clients, which are usually weaker than the servers. The advantages of P2P networks are for example the distribution of large amounts of data (large files) to numerous peers. Because every peer can provide its own files or chunks, the amount and variety of data is enormous and growing with the number of peers joined to the network. A further point is that a P2P network does not suffer from the single-point-of-failure problem known from the classical client-server topology, because the resources are distributed

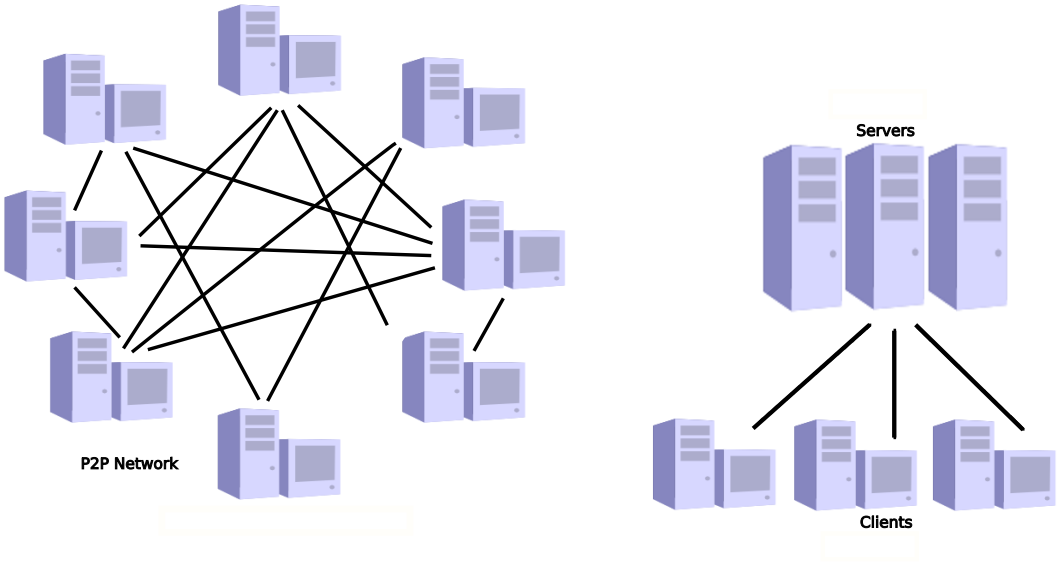


Figure 2.1: P2P vs. Client-Server Model

over the network, which increases the grade of availability of certain resources. The disadvantage of the P2P architecture is that due to the large number of peers and the small bandwidth of the peers the distribution is slowed down. The fan in and fan out of a single peer increases dramatically since each connection is established directly between the peers. In a client-server environment, the fan-in and fan-out is only concentrated at the server, which should be dimensioned to handle high load.

The general architecture of an Overlay P2P network is a layered architecture [3], which consists of the following layers:

- **Network communication layer**: Describes basic network characteristics in an ad hoc-manner. This layer does not need any centralized management device. The challenge is to manage the high dynamic nature of a P2P network.
- **Overlay peer management layer**: This layer's task is the routing, the location lookup and the resource discovery.
- **Feature management layer**: Handles security, reliability, fault-resilience, resource availability and robustness
- **Service specific layer**: Provides of parallel threads, management of compute-intensive tasks and content / file management.
- **Application layer**: Contains tools and service programs, which perform specific actions.

These layers are all packed together in a protocol stack, which is implemented in the P2P application running on the peers. Based on how the network manages the peers, two different types of P2P networks can be distinguished: Centralized and decentralized P2P networks. Centralized P2P networks use one or more central servers to manage the peer's resource requests but not the resources themselves. This is the small but important difference to client-server networks, where the resources are distributed from a central server to the clients. The servers (e.g. tracker) in a centralized network know which peer holds which part of the resource. Each peer, that is interested in a resource, asks the server to send back the IP address of the peer at which the requested resource is located. The peer then contacts the reported peer by itself and establishes a connection directly to the target peer. Decentralized P2P networks do not need any management central. Every peer is equal and takes over the roles of client and server. Each peer maintains a table about its neighborhood peers. A request for a certain resource is propagated from a neighbor to another until the requested resource is found at a specified peer. Both of the P2P network types, centralized and decentralized networks can be categorized on the basis of how the peers are placed into the coordinate or ID space. This is independent of the degree of centralization of a P2P network. Subsequently, the term "peer" is replaced by the term "node"

2.1.1 Structured P2P Networks

In a structured P2P [3] network, the ID calculation or relative placement in the coordinate space is often based on a distance metric between the nodes. Structured P2P networks have a strict hierarchical topology, which has to be maintained and controlled. For the protocol-dependent QoS approach implementation, a hierarchical structure of the network is indispensable because the nodes need to be arranged depending on their QoS requirements.

Content Addressable Network

The P2P network **C**ontent **A**ddressable **N**etwork (CAN) [4] uses a logical multi-dimensional Cartesian coordinate space on a multitorus. To each node, a partition of this space is assigned, for which the node is responsible. All the peers maintain a routing table containing IP addresses and coordinates of their neighbors.

To store a pair of key K and ressource V $\{K, V\}$, K is deterministically mapped on a point P within the coordinate space using an uniform hash function. The node, which is responsible for that section of the coordinate space, which P includes, is also responsible for this resource. To retrieve a resource assigned to a key K, the lookup protocol searches the node with the same hash function to map K onto P and retrieves the resource V from P. The node responsible for the requested resource is found by using a greedy-forward routing algorithm. This routing algorithm always makes the most progress towards the address of the resource V.

When a new node wants to join the CAN network, it needs to allocate a part of the coordinate partition of an already joined node. The joining node contacts a so-called Bootstrap Node (BS) to get the IP addresses of some random nodes and calculates a random point in the coordinate space. This point in the coordinate space is part of an area maintained by a certain node already in the CAN P2P network. The provided random nodes forward the join request message to a node, which is close to the calculated random point. This node then has to split its key space in

half and one half including all resources located in this part is assigned to the newly joined node. To update its routing table, the new node has to learn about its neighborhood.

A node leaving a CAN network has to hand over the zone it is responsible for to one of its neighbors and the neighbors have to update their neighbor table, where the leaving nodes' entry is deleted.

CAN is self-organizing, distributed, decentralized, very robust, fault-tolerant, scalable, and resilient. The robustness arises from the fact that there are several independent routing paths for retrieving a certain resource. It is also possible to improve robustness and reliability by using multiple independent coordinate spaces and replication of the resources on each coordinate space. CAN can be used in Internet-like storage applications such as OceanStore.

Chord

The Chord network protocol [5] uses a ring topology as ID identifier space for nodes and resources. The IDs are uniquely distributed on this ring using a Secure Hash Algorithm (SHA-1) hash function. The IDs are increasing clockwise on the ring. Each node has a routing table where its successor and predecessor are stored. Additionally, a node maintains a finger table with entries of nodes, which are not in the neighborhood of the node. These entries point to nodes, which are e.g $\frac{1}{4}$ or $\frac{1}{2}$ of circular distance away allowing fast forward routing. The routing information held by a node is updated using a stabilization protocol running periodically on every node.

To store a resource object, its ID is generated from the same identifier space like the node IDs, and therefore inserted into the same Chord ring. A resource is appended to the first node, whose identifier is equal or greater than the resources' ID. To retrieve a resource, the closest node to this resource is searched. With the given ID of the resource, the first node with the same or greater ID is tried to be reached. First, the requesting node compares the ID of the requested resource with the entries in its routing table, which represent its neighborhood. If the lookup is not successful, the finger table is used to jump in other regions of the Chord ring to find the node.

When a node wants to join a Chord ring, a free (randomly chosen) ID will be assigned to it and certain resources of its successor will be shifted to the new node. Leaving a Chord ring triggers shifting the resources of the leaving node to its successor.

Chord is used for wide-area name resolution services or cooperative mirroring as well as Cooperative File System (CFS), in which multiple providers of content cooperate together.

Pastry

Like Chord, Pastry [3] uses a ring topology to establish a network between the nodes. Each node randomly chooses a 128-bit long ID. These IDs are uniformly distributed over a circular coordinate space with 2^{128} unique IDs. Each node maintains three lists: the routing table, the neighbor set, and the leaf set. The leaf set contains the nodes, which are close in terms of the identifier space. Half of this set is populated with successors and the other half with predecessors. This information is used for routing. The neighbor set maintains information about the closest nodes

in terms of a proximity metric (e.g Round Trip Time (RTT)) and is used for maintaining locality properties. The routing table stores the IDs of other nodes according to their degree of ID prefix matching with the current node. The degree of perfix matching is equal to the number of matchting digits. This mechanism corresponds to Plaxton routing [6].

A resource object receives an ID from the same identifier space, from which the node IDs are assigned. A resource object is appended to the node with the longest ID prefix match. A lookup for a certain resource is routed as follows: First, the requesting node looks for an ID matching with the ID of the requested object in the leaf set. If the ID is also found, the node responsible for this resource object is found. If not, the routing table is consulted to find the node with the longest prefix match. The request is then forwarded to this node.

When a new node is joining the Pastry network it first contacts a random or a bootstrap node, which is already joined in the Pastry network. The joining node sends a join message to the bootstrap node with its own ID. The bootstrap node then sends the message to the next joining node's ID using Pastry's routing mechanism. On each hop, the routing table information of that node is added to the join message. When the closest node Z to the joining node's ID is reached, the leaf set of Z is added to the join message. Z sends the join message back to the joining node, which uses the information in the message to build its own leaf set and routing table.

The maintenance of the routing information is done by periodically exchanging keep-alive messages. Nodes, which are not responding after a certain time period are deleted form the lists.

Scribe, which runs on top of Pastry, can be used for building multicast groups and multicast trees to efficiently distribute data.

2.1.2 Unstructured P2P networks

Unstructured P2P networks [3] do not have an ordered ID assignment mechanism and are not as scalable as structured P2P networks. The nodes in an unstructured network have a very restricted view of the network - they only know some arbitrary nodes in the network. This makes the search for a requested resource very slow, because a node is not able to preview the location of the resource.

A request for a resource has to be propagated to a certain set of nodes which is time consuming and stresses the network by increasing the overhead traffic. This process is known as flooding. The flooding algorithm [7] is not efficient in terms of bandwith use. It also causes useless duplicates of data packets. The advantage of flooding is that no knowledge of the network is needed. This method of searching a resource object is faster for highly popular content but not for rarely requested resources. The fact that in unstructured P2P networks the nodes and the resource object are not correlated makes the flooding search algorithm unreliable.

Gnutella

In Gnutella [3], resource objects are placed without any knowledge of the topology of the network. The request routing is based on the earlier mentioned flooding algorithm. The modification to this algorithm Gnutella uses, is that the flooding radius is limited to a certain number of

hops. This increases the performance but decreases the reliability because a resource outside the radius is not detected by the requesting node.

BitTorrent

BitTorrent [3] is used for distribution of large files and is designed to be very social. This means that a node with a high upload data rate can also download with a high rate. In the early days, BitTorrent consisted of so-called tracker, which maintains information about a certain file or chunk of a file requested by a node. The tracker keeps track of all the nodes, which have the requested file. These nodes then are reported to the requesting nodes as servers which can directly download from the reported nodes. Nowadays, there is a trackerless approach of BitTorrent (for example μTorrent [8]). In this approach, each node has the tracker functionality, which uses a Distribute Hash Table (DHT).

2.2 Application Layer Multicast (ALM)

IP-Multicast [9] is an efficient technique to distribute data in a network. The main advantage of IP-Multicast is that if several network devices are interested in a datagram located at one single host, the host has not to replicate the data by itself. The replication is done by the routing devices in the network. This circumstance reduces the load on a data serving host. Unfortunately, IP-Multicast is not widely available to Internet end-users, altough this technique has several application fields like multimedia streaming, data distribution, etc. One of the reasons why IP-Multicast is not available in the Internet is the contracts between the Internet Service Providers (ISP) about traffic routing due to political or commercial constraints. Another reason might be the Internet infrastructure, namely the routing devices, which do not support native IP-Multicast. The solution to overcome this limitation is to introduce Application Layer Multicast (ALM). The idea of ALM is to bring the multicast functionality to the end user's system.

ALM enables the usage of multicast without changing the Internet infrastructure. This is achieved by modifying the hosts directly instead of the network devices. The main component of ALM is a service application, which provides multicast functionality. This component is present as an application in the application layer of the network stack. This service makes multicast available on unicast networks by using an Overlay Network. Each node has simply to run the ALM service application to take part in the multicast network. Compared to native IP-Multicast, the ALM approach is not as efficient as native IP-Multicast. This arises from the fact that the links between the nodes are more stressed since the packet replication is done in the end system and not only on the interconnecting devices as in native IP-Multicast. In exchange for this drawback, ALM is much easier to deploy over the Internet than native IP-Multicast, since in ALM, only an application has to run on the nodes and not the whole network has to be replaced with expensive devices, which support native IP-Multicast.

A special implementation of the ALM concept has been done in [10]. The multicast functionality is provided through a Multicast Middleware, which runs in the user space, and through a virtual network interface in the kernel space. An application, that uses multicast communicates with the virtual network interface. This interface is connected to the Multicast Middleware,

which performs all the multicast functions. The Multicast Middleware translates the received traffic from the virtual adapter into unicast traffic and sends it through a physical network device to the network. This makes the ALM feature transparent for the applications.

2.3 The NICE Protocol

In this Bachelor thesis, the "**N**ICE is a **I**nternet **C**ooperative **E**nvironment" (NICE) [1] protocol is investigated. NICE is a P2P protocol, which uses ALM as a substrate for low-bandwidth applications. NICE scales well which means that it can handle a small number of nodes as well as large numbers of nodes.

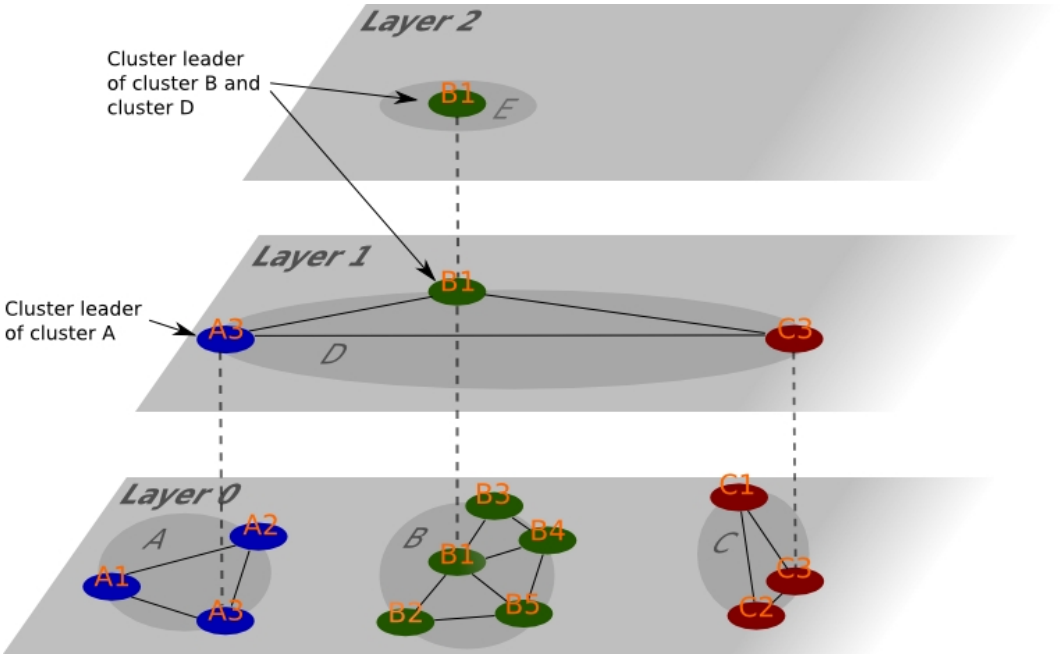


Figure 2.2: NICE Network Tree Structure

The nodes in the NICE network are arranged in clusters and layers as shown in Fig. 2.2. The smallest unit of nodes are the clusters. The nodes within the same cluster are called cluster mates. A cluster mate represents a single node connected to the NICE network. Each cluster mate knows its cluster leader and some or all of the other cluster mates by storing their node ID in a so-called neighborhood table. The cluster size is variable between the lower bound k and the upper bound $3k - 1$, where k is a predefined cluster constant greater than zero. These boundaries are chosen such that conflicting maintenance operations are avoided. The maintenance operations are discussed later in Section 2.3.3. Within each cluster, one of the cluster mates is determined to be cluster leader, which then is additionally located in a cluster of the next higher layer. The cluster leader is determined out of the nodes in the so-called graph-theoretic center of the cluster.

The calculation of the center is done as follows: For each node, its eccentricity is calculated. The eccentricity is the maximum distance from the node to any other node in the cluster. The set of nodes with the minimum eccentricity then build the center of the cluster [11]. One of these nodes is chosen randomly to be the cluster leader.

On a single layer there are one or more clusters. The layers are ordered from the bottom layer zero to the top layer n. On the top layer, there is only one cluster containing a single cluster member: the so-called root node. This topology leads to the tree structure shown in Fig. 2.2.

This structure of NICE is specified with five invariants, which have to be fulfilled at any time:

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

Our implementation of NICE has an additional virtual node called Bootstrap Node (BS). This node is responsible for new nodes, which want to join the NICE network. The BS is not part of the network. The knowledge of the BS is limited to the ID of the root node of the NICE network.

The maintenance of the NICE protocol is implemented with several operations (called refinement operations), which handle the nodes and maintain the NICE tree structure. These operations are based on the exchanged information using so-called Heart Beat Messages. Through these messages, the nodes exchange status information. The Heart Beat Message contains information about the view of a node on its cluster including its cluster leader, its cluster mates, and the Round Trip Time (RTT) to them. These messages are unreliable and periodically sent by each node.

2.3.1 Join Operation

When a new node wants to join to the NICE network, it has to contact the BS first. The BS responds with the information about the root node to the query of the new node. The new node then contacts the root node and receives from the root all cluster leaders on the next lower layer. Then it contacts every reported node to identify the closest node to itself. The idea of the selection of the closest node is to find the most appropriate cluster to join. This means that nodes, which are physically close, are grouped together in a cluster. Afterwards, the joining host sends its request to the determined closest node. This process is iteratively repeated until layer 0 is reached. There, the new node becomes a cluster mate of a cluster.

2.3.2 Leave Operation

There are two types of leaving methods of a node within a NICE network: the graceful and ungraceful leave. The graceful leave is the case when the node, which wants to leave is able to announce its departure to its cluster mates before it really disappears. This allows the other nodes to react to this situation with first stopping transmission of data to this node and then invoking the closure of the connection and deletion of the nodes' entry in their neighborhood table.

The ungraceful leave happens when a node suddenly disappears. The cluster mates and the cluster leader do not recognize the departure of the node directly. This can cause that certain traffic does not reach its destination. The cluster mates recognize such a ungraceful leave by the absence of periodic heart beats of this node for a certain time. The subsequent reaction is the same as in the case of a graceful leave: the disappeared node is erased from the cluster mates list.

2.3.3 Refinement Operation

The maintenance of the cluster hierarchy according to the invariants of a NICE network is the task of the refinement operation. Refinement in NICE consists of several operations (split a cluster, merge two clusters, determination of a new cluster leader), which are invoked by a cluster leader when it detects an invalid or unoptimized state of its cluster. As an example, when a cluster leader detects that the number of its cluster mates exceeds the upper limit of $3k - 1$ nodes, it has to invoke a cluster split. If this happens, the cluster is split into two clusters with at least $3k/2$ nodes in each new cluster. The cluster is partitioned into two subsets in such a way that the graph-theoretic radius of the new set is minimized. Each subset determines a new cluster leader. If the size boundaries of the clusters are still violated, the same procedure is invoked on each new partition. If the cluster size is below the lower boundary k, the cluster merge operation takes place. The cluster leader sends a cluster merge request to the closest cluster leader on the same layer. Then the merge information is spread among these clusters and the invoking cluster leader has to give up its cluster leadership and leaves the upper layer. Leave and join operations also cause that the cluster leaders could change. It is periodically checked if the current cluster leader is still valid. When a cluster leader receives the information that it is not the best suitable cluster leader anymore, it invokes a cluster leader transfer. A new cluster leader is determined by calculating the graph-theoretic center within the cluster mates. The new cluster leader replaces the old one. Therefore, the new cluster leader has to join the upper layer and the old one has to leave it.

2.4 OM-QoS

Quality of service (QoS) is a generic term for managing mechanisms to reserve resources for a host in a network. Such resources can be for example bandwith, delay, computation power, etc. The idea is that a node that is requesting a resource, can specify with which quality the resource is delivered. The node requests the resource with a certain QoS. The QoS-aware networks' task is to fulfill these requirements. To achieve this, the path from the resource hosting node to the

node, which is requesting the resource has to be built to support the requirements. If the QoS requirements are too high, the resource providing node can reject the request when it can not guarantee the abidance of the requirements and the node has to decrease its requirements or even drop the request.

In this thesis, Overlay Multicast Quality of Service (OM-QoS) [12] is basically used, which is the application of QoS in a Overlay Multicast P2P network. The idea behind OM-QoS is to assign a so-called QoS class to each node. Such a QoS class is represented as an number, which is the weighted sum of several network parameters like bandwith, delay, etc.

According to the QoS class of the nodes, the network structure is built. The general structure of an OM-QoS enabled network is a tree. In case of a NICE network, the root node, which acts as sender has the highest QoS class. The path to each receiver nodes has to be monotonically decreasing because otherwise the root node can never fulfill the QoS requirements of any receiver node. An example is shown in Fig. 2.3 where bandwidth is used as QoS constraint.

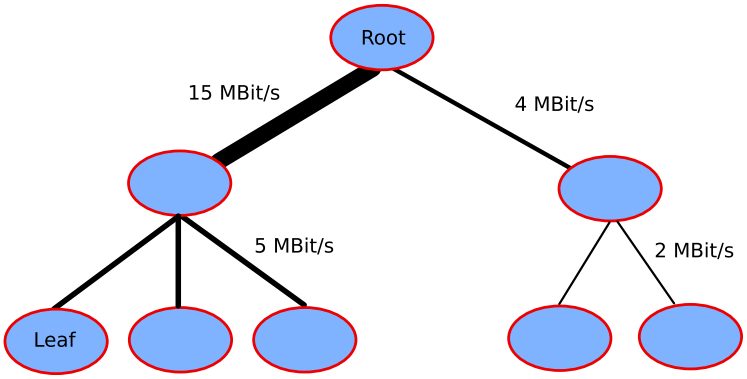


Figure 2.3: QoS Example Tree with Bandwith Requirements

To maintain this structure, the QoS classes have to follow certain rules:

- A total order relation for all QoS classes exists.
- QoS class parameters are independent of link length and number of hops in the network.
- The amount of QoS classes is finite.

The first rule is essential to guarantee the monotonically decreasing path in terms of QoS from the sender (root) to the receiver (leaf). The total order relation allows to compare two nodes according to their QoS class which is eminent for their positioning in the tree structure.

The second rule has to be respected because of the repositioning of nodes in the tree without change of their QoS class. Repositioning a node happens when the decreasing order within a path from the root to the leaf is violated to that node. The node has to be replaced in the tree, which may cause a change in its link length and may also affect the number of hops to the root but the nodes' QoS class does not change.

The third rule guarantees that the structure is manageable. We can not deal with an infinite number of QoS classes in a tree.

2.4.1 Hard- / Soft-QoS

Hard-QoS

When a node joins a hard-QoS supporting network, the node decides, which QoS requirements it needs to perform its task. The node then requests this QoS from the network. The task of the network between the requesting node and the resource provider is to choose a path between them, which guarantees the requested QoS. The hard-QoS functionality is static, because once the resource is reserved, it is guaranteed that the requesting node receives the requested QoS.

Applications for hard-QoS enabled networks are for example remote surgery or any network-based security systems where certain communication channels have to stay established even under heavy load in the network and where a certain quality of the communication channel is essential.

Soft-QoS

Soft-QoS is also called best-effort-QoS or measurement-based-QoS. This approach does not need a reservation system for resources and is therefore more flexible than the hard-QoS mechanism but less reliable. The general situation is the same like in a network, which supports hard-QoS: A node requests certain Quality of Service requirements. The fulfillment of the QoS is monitored by periodic measurements of the necessary QoS-parameters by the nodes. The state of the QoS satisfaction can change. This means that suddenly, the service provider is no longer able to serve the resource as the node has requested. Therefore, the resource provider has to inform the node to change the connection to another node, which can provide the resource with the necessary QoS requirements. This leads to a path change, which is the reason for the dynamic manner of soft-QoS.

2.4.2 Protocol-Dependent QoS Approach

To enable OM-QoS in NICE using the protocol-dependent QoS approach, the behavior of NICE has to be modified. The modification affects the cluster leader determination method.

Without OM-QoS, the cluster leader of a cluster is determined as one of the cluster mates in the graph-theoretic center within the cluster. To enable OM-QoS for NICE, we change this determination method. In this case, the cluster mate with the highest QoS class becomes the cluster leader. This selection is possible, because each cluster mate knows its QoS class. This modification fulfills the requirement of the OM-QoS principles that the path from the root to a leaf has to be monotonically decreasing.

End-to-End Delay Optimization

The End-to-End Delay Optimization (E2E-Delay Optimization) is an additional mode, through which RTT constraints for the path from the node to the root can be guaranteed. This mode can be switched on independently from the other NICE modes. Like the protocol-dependent QoS-approaches, the E2E-Delay optimization affects the NICE protocol in its functionality. The modification is applied to the initial join process of a node. Each node receives a random value

in a predefined range as its E2E-Delay constraint. The joining node connects to the cluster leader in such a way that the joining node fulfills its E2E-Delay constraint as close as possible.

2.4.3 Application to NICE

To provide QoS mechanisms to the NICE protocol, first all nodes need to have a QoS class assigned when they join the network. This class is represented as a number between 0 and 255. The value can be interpreted for example as a representation of bandwith requirements: The higher the value, the higher the bandwith requirements are. Since this representation fulfills the constraints mentioned in section 2.4, it can be used to build a QoS-aware network topology.

The changes to be applied to the original NICE protocol are small. As mentioned before, only the cluster leader selection is adapted. Unlike the original NICE, which selects the cluster leader according to a underlying topology-aware graph-theoretic calculation, the QoS-aware NICE selects the cluster mate with the highest QoS class instead of one in the cluster center as new cluster leader. Due to this, the topology fulfills the requirement of decreasing QoS paths from the root node to the leaf nodes as shown in Fig. 2.4.

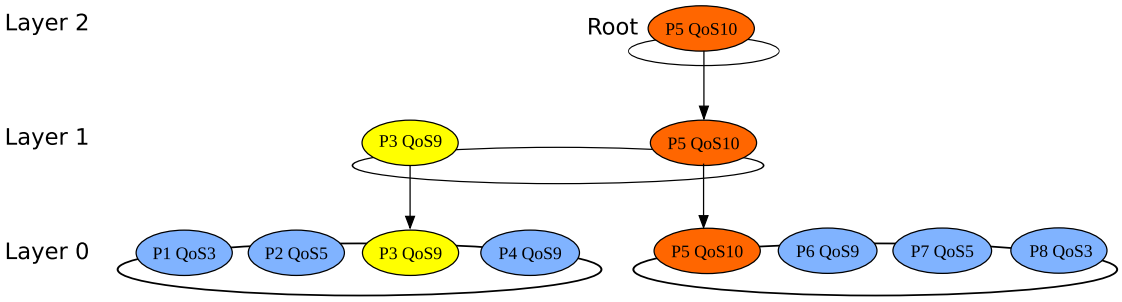


Figure 2.4: QoS-aware NICE Topology

The node's names are Pi where i is any number and QoSx where x is the QoS class of the node. The root node P5 is the node with the highest QoS class and is present on the top layer (layer 2) as well as in all lower layers (layer 1, layer 0). All other nodes have a smaller QoS class than node P5, which is cluster leader and root. Node P3 is the cluster leader for the left cluster since it has the highest QoS class in it and is therefore present in a cluster in layer 1. If two nodes have the same highest QoS class, one of them is chosen randomly as cluster leader.

2.4.4 Protocol-Independent QoS Approach

In contrast to the protocol-dependent QoS approach, the protocol-independent QoS approach does not require any modification of the NICE protocol itself. Here, we have multiple NICE networks arranged in disjoint layers. This is the reason why we also call it "Layered OM-QoS Framework". The only common point among the layers is the BS, which is not part of any network. These layers are not to be mistaken for the layers, which NICE uses internally. The BS is the only node, which has to be able to handle the QoS classes. Each of the layers handle

nodes of a designated QoS class. This means, that all nodes within one layer have the same QoS class.

In the protocol-independent QoS approach, the BS knows not only one single root like in the native NICE and the protocol-dependent QoS approach, but knows each root of the different NICE networks in the layers. This allows to assign new joining nodes directly to the correct NICE network according to their QoS class. The root nodes of the NICE networks are called gateway nodes. The interconnection of the gateway nodes build a chain through which the multicast traffic to every dedicated NICE network is sent.

2.5 OMNeT++

OMNeT++ [13] is public-source, component-based, modular and open-architecture simulation environment with a strong Graphical User Interface (GUI) support and an embeddable simulation kernel. OMNeT++ is written in C++ and runs on several UNIX platforms as well as on Windows platforms. The purpose of OMNeT++ is to simulate IT networks containing different devices such as routers, computers, servers, etc. OMNeT++ is able to handle small or large networks, which means that it scales well. The application area of OMNeT++ is in scientific laboratories as well as in industrial settings and experiences rising popularity. Each component of a realworld-network is represented in OMNeT++ as a model. Due to the fact that OMNeT++ is public-source, it's highly modifiable. OMNeT++ consists of several components:

- Simulation kernel library.
- Compiler for the NED topology description language (nedc).
- Graphical network editor for NED files (GNED).
- GUI for simulation execution, links into simulation executable (Tkenv).
- Command-line user interface for simulation execution (Cmdenv).
- Graphical output vector plotting tool (Plove).
- Graphical output scalars visualization tool (Scalars).
- Model documentation tool (oppneddoc).
- Utilities (random number seed generation tool, makefile creation tool, etc.).
- Documentation, sample simulations, etc.

A simulation network in OMNeT++ is described in a so-called NED file. The NED file has its own syntax, which is easy to understand. NED files can be loaded dynamically into simulation programs, or translated into C++ by the NED compiler and linked into the simulation executable. These files provide information about the topology of the network as well as of the modules (models of the devices) and their connections. The GUI shows the topology of

the network through parsing and visualizing the NED files. The communication between the network modules is realized by exchanging messages. A message can represent, for example a frame, a packet or a job in a computer network. Messages are defined in the MSG files. Like methods in a C++ class, in a MSG file, more than one message can be defined. A definition of a message simply contains its parameters, which can be basic C++ data types and arrays of them. While compiling a simulation program, OMNeT++ translates the message definitions into C++ code which allows to use the automatically generated accessor methods in other parts of the simulation code.

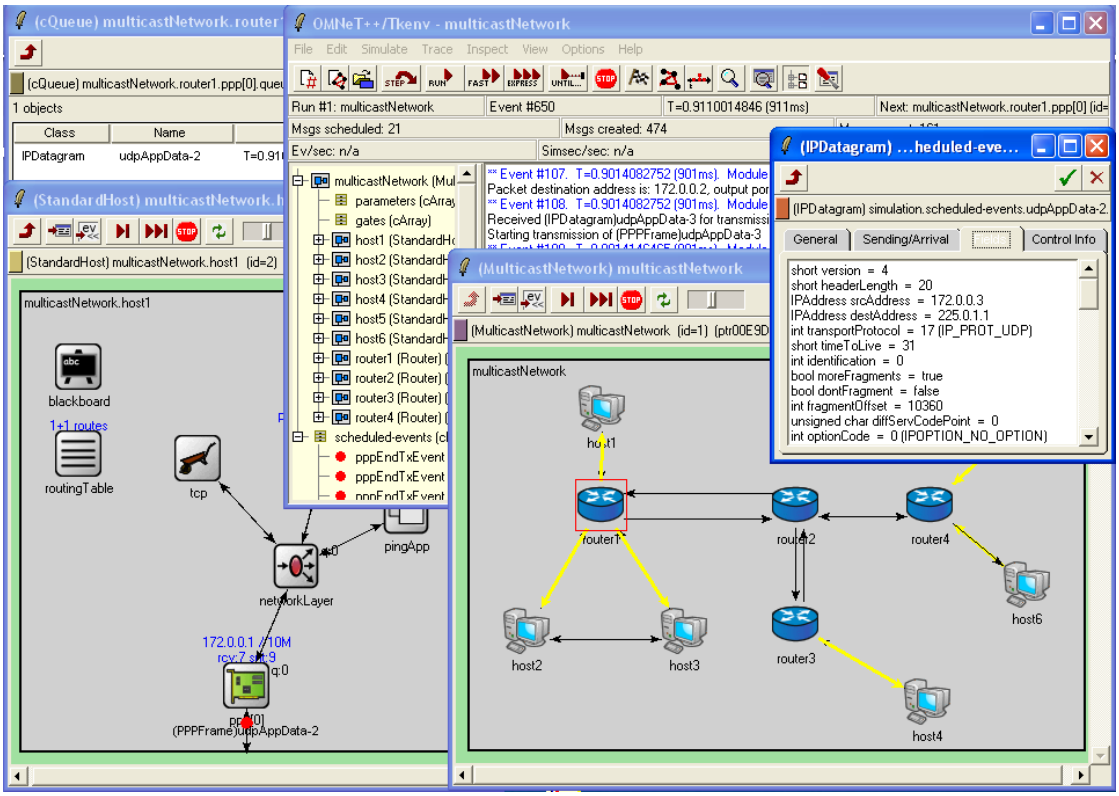


Figure 2.5: OMNeT++ GUI

OMNeT++ provides two statistical tools to analyze a simulation run and its underlaying network: Scalars [14] and Plove [15]. Plove is used to plot one or more OMNeT++ output vectors (VEC-files) in one graph. There are also built in filters, which modify or extend the data before plotting. Such filters are averaging, truncation, smoothing, histogram, etc. The drawn graphs can be exported to various formats like EPS, GIF, etc. or copied to the clipboard from where the data can be modified in spreadsheet applications. The vector files are deleted for each new run of the simulation.

The second tool used for statistics is Scalars, which visualizes the output scalar files of an simulation run in a x-y-axis plot. Like in Plove, the graph can be exported to EPS as well as to GIF and many other formats. Such a scalar file (SCA) produced by OMNeT++ contains

information about several different simulation runs. This allows to create a statistical statement about how a network behaves under different configurations (e.g. variate number of nodes). Unlike the output vector file, the scalar file grows with growing number of runs because the data of each run is appended to the file. The statistical output files are not created automatically. To create the output files, certain commands have to be implemented in a modules' source code.

Because the OMNeT++ simulation framework is highly flexible and scales well, it is very suitable for this work. In contrary to other network simulators, OMNeT++ is free for academic use, what makes OMNeT++ fit for these studies.

Chapter 3

Implementation of NICE

The implementation of the NICE protocol is realized in the OMNet++ network simulator, which is described in section 2.5. In this Chapter, we explain the most important parts of our NICE implementation. We show how the principal components, the filters, the nodes, and the messages interact. Additionally, we explain the NICE protocol with a small topology example.

3.1 P2P-Framework in OMNet++

Our implementation is based on a P2P Overlay Framework as used in [16, 17]. In this framework, all nodes are connected to a module called Overlay Module. The Overlay Module is able to establish the connections between the nodes automatically based on the source and destination address of a message. Without such a module, we would have to specify each connection manually, which is very hard to do for large sets of nodes.

The Overlay Module simulates a real-world link between two nodes by delaying messages between them. These delays are read from a distance matrix, which has been extracted from topologies built by BRITE [18]. The configuration setup for BRITE is explained later in Section 4.1. The use of BRITE allows us to simulate the behavior of our NICE implementation within an Internet-like environment.

3.2 Example Topology

As an example topology, a NICE network with 9 nodes (including the BS with the ID 0) is investigated and the cluster constant k is set to 3 ($k = 3$). The simulation environment first invokes the initialization of the nodes. The initialization process is realized by sending Node Init Messages to the nodes. After receiving the Node Init Message the node 0 knows that it is the BS node and is ready to manage root node information. The first "real" node, that is joining the NICE network, sends a Join Query Message to the BS. Since this node is the first node the BS stores this node as root. The response to the Join Query Message is a Join Response Message, which contains information which node is root. In this case, the response contains the first node. The response is received by the first node and it reads the information about the root. The node changes it state to be root since it gets its own ID back as root from the Node Init Message. This

node is now present in the layer 0 and 1. Since it is root, it is cluster leader for the only cluster in layer 0 too. All other nodes join the NICE network one after the other. When all nodes have joined, a single cluster exists in the NICE network.

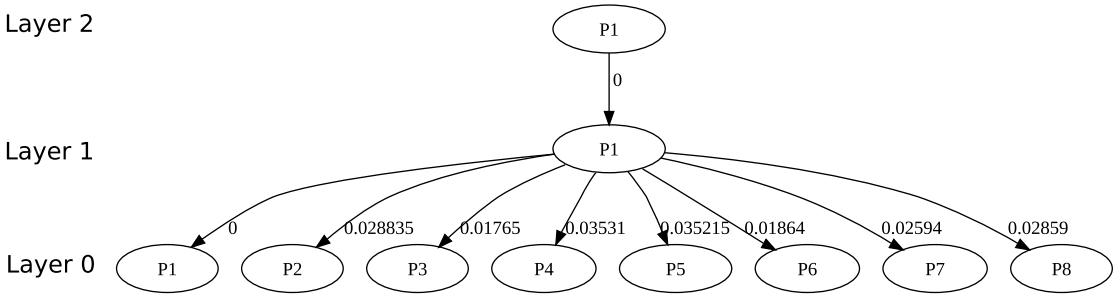


Figure 3.1: NICE Example Topology after 4.5 Seconds of Simulation Time

Figure 3.1 shows the topology of the described NICE example after 4.5 seconds of simulation time with an arrival rate of 2 nodes per second. The BS is not part of the network itself. The cluster in layer 0 will be splitted when the number of cluster mates exceeds maximal cluster size.

Figure 3.2 shows the joining process. After receiving the Node Init Message from the Overlay Module P1 sends a Join Query Message to the BS. The BS returns a Join Response Message including the ID of the root, which is in this case the node ID of the joining node P1, since it is the first one to join. For the rest of the figure, assume that nodes P2 and P4 have already joined. Then node P3 joins the network. It has first to contact the BS to receive the information about root node P1. This is done via Join Query / Join Response Messages like before. The RTT QoS Info Request Messages and the RTT QoS Info Response Messages determine the RTT to the root. The RTT QoS Info Request Message collects the RTT to several given nodes. In this case, only one node (P1) is measured. This mechanism is used to find the nearest cluster leader to join. In our scenario, only one cluster leader is available. After finding the appropriate cluster leader, the joining node sends a Join Cluster Request Message to the cluster leader found. When the cluster leader P1 receives this message, it takes P3 in its cluster and sends a Join Cluster Response Message to P3. This signals P3 that it has successfully joined to the NICE network. To inform the other cluster mates, which are already joined to the same cluster, P3 sends heart beats to the cluster mates to make them aware of the newly joined node. Sending Heart Beat Messages is periodically invoked by all joined nodes.

3.2.1 NICE Node

Each node, that is connected or is about to join the NICE network, is represented as a NICE node object (nice_node.cc). The nodes we use in our simulations are derived form the simple module object in OMNeT++ (cSimpleModule). A NICE node acts as a container for several other classes (for example filter), which contain the logic of the protocol. These filters are collected in a filter chain as shown in the Fig. 3.3. Each NICE node has the same filter chain. The concept of the

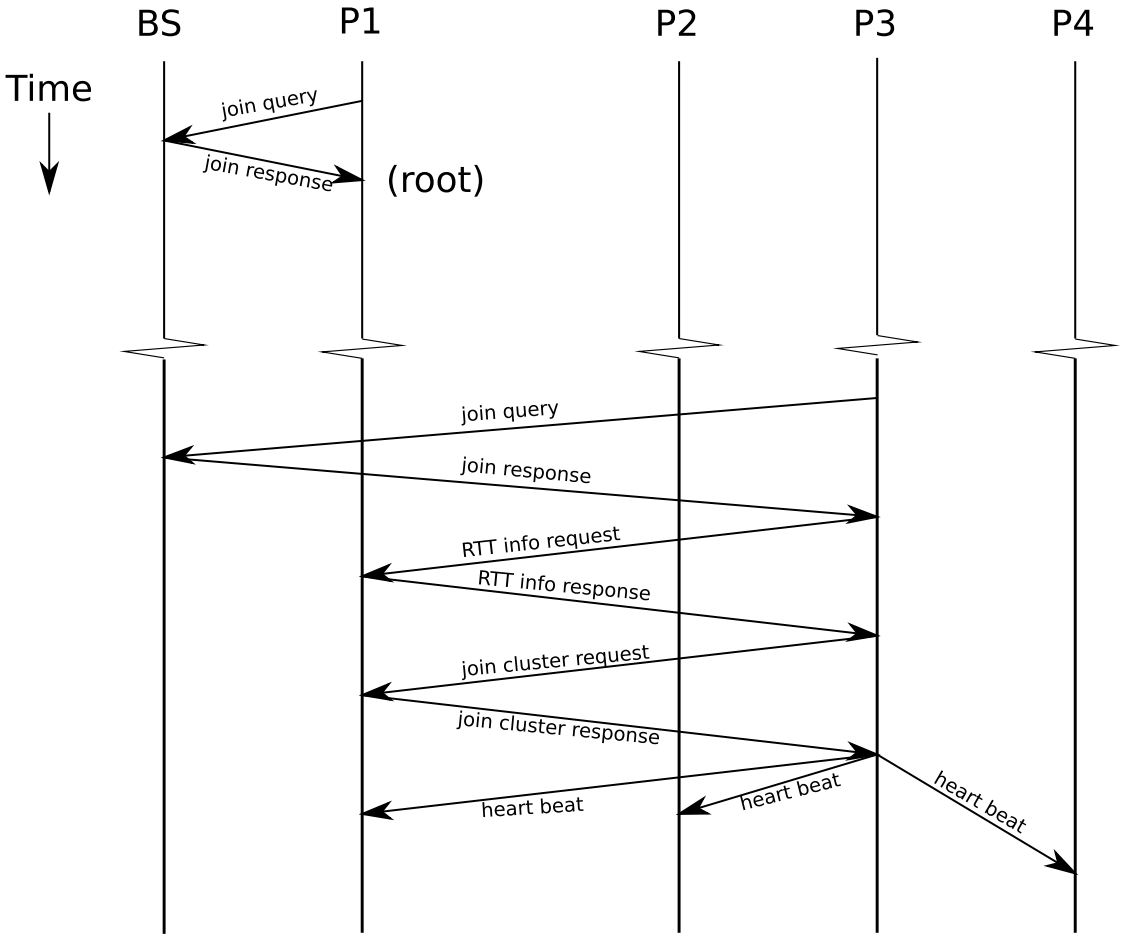


Figure 3.2: NICE Join Process

filter chain is a part of the P2P Overlay Framework. Every message received or sent by any node passes this chain. If an incoming message has passed through the chain, it is deleted.

The QoS Manager Filter is the filter, that sent messages enter first. Before the messages leave the node, they pass the Outgoing Address Filter. After the message passed the Outgoing Address Filter it is forwarded to the Overlay Module, which was mentioned earlier in Section 3.1.

Besides the filter chain, a NICE node stores a unique range of message IDs, which are applied to Overlay Messages generated by the filters. Such unique message IDs are required to map the appropriate response message to a request.

3.2.2 Filters

A filter invokes different actions depending on the message type and its payload, for example, store the information from the payload or change states. A filter detects the direction of a mes-

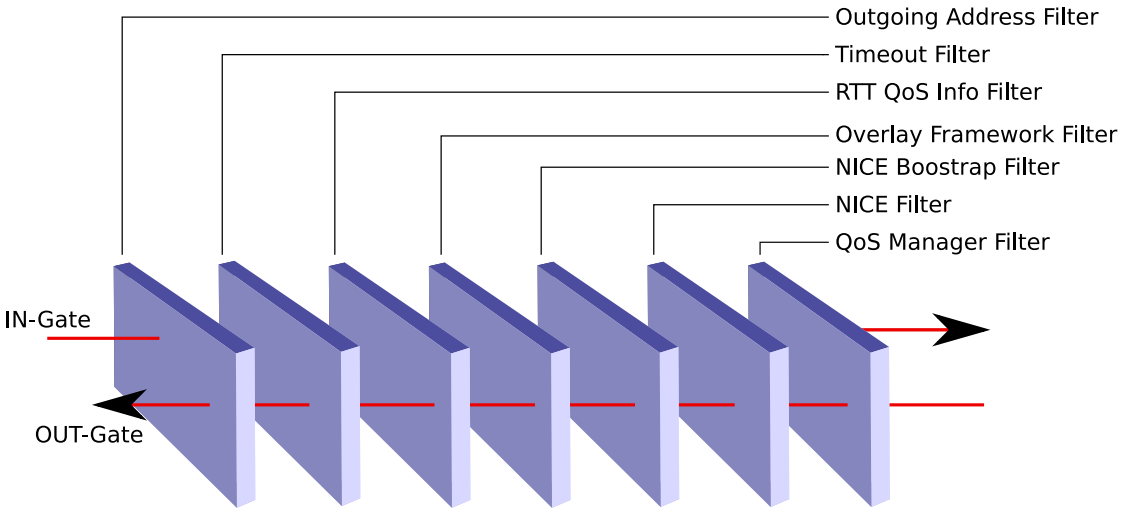


Figure 3.3: NICE Node Model with the Filter Chain

sage. The same message can not be sent back to the previous filter directly. Nevertheless, if this would be required, the use of a so-called Self-Message is advised. Self-Messages are explained in Section 3.3.1. The logic of the NICE protocol is implemented in the NICE Filter (nice_filter.cc). A filter class typically consists of three methods:

- processIncoming
- processOutgoing
- finish

```
void Filter::processIncoming(cMessage *msg, FilterContext *context);
```

The processIncoming method invokes actions when a message passes the filter from the Overlay Module to the NICE node. Normally, the processIncoming method passes any incoming message through to the next filter. If a filter in the filter chain is not interested in the current message, it has to pass the message to a following filter. Without the passthrough, the message would be killed and never arrive at the filter, which would be interested in the message.

```
void Filter::processOutgoing(cMessage *msg, FilterContext *context, double delay);
```

The processOutgoing method invokes actions when a message passes the filter from the node to the Overlay Module.

```
void Filter::finish(cSimpleModule *module);
```

When the OMNet++ simulator is closed, the user is asked if the finish methods should be invoked or not. In our implementation this method is empty. This method can be filled with actions to record statistics.

In our implementation, we use several filters, which all inherit and extend the described general filter. The use of filters allows an encapsulation of several tasks into an easily understandable

structure and allows to extend the nodes functionality without much effort by adding a new filter to the nodes' filter chain. It is important to know that message handling depends on the order of the filters in the filter chain. This is not because the message would be consumed by one single filter and deleted afterwards and so the message would not be available to any following filter. This is because the information in a message can cause changes of the state of the node, which depends on the processing order in a node. In general, filters should pass every message to the next filter since other filters down the filter chain could be interested in the message. In the next few lines, we will shortly explain the filters we use. The first three filters are part of the P2P Overlay Framework mentioned earlier.

Outgoing Address Filter

The task of this filter is only to ensure that each message, which leaves the Outgoing Address Filter has the source address set to the ID of the node, in which the filter is placed. Therefore, the Outgoing Address Filter is the last filter a message passes before it is leaving a node.

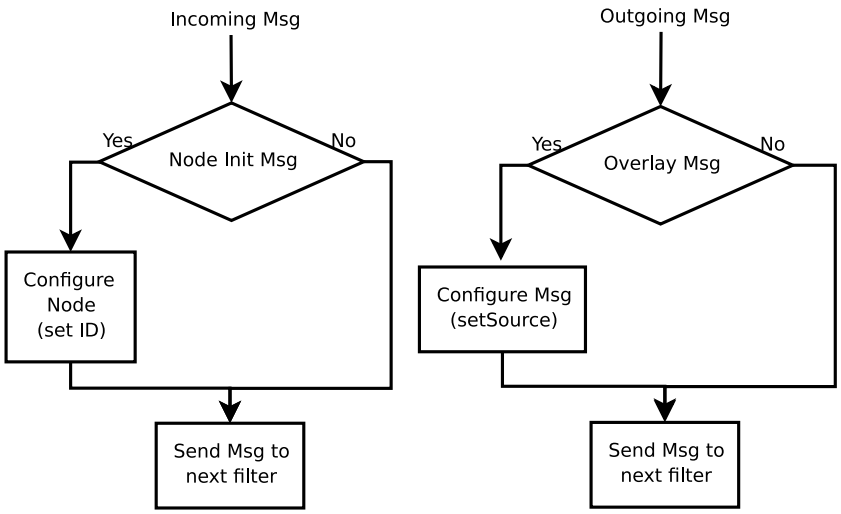


Figure 3.4: Outgoing Address Filter Flow Chart

Timeout Filter

Timeout Filter blocks any incoming messages sent over the network (not direct messages) if the node has left the NICE network. The state that the filter blocks messages is set if the filter receives a Node Dead Message indicating that the node has left the NICE network. If the filter is blocking messages, it responds with a Timeout Message sent to the source of the blocked message. Every outgoing message is allowed to pass the filter.

RTT QoS Info Filter

The RTT QoS Info Filter handles RTT QoS Info Request Messages and provides an extended ping mechanism. A RTT QoS Info Request contains a list of nodes to which the RTT should be measured. The RTT QoS Info filter generates for each of these nodes a QoS Ping Message and waits until every pinged node has answered with the corresponding QoS Pong Message. The filter then sends the same list of nodes extended by the measured RTTs and the measured RTT to root as a RTT QoS Info Response Message back to the source of the RTT QoS Info Request Message. In NICE, this filter is used during the join process to find which of the cluster leaders is the closest to the joining node

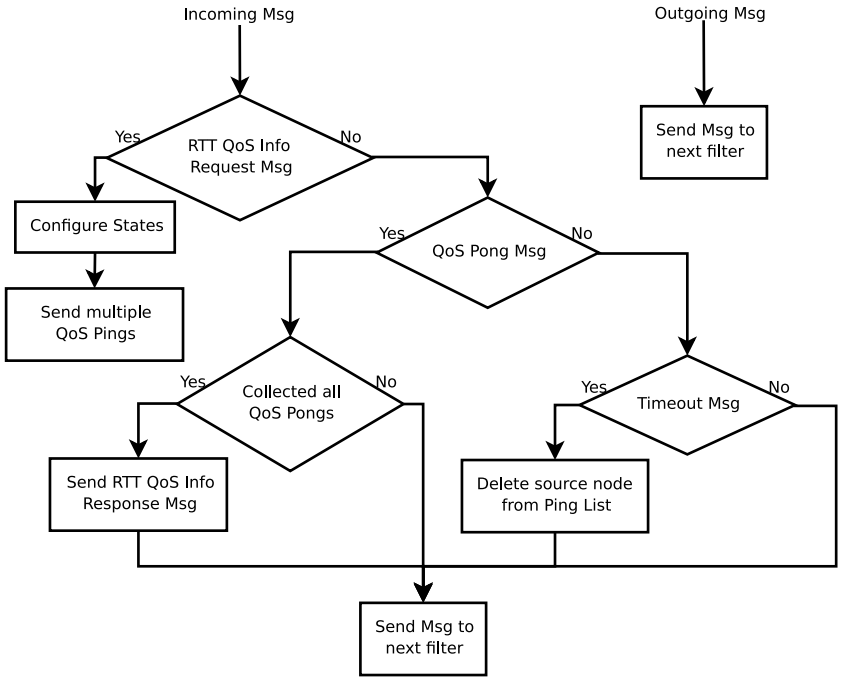


Figure 3.5: RTT QoS Info Filter Flow Chart

Overlay Framework Filter

This filter enables the protocol-independent QoS approach mentioned in Section 2.4.4. It handles several NICE network instances and is part of the Layered OM-QoS Framework described in [16].

NICE Bootstrap Filter

This filter is important for the node, that acts as BS. The BS is responsible for managing the root node. This filter contains the logic for handling root node changes. First of all, the NICE Bootstrap Filter is interested in root transfers since it always has to know which node is root. The

message used to inform the bootstrap filter is the Root Transfer Messsage. The filter maintains a variable, which contains the current root node and its QoS class. The QoS class parameter is only important, if we want to use the QoS functionality in NICE. If the filter receives a Root Transfer Message with a new root, it simply replaces the current root with the newly reported one in the Root Transfer Message. To make use of the protocol-independent QoS approach the Bootstrap Filter is inherited from Overlay Framework Bootstrap Filter (which is part of the Layered OM-QoS Framework), which is able to manage several NICE networks, and therefore several root nodes.

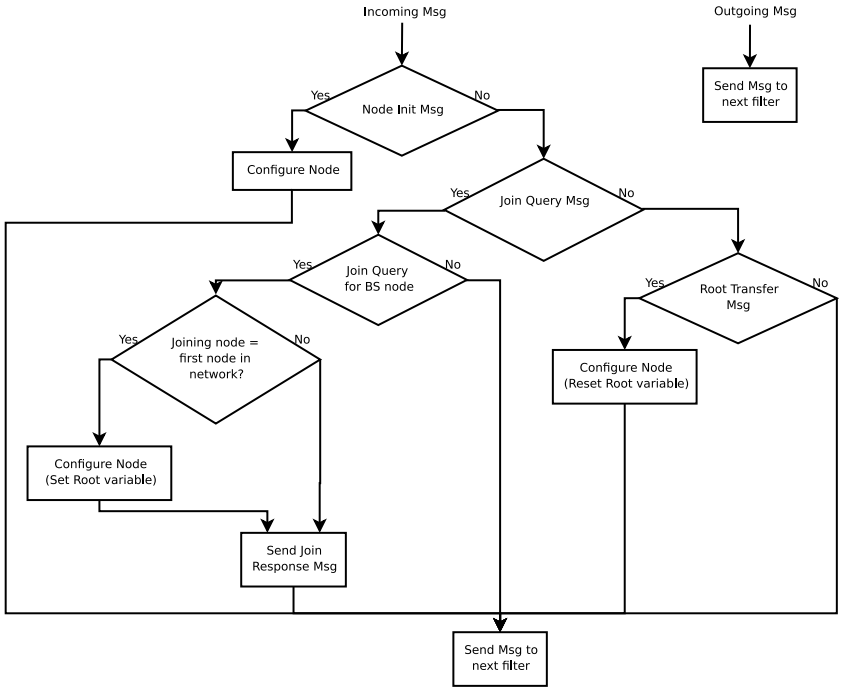


Figure 3.6: NICE Bootstrap Filter Flow Chart

NICE Filter

This filter is responsible to perform all actions, which are specified in the NICE protocol (e.g join, leave, refinement, etc). This filter makes the node to be a NICE node. It reacts to all messages used in the NICE network. The status information about the cluster mates and the cluster leader of a node is stored within the NICE filter. To store this information I use a C++ STL map of maps. The outer map is defined as follows:

```
map<int, map<int,NiceInfoEntry *> >*
```

This map uses an integer as key, which represents the layer. The value of such a key is the inner map which represents the nodes on this layer. The inner map looks like this:

```
map<int,NiceInfoEntry *>*
```

The key of the inner map is a node ID. For each key, the value is a pointer to a data container object called Nice Info Entry (nice_info_entry.cc). This container object represents the information about a node and its view of the cluster. The Nice Info Entry object provides several methods to manage and update this information. Refer to the source code file for more details of these methods.

The illustration of the information representation is shown in Fig. 3.7. This information was taken from the root node with ID 3, which has the information about three layers: L0, L1, L2 with running native NICE. It is mandatory that node 3 is present in every layer. Moreover, it is indispensable that node 3 is located in the top layer (L2) since it is the root node. In layer L0 we can see that node 13 has not yet sent all its cluster info to node 3, since the entry of node 13 in layer L0 of node 3 contains no info about how node 13 sees its cluster. This information would be filled with the next heart beat message sent from node 13 to node 3. If a node has the RTT value -1, this means that the node has not yet measured the RTT and send that RTT value in a heart beat to the node 3. The RTT value 0 of node 3 is due the fact that the node 3 has a RTT to itself equal to zero.

A rough overview of the implementation over the NICE Filter (nice_filter.cc) is shown in Fig. 3.8. It illustrates the most important message interactions of a cluster mate. For a closer look at the implementation details, consider the source code or see Section 3.3.2 about messages.

3.3 Information Exchange with Messages

The medium, that is used to exchange information, is a message object. A typical usage of a message object is for example to model a Transmission Control Protocol (TCP) packet. Such messages are defined in the MSG files, which contain one or more message definitions. The messages used in NICE are defined in the file nice-messages.msg. A definition of a message has to specify its name and the parameters, called fields of the message. The parameters contain the information, which is transported, and which can be any basic data type known in C++. A message can also be inherited from another message.

3.3.1 Message Types

In OMNeT++, there is a message type called cMessage [19]. It models every event and information exchange in the simulator. In the P2P Overlay Framework implementation, we distinguish several message types, which all are inherited from the cMessage object. OMNeT++ provides the ability to add a delay to the message. This allows to delay the sending of a message, which is a very important functionality. This allows to schedule the sending of a message at a specific time in the simulation. It is important to keep in mind that all messages regardless of its type, pass the filter chain.

Self-Message

Self-Messages are special messages, that do not leave the node that generated it. This means that such a message is not sent out to any other module. The purpose of Self-Messages is to

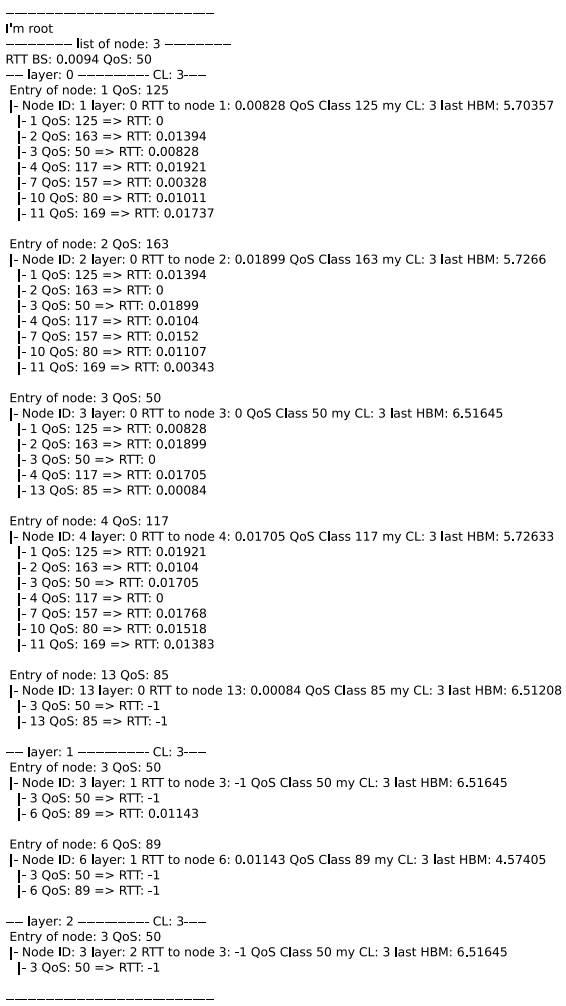


Figure 3.7: NICE Node Info

have an object, which can be used for timers and reminders. In our implementation, all reminder messages are of this type. They are scheduled and sent through the filter chain like any other message when the timer has expired.

Direct Message

Direct Messages are important if we want to send information to a node without any delay. They are not routed via the Overlay Module which means that such messages do not experience any delay. Direct Messages are not often used, since they ignore the network topology. This circumstance makes them to be used mostly for statistical analysis, since they can instantly provide information, which is statistically relevant.

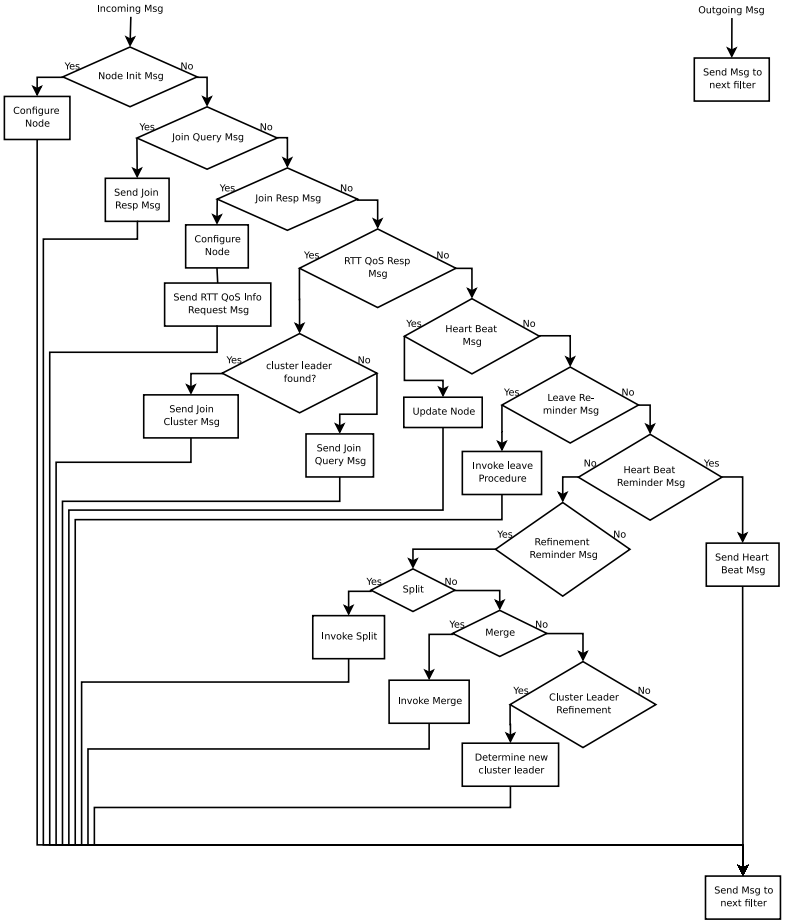


Figure 3.8: NICE Filter Flow Chart

Overlay Message

The Overlay Message is the base type (parent in the inheritance chain) for almost all of the messages except the Self-Messages and some specific messages. Such an Overlay Message fits best in the picture of an information transportation object. It is sent from the source node to the Overlay Module, which delays the message by the RTT time between source and destination. Then the message is delivered to the destination node.

```
message OverlayMessage
{
    fields:
        unsigned int source;            // the source node Id as address
        unsigned int destination;       // the destination node Id as address
}
```

We often used an important extension to the Overlay Message called TraceableMessage.

```
message TraceableMessage extends OverlayMessage {
   fields:
        unsigned int id;                // the message ID
}
```

This message type contains an additional field, called ID, which allows to set an unique ID to the message. Such a mechanism is important during handshakes, when a response has to be mapped to an earlier used request message. This allows to uniquely identify a response message and match it to the corresponding request message.

3.3.2 NICE Protocol Messages

All of the following messages are either defined in the nice-messages.msg or overlay-messages.msg file.

Node Init Message

The Node Init Message is sent to every node joining the NICE network and is sent directly to the new node without any delay because it is not derived from the Overlay Message. According to the information in this message, the node is configured. Each node has an unique ID transmitted with the Node Init Message. This ID is read and stored by each filter in the filter chain. Another ID delivered with the Node Init Message is the ID of the BS. It is important that each node knows at least the BS, because the BS acts as the entry point to the NICE network and is needed by a node to join the NICE network. This ID also does not change during the simulation. Each message (inherited from Traceable Message) sent by a node has an unique message ID as described above. The range of the message IDs is also stored in the Node Init Message. Every node has a distinct message ID range. The parameters described so far are common for all P2P networks simulated in OMNeT++ using our P2P Overlay Framework.

Every NICE relevant parameter is passed to the node using the initParameters field, holding comma separated values. This field contains the NICE network parameters (e.g. the cluster size constant k, the refinement period, the Heart Beat Message period, etc.). This string specifies in which mode the NICE network operates.

To configure the QoS parameters, there is another field called qosChangeParameters. There, the QoS class range is specified and the node chooses randomly one QoS class out of this range.

All these parameters are set in the omnetpp.ini file, from which the Init Messages are configured.

Join Query Message

The Join Query Message is a message that is mainly used by a new node that wants to join a NICE network. A node, which wants to join, sends this message to the BS which responses with a Join Response Message. The purpose of the Join Query Message is to find the possible cluster leader nodes to join. For an illustration of the joining process see 3.2.

According to the first Join Query Message received by the BS, the BS sets the root ID to the source of this first Join Query Message. If the Layered OM-QoS Approach is enabled, the BS stores for every QoS class the first node which joins. Each of these stored nodes become root nodes respectively gateway nodes - one for each dedicated NICE network. Every further joining node joins the root node which has the same QoS class like itself.

Join Response Message

When a cluster leader receives a Join Query Message, it generates a response to this query - a Join Response Message. This message contains an array of node IDs, which all are possible cluster leaders for the joining node and a layer number. These nodes are located on the next lower layer to the layer number in the Join Query Message. One of the reported nodes will receive another Join Query Message form the joining node if the new node has not yet reached the layer it wants to join.

RTT QoS Info Request Message

To summarize RTT measurements for multiple nodes, we can use a RTT QoS Info Request Message. The message consists of one single array of node IDs. These numbers are the IDs of the nodes, to which we want to measure the RTTs.

The RTT Info Request Message is processed by the RTT QoS Info Filter earlier described in Section 3.2.2. This filter does the effective RTT measurement by sending QoS Pings and collecting QoS Pongs for the reported nodes in the array of the message as described in Section 3.3.2. The response message - the RTT QoS Info Response Message contains the RTT to the requested nodes and their RTT to the root node. The RTT to root information can be used to optimize the RTT to the root node when joining the NICE network.

RTT QoS Info Response Message

Like mentioned in the description of the RTT QoS Info Request Message, the RTT QoS Info Response Message is returned as an answer to the source node of the RTT QoS Info Request Message. The RTT QoS Info Response Message is used during the join process. If a Join Query Message has to be forwarded to a cluster leader on the next lower layer, the joining node uses the join response to detect the closest node in terms of RTT on the lower layer to forward the next join query on the next lower layer.

This process of exchanging Join Query- , Join Response- , RTT QoS Info Request- , and RTT QoS Info Response Messages is repeated during the join process until the node knows which node it has to choose as appropriate cluster leader in the layer it wants to join.

Join Cluster Request Message

The Join Cluster Request Message is sent to a cluster leader to which the source node wants to join in a particular layer. The node, that receives this request, checks if it is able to take the requesting node into its cluster. If true, the cluster leader sends a Join Cluster Response Message to the source of the request and inserts the node into its cluster mates list on the layer the new node joined.

Such a Join Cluster Request Message is not only used during the initial join procedure but also if a node has to change its cluster leader, and therefore has to join to a new cluster. Then, no Join Query Message via the BS is necessary, since the node already knows which cluster leader it wants to join.

Join Cluster Response Message

The Join Cluster Response Message represents the acknowledgment of the corresponding Join Cluster Request Message. This message is sent back to the source of the Join Cluster Request Message and indicates that the request was accepted and the node has successfully joined the cluster.

Cluster Leader Validation Message

The Cluster Leader Validation Message is used to correct network anomalies, which can occur when a node has outdated cluster information. A cluster leader validation is invoked when a node sends a Join Cluster Request Message to a cluster leader, which is not cluster leader anymore for the layer a node wants to join. Such a Cluster Leader Validation Message is returned instead of the Join Cluster Response Message and can be considered as a negative acknowledgment. The reception of the Cluster Leader Validation Message invokes a rejoin of the node by sending a Join Query Message via the BS. This is repeated until a cluster leader has been found, which accepts the join.

Heart Beat Message

One of the most important message is the Heart Beat Message. This message transmits status information of a node to another node. A Heart Beat Message contains the cluster view of the sending node, which means, it delivers information about the cluster mates it knows and what RTTs it has measured to them. Every node sends such Heart Beat Messages periodically to the other cluster mates of the same layer, which store this information and therefore the nodes are aware of each other. Additionally, a cluster leader broadcasts in the cluster of which it is cluster leader its own Heart Beat Messages containing information of its upper layer. Therefore, nodes also have some information of their cluster leaders view of its cluster one layer above. This periodic broadcast is invoked by Reminder Messages as described in Section 3.3.2. Heart Beat Messages are periodically exchanged and also immediately sent from a cluster leader to its cluster mates if there was a change in the cluster configuration (for example if a node has joined or has left the cluster).

Depending on the source node type of a Heart Beat Message, the receiver performs different update operations. If the source is the receivers' cluster leader, the receiver adapts its own list of mates in this cluster by deleting or inserting nodes so that it has only nodes as cluster mates, which the cluster leader has reported. This list adaption at the receiver is important, since it can happen that due to timing problems, a cluster mate does not get aware of the departure of another cluster mate. Since the cluster leader is the only node, that has the power to tell which nodes are present in the cluster, Heart Beat Messages of the cluster leader invoke the adaptation of the node list at cluster mates.

The other case is that a sender of the Heart Beat Message is only a cluster mate in the same cluster as the receiver. In such a case, only the information about the view of the sender is updated at the receiver side. Updating the view means for example the update of the RTT values.

We implemented a mechanism that deletes a cluster mate from a cluster mates' list on a certain layer if there was no Heart Beat Message received from this cluster mate for a certain time on that layer. This avoids dead nodes remaining in the cluster mates' list of other cluster mates although the node has already left the cluster. Such a mechanism can also be used to model an ungraceful leave, which we did not evaluate in this thesis (for leave procedure information see Section 3.3.2).

Reminder Message

Reminder Messages are empty Self-Messages, and therefore do not leave the node. These message do not carry any information and are used to remind the node of doing a particular activity. In our implementation, we use several different Reminder Messages to invoke periodic broadcasts of Heart Beat Messages, to invoke a periodic refinement, and to invoke the departure of a node. Which action is invoked depends on the type of the Reminder Message. The Reminder Message are deactivated when a node leaves the NICE network completely.

Refinement Reminder Message

The Refinement Reminder Message is one kind of a Reminder Message, which is periodically send. When a cluster leader receives a Refinement Reminder Message, it has to invoke the refinement process. The refinement process consits of three operations, which are used to maintain the network according to the NICE topology constraints mentioned earlier in Section 2.3, where the NICE protocol is described: Cluster split, cluster merge, and cluster leader determination.

QoS Ping Message

The QoS Ping Message is the measurement tool in the NICE protocol. The message is derived from the RTT Measurement Message and is used to measure the RTT between two nodes. The message ID is important in order to identify the corresponding QoS Pong Message. First, in the QoS Ping Message, the pingTime field is set to the current simulation time and the destination address field is filled with the ID of the node, to which we want to measure the RTT. Then, the QoS Ping Message is sent. The QoS Ping Message is captured by the QoS Manager Filter at the destination node. The QoS Manager Filter returns a QoS Pong Message as response to the source of the QoS Ping Message.

QoS Pong Message

The node, that receives a QoS Ping Message generates immediately a corresponding QoS Pong Message taking the same message ID and copies the PingTime field value. Additionally, the QoS Manager Filter knows the RTT of the pinged node to the root. The QoS Manager Filter adds this value in the field rtt2root in the QoS Pong Message. The destination of the QoS Pong Message is set to the source of the received QoS Ping Message.

The node, that receives a QoS Pong Message first checks if the QoS Pong Message is a response to a QoS Ping Message previously sent by itself. If this is true, the RTT is calculated as

the difference between the current simulation time and the pingTime value reported in the QoS Pong Message. The reception of the QoS Pong Message automatically updates the concerning node entries at the receiving node.

Leave Reminder Message

The departure of a node is invoked by the scheduled Leave Reminder Message. When a node receives this message, it has to leave the NICE network completely. This can be more or less complex. In the simplest case, the node is only a cluster mate, and therefore has only to broadcast a Leave Message in the cluster it is located. The Leave Message causes the deletion of the node entry of the source node in all the destination nodes. The leaving node sends as a last step a so-called Node Dead Message, which triggers the Timeout Filter to shutdown the filter chain at the leaving node by blocking every message except Direct Messages. At the same time, it blocks all Overlay Messages and sends back a Timeout Message to the source of the blocked message.

The more complex case occurs, when a cluster leader wants to leave the NICE network as has to leave from at least two layers, and therefore from two clusters. In this case, we implemented the leave procedure on a layer-wise base. First, the cluster leader has to give up its cluster leadership and determine a new cluster leader for the cluster. Then, the situation can be broken down to a simple leave of a cluster mate as described in the preceding paragraph. It is obvious that the more clusters a cluster leader has, the longer the leave procedure takes.

When the root node leaves, the procedure is the same as when a cluster leader leaves, but instead of a cluster leader transfer, a root transfer has to happen. In case that the root node is the last node in the NICE network, the simulation ends with the departure of the root. When the last node has left the network, then there are no more scheduled events in the simulator, and therefore the simulation is terminated.

As a remark we have to mention that in this thesis only graceful departures were evaluated. A graceful departure means that every node, which wants to leave, announces its departure to its known cluster mates. In an ungraceful leave scenario, a node would simply leave the network by shutting down any communication with the network without informing the cluster mates. Since the communication in the NICE network is not reliable because of timing issues it can still happen that some leave messages do not reach its destination. Therefore, we have implemented a mechanism for checking when the last Heart Beat Message was received from a node. This mechanism is described in Section 3.3.2. This is only a simple mechanism, which may not be able to avoid the problem that the network may get seriously destabilized by frequent ungraceful leaves.

Timeout Message

As mentioned earlier, a node that has left the NICE network responds to any NICE message with a Timeout Message. This Timeout Message models a timeout in the source node of the message when it does not get any response from the destination node. The challenge is to invoke the correct reaction, when a Timeout Message is received at the original sender, since this depends on the message, which was answered with a the Timeout Message. Therefore, we need to keep track of the state a node is currently in. Moreover, it is required that the node does not change its

state until it receives either a timeout or another response to the messages sent during a certain state. By state, we mean especially the current refinement operation taking place.

3.3.3 Cluster Refinement Process

All the refinement operations are controlled in such a way that only one refinement operation per cluster leader and layer takes place at the same time. Without this control mechanism, it would lead to conflicting refinements and we risk a destabilization of the NICE network. Moreover, since all these operations are critical for the stability of the network, they were implemented using handshakes.

Cluster Split

The cluster split operation splits a cluster into two subsets if the cluster size is bigger than $3k-1$ where k is the cluster constant. The cluster is splitted in such a way that the maximum radius (in a graph-theoretic sense) of the new set of clusters is minimized. This is achieved by minimizing the eccentricity within both subsets. The eccentricity of a node v is the greatest RTT between v and any other node. We used a permutation generator template class [20] to calculate all subset configurations. For both of the subsets, a cluster leader has to be determined. For the subset in which the invoking cluster leader is present, it remains acting as the cluster leader. In the other subset, a new cluster leader is determined following the same procedure like in the cluster leader refinement operation described later. This is not exactly the same split procedure as described in [1], where for both subsets a new cluster leader is determined, since this approach can lead to conflicting procedures, which affects the stability of the network. Such a conflict can occur when the invoking cluster leader is also root. Then, the split causes a root transfer and a split at the same time. The problem then could be that the new cluster leaders of both of the clusters may not join the upper layer, since the root is still not ready to handle the joins of the subset cluster leaders. Our modification is suboptimal, but the only drawback is that the optimal cluster leader is only determined the next cluster leader refinement process. When the most appropriate subset configuration is found, a Split Request Message is sent to the newly determined cluster leader. If this node is able to become cluster leader, it answers with a Split Accept Message, otherwise a Split Deny Message is delivered to the source (the split invoking node). Receiving a Split Accept Message triggers sending of the effective Split Messages, which are sent to the cluster mates of the subset, which have to join the new cluster leader. If a Split Deny Message is received, the node which wanted to invoke the split waits for 0.5 seconds and retries another split. It is necessary that a split happens as soon as possible, since if the cluster gets too big, the calculation of the permutation can become very time consuming. We experienced during our simulation that if the clusters have more than 20 nodes, the calculation time of the permutations were not manageable.

Cluster Merge

If a cluster leader detects that its cluster is too small (cluster size lower than k), it has to merge its cluster with another cluster. In such a case, the cluster leader looks for the closest cluster

leader in terms of RTT on the same layer. With this closest cluster leader, a merge handshake is invoked. First, a Merge Request Message is sent from the current cluster leader to the closest cluster leader. If a cluster leader receives a Merge Request Message, it can either accept or deny it. A Merge Request Message is denied if for example the closest cluster leader is not cluster leader anymore for the layer on which the merge should be performed or if the cluster size of the merged cluster exceeds the upper cluster size boundary. This avoids that the merged cluster has to be splitted immediately after a merge. If a cluster leader accepts a merge request, it acknowledges the Merge Request Message with sending a Merge Accept Message to the source of the request. Receiving a Merge Accept Message makes the merge initiating cluster leader to give up its cluster leadership by sending a Leave Message on the layer where it was cluster leader. Furthermore, a Merge Message is broadcasted in the cluster where it was cluster leader. This invokes the cluster mates to join the chosen closest cluster leader, which becomes their new cluster leader.

Cluster Leader Determination

A cluster leader determination is done if a cluster leader detects that it is not the best cluster leader anymore and has to be replaced. This operation permits that the NICE topology is RTT optimized. The cluster leader has the minimum maximum RTT to all other nodes in the cluster. Like a split operation, the new cluster leader is determined by calculating the graphic-theoretic center of the cluster. In the simple case, the node which has to determine a new cluster leader is only a cluster leader and not the root node. The current cluster leader initiates a cluster leader transfer handshake by sending a Cluster Leader Transfer Request Message to the new cluster leader candidate for the layer in which the cluster leader transfer should occur. Like with the other refinement operations, the receiver of this message can either deny or accept it. In case of an accept, the new cluster leader has to join a cluster in the next higher layer. Therefore, it sends a Join Query Message via the BS and joins a cluster in the upper layer. In the meantime, it also sends a Cluster Leader Transfer Accept Message to the old cluster leader. Receiving this message, the old cluster leader can be sure that the chosen node is about to become cluster leader and that itself has to give up its own cluster leadership on this layer. This is realized by first sending a Cluster Leader Transfer Message to all cluster mates in the cluster of which it was cluster leader and then sending a Leave Message in the cluster in the layer above. The mates in the cluster where the cluster leader has changed receive the Cluster Leader Transfer Message, which tells them to join the new cluster leader. They send a cluster Join Cluster Request Message, which is confirmed by the new cluster leader.

In the more complex case, the cluster leader is also the root node, and therefore a root transfer has to be invoked. Basically, the procedure and the message exchange is the same like in the simple case of a cluster leader transfer. The additional request used is a Root Transfer Request Message and is answered either with a Root Transfer Accept Message or a Root Transfer Deny Message. In case of accepting the root transfer, also the BS has to be informed about the root transfer. This is done by sending a Root Transfer Message to the BS. The BS replaces its old root node entry with the new one. The information about the root transfer for the cluster mates is done as in the simple case spread with a Cluster Leader Transfer Message since the cluster mates are not aware that the root has changed.

3.4 Quality of Service for Overlay Multicast (OM-QoS) Applied to NICE

This Section explains how we implemented Quality of Service (QoS) in NICE. Like mentioned in Section 2.4, we use two different approaches to bring QoS to the NICE protocol. The goal of both of these approaches is that all multicast paths from the root to the leaves support QoS.

The implementation of the protocol-dependent QoS approach slightly modifies the protocol in such a way that the cluster leader selection is not done using RTT optimizations anymore but using QoS classes instead. This means that the node with the highest QoS class in a cluster has to become cluster leader, and the node with the highest QoS class of all nodes has to become root. The determination of a new cluster leader becomes simpler, because the cluster center calculation has not to be done anymore. Now, only the node with the highest QoS class among the cluster mates has to be found. Since the current cluster leader knows all the QoS classes of its cluster mates, this procedure is reducible to a simple search. If the cluster leader change happens because of a cluster leader refinement, the old cluster leader will remain in the cluster and has only to take into account cluster mates that have a higher QoS class than itself. In case that the node leaves the cluster completely, the selection of a new cluster leader depends on the remaining cluster mates. Then, the cluster mate with the highest QoS class is selected as new cluster leader. Another part affected by the QoS class-awareness is the selection of the cluster leader for both subsets during a split, since it includes a cluster leader selection as well.

The protocol-independent QoS approach is realized by using the Layered OM-QoS Framework presented in [17, 16]. This framework manages independent layers - each with an individual Overlay Network. In our case, each layer consists of a NICE network with nodes of one single QoS class. For this approach, the NICE protocol is not changed and operates in the native, RTT-optimized mode. The connection between the layers and the Layered OM-QoS Framework is made by an extension of the BS, which is not part of any network. The technical extension of the BS consists of the fact that the NICE Bootstrap Filter inherits from the Overlay Boostrap Filter. This allows management of several root nodes. With the inheritance, the BS maintains a list, which holds a root node for each QoS class. By inheriting also the Root Transfer Message from the Layered OM-QoS Framework's Root Transfer Message, the Overlay Framework Filter becomes aware of root transfers. The Overlay Framework Filter then sends a notification to the Overlay Bootstrap Filter to update the root list.

We implemented another mode called End-to-End Delay Optimization (E2E-Delay Optimization) for NICE using an extension of the QoS Manager Filter described in [21]. This extension allows us to guarantee RTT to root node constraints. Each node receives a maximal RTT to root node value, which is randomly chosen out of an interval between a maximal and minimal RTT value specified in the Node Init Message. This value affects the join process of NICE nodes. In contrast to the join process in the native NICE implementation, the joining node in the E2E-Delay Optimized mode selects the cluster leader according to the received maximal E2E-Delay value. It selects the cluster leader, which has the closest RTT to root value that is below the constraint. If none of the available cluster leaders fulfills the maximal E2E-Delay constraint, the joining node selects the cluster leader with the smallest RTT to the joining node.

Chapter 4

Results and Evaluation

4.1 Simulation Scenario

To evaluate our implementation, we used OMNeT++ v3.3 running on a Fedora 8 Linux using Kernel 2.6.26.8-57 on a x86_64 machine. The common parameters for all simulations were the random seeds and the distance matrices. Each seed is an integer number that controls the Mersenne Twister Random Number Generator (RNG), which is the default RNG in OMNeT++ [22]. The Mersenne Twister RNG has a period of $2^{19937} - 1$, is very fast, and 623-dimensional equidistribution property is assured. Therefore, a particular seed influences any randomly chosen number in an interval. In our implementation, we choose for example the timestamp for sending a Leave Reminder Message randomly in a certain interval. The random values are important in cases where a request is denied and a retry has to be done. In some of these scenarios, the resending of such requests is scheduled randomly in a time interval to avoid the same situation again.

To simulate a network topology, we used symmetric distance matrices, which contain the delay between each pair of nodes. These matrices were generated using the real-world topology generator BRITE [18]. To generate such network topologies, we fed BRITE with the following configuration file.

```
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
```

The output of BRITE is a topology file containing the nodes and edges, where distance between the nodes are specified with a delay value in ms. We calculated the shortest path for every pair of nodes in terms of delay. These values were then stored in our distance matrix as the delay for a certain edge between two nodes. We computed 13 distance matrices, each based on a topology file.

Table 4.1 lists the scenarios with their modes and configuration of the relevant parameters. For all the evaluations, the same number of random seeds is used. We vary the cluster constant k, which controls the boundaries of the cluster size. The number of QoS classes is only interesting for the Layered OM-QoS Framework, since there the number of QoS classes determines the number of dedicated NICE networks. We evaluated the Layered OM-QoS Framwork with 32 and 256 QoS classes, since in [16] the Layered OM-QoS Framework is evaluated with other Overlay P2P protocols using 32 and 256 QoS classes too.

For each evaluation, we used three random seeds as listed in Table 4.2 and the 13 different distance matrices listed in Table 4.3. Each of these 39 seed-matrix-combinations were combined with different number of nodes per NICE network. We simulated NICE networks ranging from 100 to 2000 nodes with interval steps of 100 nodes. This led to 780 simulation runs per evaluation. All the results shown are extracted from the simulation data after removing 0.5% of the lowest and highest values to reduce the impact of outliers.

Table 4.4 explains the evaluated values. The domain of the values is depicted in the particular graph where the results are presented.

Table 4.1: Evaluated Scenarios

NICE Mode	**Cluster Constant** k	**# QoS classes**	**# Seeds**
Native (RTT)	3	256	3
Native (RTT)	4	256	3
Native (RTT)	5	256	3
Static QoS (Hard-QoS)	5	256	3
Dynamic QoS (Soft-QoS)	5	256	3
Layered QoS (Protocol Independent)	5	32	3
Layered QoS (Protocol Independent)	5	256	3
Static QoS (E2E-Delay Optimization ON)	5	256	3
Native (RTT) (E2E-Delay Optimization ON)	5	256	3

Table 4.2: Random Seed Numbers

1768507984
33648008
1082809519

Table 4.3: Delay Properties of Distance Matrices in ms

Matrix	**min RTT** (ms)	**average 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

Table 4.4: Explanation of Measured Values

Value	**Explanation**
FanOut	Describes how many cluster mates a cluster leader has to serve with multicast data
HopCount	Counts for every node how many hops have to be taken to reach the root of the multicast tree
LeaveDuration	Time in seconds needed by a node to leave the NICE network
JoinDuration	Time in seconds needed by a node to join the NICE network
Node2RootQoS	Percentage of paths, which fulfill the requirements to support QoS
Node2RootRTT	The Round Trip Time from the node to the root of the multicast tree in seconds
NumOfClusterMates	Measures the number of cluster mates per cluster leader
PercentageOfMcast	Percentage of Multicast Messages received per node
RTT2RootSatisfiedNICEInitJoin	Percentage of paths that fullfill the E2E-Delay constraint after initial join
RTT2RootFailedDifferenceInitJoin	Difference of the RTT to root to the E2E-Delay constraint after initial join in seconds
NextHopRTT	RTT from cluster leader to cluster mates in seconds
RTT2RootSatisfiedInitJoin	Percentage of nodes for which NICE can satisfy the given RTT constraint during the initial join process

4.2 Comparison of Results

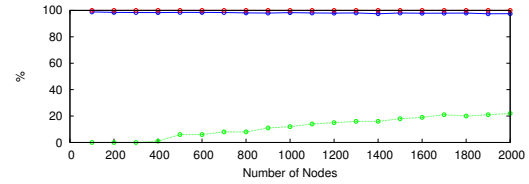
To check whether our implementation and modifications work, we evaluated the percentage of received Multicast Messages in each node for every scenario as presented in Table 4.1. According to the graphs shown in Fig.4.1, we can conclude that our NICE protocol implementation including all modifications works well. These Multicast Messages are sent from the root to every leaf node. Therefore, this measurement is an indicator of quality of the NICE network topology in terms of stability. The loss of multicast messages is caused by nodes, which currently do not have a cluster leader or do not receive certain multicast messages from their cluster leaders. Such a situation may occur during a rejoin of a cluster leader to another cluster, which may first deny the join. Such a deny of a join happens for example, when the cluster leader to be joined is involved in a cluster leader refinement process itself. Since NICE uses a tree topology, it is clear that if such a situation occurs in a cluster in a layer near the top layer, all cluster mates of the rejoining node are affected by the loss. In contrast, such a situation close to the layer 0 affects less cluster mates, and has therefore only local significance. The reason for the variability of the minimum curves in Fig. 4.1 is that situations like discribed above are dependent on where in the NICE tree structure they happen and for how long they last.

4.2.1 Native NICE

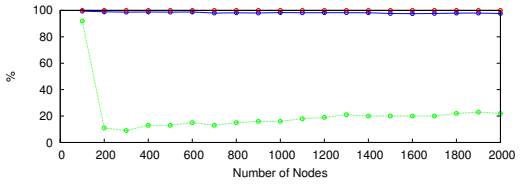
If we use NICE without any modification, we talk about the native mode of NICE. This means that NICE optimizes the join procedure of the nodes regarding the hop-by-hop RTT values to the potential cluster leaders and selects the cluster leader according to the graphic-theoretic center calculation. We evaluated NICE in this mode for three different values of the cluster constant k, with $k \in \{3, 4, 5\}$.

The variation of the cluster constant shows its effect in the cluster size, since the cluster size is depending on the cluster constant. Due to the fact that a smaller cluster constant leads to less cluster mates per cluster, there are more clusters built out of the same number of nodes compared to a higher cluster constant. The more clusters there are, the higher is the chance to have more layers. The reason for this relation is that if a cluster gets splitted, a new cluster is created and therefore, a new cluster leader joins a cluster in the layer above. This fills the cluster in the layer above. Moreover, the chance that in this layer a cluster split has to be invoked increases. Potentially, this creates another layer, since if a new cluster leader joins the root layer, the root creates a new top layer. An additional layer, increases the hop count. Therefore, the hop count is higher when using a small cluster constant than when a high cluster constant. Figure 4.2 shows the hop count from any node to the root. One can see that the hop count for $k = 3$ is the highest.

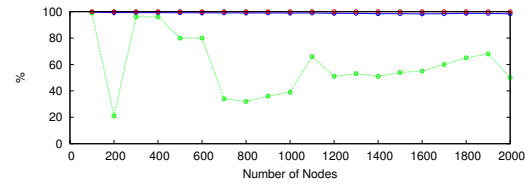
Figure 4.3 depicts the dependency of the cluster size and the cluster constant. The cluster size is independent of the number of nodes present in a NICE network, which explains that the average cluster size is stable through all simulations with different number of nodes. The maximal number of cluster mates per cluster, which are managed by a cluster leader, is slightly lower than $3k - 1$ and the minimal number of cluster mates is lower than k. This arises from the circumstance that a cluster leader avoids to exceed the upper boundary very strictly. This means that a split happens before the cluster really exceeds the maximum size of $3k - 1$ cluster mates. If the cluster size falls below the lower boundary k of the cluster size, the cluster leader has to



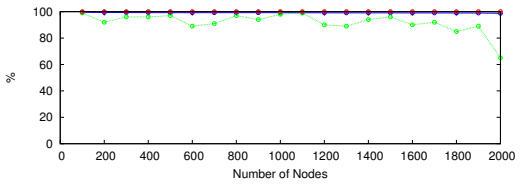
(a) Native NICE with $k = 3$



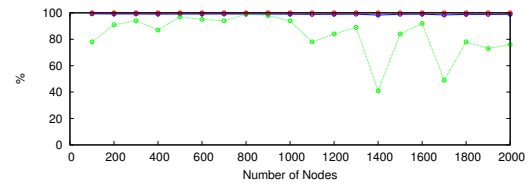
(b) Native NICE with $k = 4$



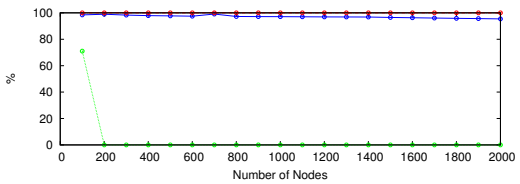
(c) Native NICE with $k = 5$



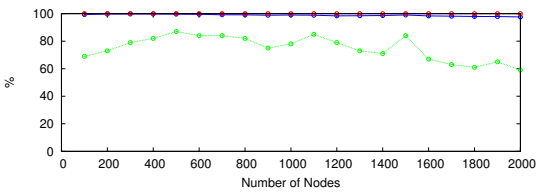
(d) NICE with Hard-QoS Support with 256 QoS Classes



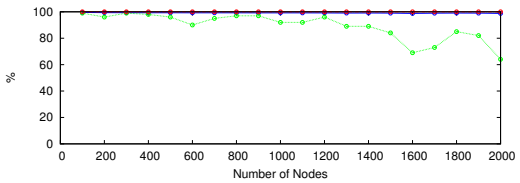
(e) NICE with Soft-QoS Support with 256 QoS Classes



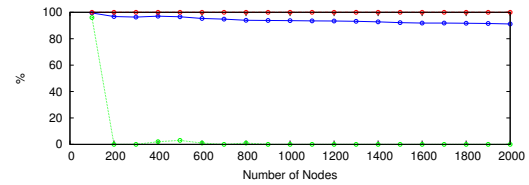
(f) NICE with Layered QoS with 32 QoS Classes



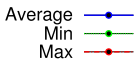
(g) NICE with Layered QoS with 256 QoS Classes



(h) NICE with E2E Support and Hard-QoS with 256 QoS Classes



(i) Native NICE with E2E Support



(j) Legend

Figure 4.1: Percentage of Received Multicast Messages for all Evaluations

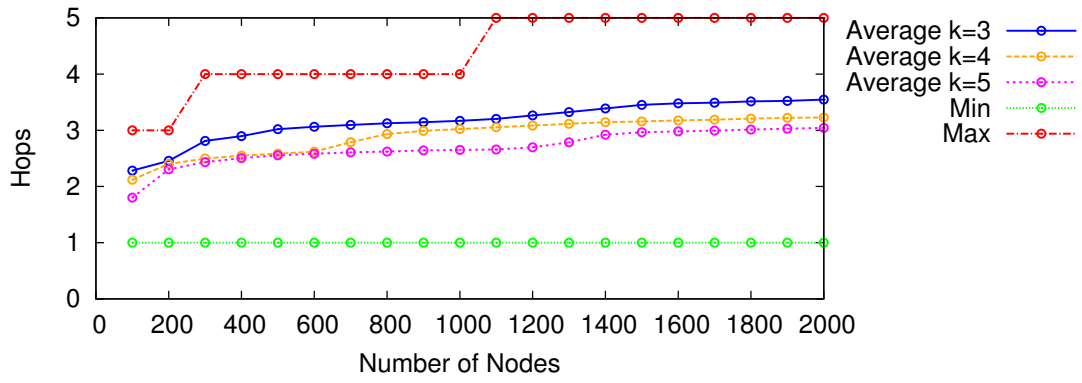


Figure 4.2: Hop Count

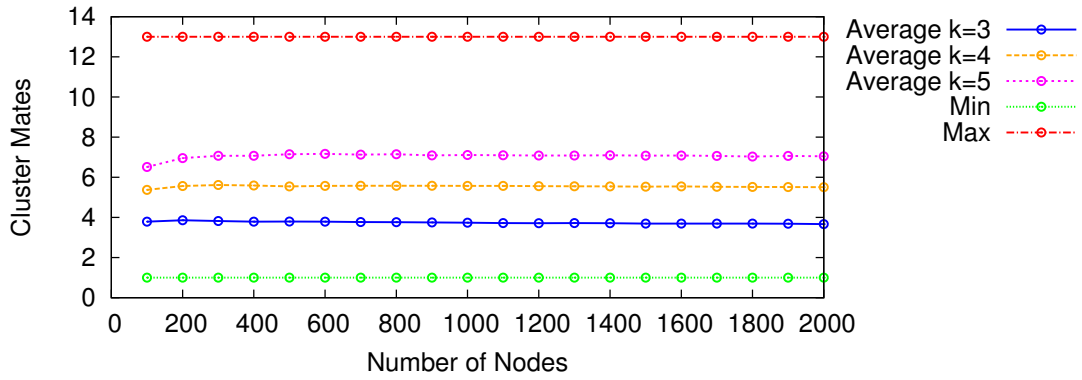


Figure 4.3: Number of Cluster Mates

invoke a cluster merge. Unlike the joining of nodes, the departure of nodes can not be denied by the cluster leader (even not during a cluster merge). It can happen that there are clusters, whose cluster size falls below the lower cluster size boundary k, since cluster mates leave the cluster regardless of the cluster size boundaries.

As shown in Fig 4.4, one can see that the fan out is higher with higher cluster constant. Once again, the reason for this effect is the cluster size dependency on the cluster constant. Since the cluster size using a low cluster constant ($k = 3$) is small, the fan out is low compared to a higher cluster constant. Having smaller clusters with a low number of cluster mates reduces the load on the cluster leaders. Moreover, smaller clusters lead to more cluster leaders in the whole NICE network. This means that the overall load of distributing data via the cluster leaders is smaller having a small cluster.

The maximum value is always constant determined by the fan out of the root node. Since the root node is present in one cluster on every layer, it has to serve the highest number of cluster mates. The load of the root node is also one of the main problems of the NICE protocol. There is an approach to reduce the load on the root node in NICE as presented in [23] using delegation

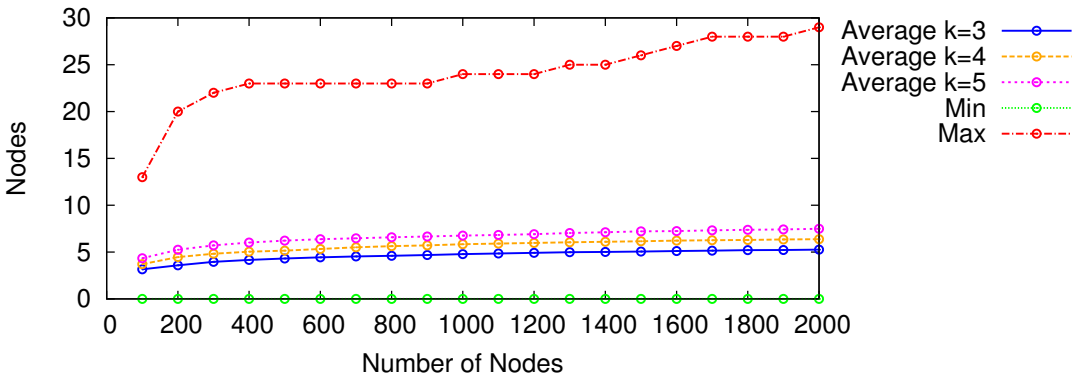


Figure 4.4: Fan Out

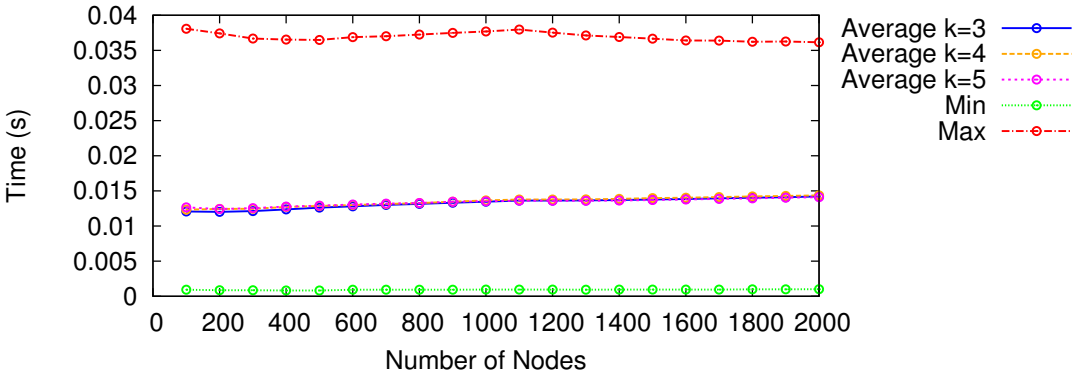


Figure 4.5: Next Hop RTT

nodes. We did not consider this approach for this evaluation.

We also evaluated our implementation of the cluster building process, measuring the next hop RTT. This value indicates the quality of the cluster in terms of RTT. It measures the RTT from the cluster mates to their cluster leader. Figure 4.5 shows that the average RTT from a cluster leader to its cluster mates is much lower compared to the average RTT between two nodes as shown in Table 4.3. This fact shows that the locality of the clusters, and therefore th topology of the underlying network is well considered in terms of RTT among the cluster mates. Moreover, the cluster building process is independent from the cluster constant k.

Since there are more layers using a small cluster constant, the join and leave procedure takes longer. This can be explained with the layer-wise announcements respectively the requests from the top to the bottom layers during joining and leaving of nodes.

The impact of the cluster constant k is also visible in Fig. 4.6. It shows that the join duration is higher when using a smaller cluster constant. The reason for this is like mentioned before that a smaller cluster constant results in more layers. Since the joining process of a node consists of selecting the most appropriate cluster leader in each layer until the joining node reaches layer

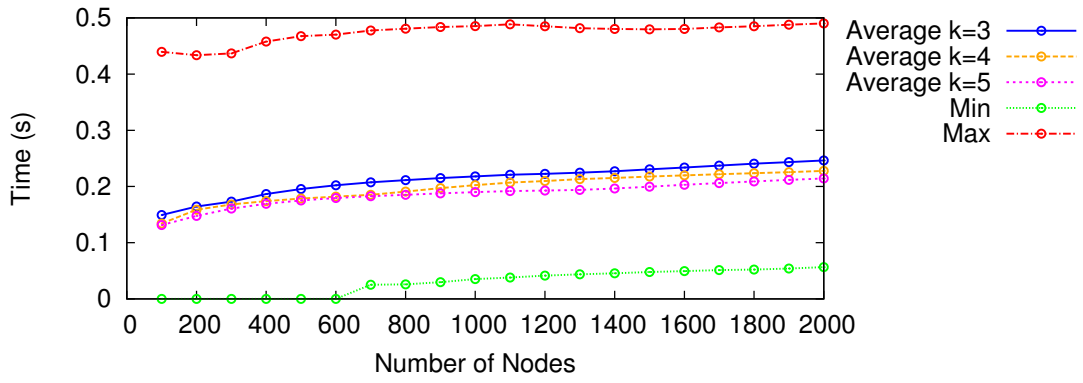


Figure 4.6: Join Duration

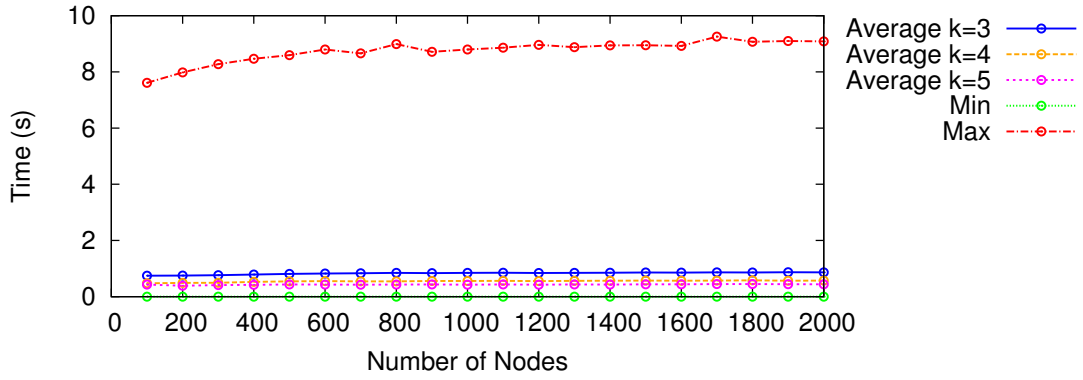


Figure 4.7: Leave Duration

0, the more layers there are, the more cluster leaders have to be contacted during the joining process until the joining node reaches layer 0. Therefore, a higher number of layers implied by a low cluster constant k prolongates the joining duration.

Compared to the joining time, the average leaving time of a node is much higher. This is depicted in Fig. 4.7. This effect can be explained with a cluster leader, that has to transfer its cluster leadership to another node before it can leave the NICE network. Moreover, if a cluster leader is present in multiple layers, this process is invoked layer-wise, which also needs additional time.

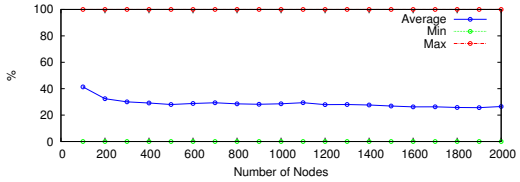
4.2.2 NICE with QoS Support

The following Sections present the results using NICE with different QoS modes. In all of the QoS modes, a cluster leader is selected according to the QoS class, which is an abstract representation of one or more QoS parameters.

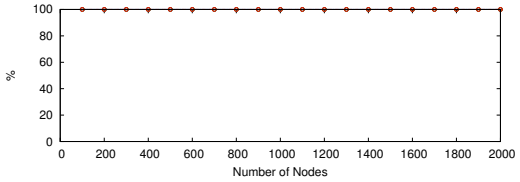
Protocol-Dependent OM-QoS Approach: Hard-/Soft-QoS

The hard- and the soft-QoS scenarios consist of nodes, which get a QoS class assigned as a randomly chosen integer from the interval 0-255. Each node then keeps this QoS class during the whole simulation. In hard- and soft-QoS cases, the NICE network is built according to the QoS-class hierarchy explained in Section 2.4. The difference between hard- and soft-QoS is that in the soft-QoS case, a sudden QoS path violation is simulated. The node, that detects this violation then needs to find a new cluster leader, that fulfills again its QoS requirements. The QoS path violation is simulated by sending a QoSNotSatisfited Message to the node, which then has to look for a new cluster leader, that fulfills the QoS requirements of the node. We evaluated this scenario with a 50% chance for every node to receive such a message in the time interval of 10s to 100s after it has joined the NICE network.

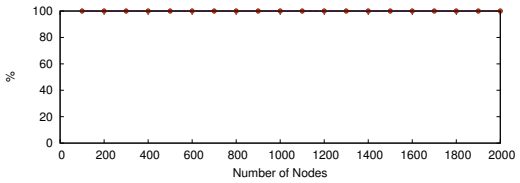
We compare how many paths from the root node to the receiver nodes in a native NICE network already support the requirements for QoS. Therefore, we compare the results from the evaluation of the native NICE using the cluster constant $k = 5$ to the results of the NICE network operating in the QoS-enabled modes with hard and soft-QoS. The assumption that NICE running in QoS modes supports QoS for 100% of the paths is confirmed as shown in Fig. 4.8.



(a) Node to Root QoS in Native NICE



(b) Node to Root QoS in Hard-QoS NICE with 256 QoS Classes

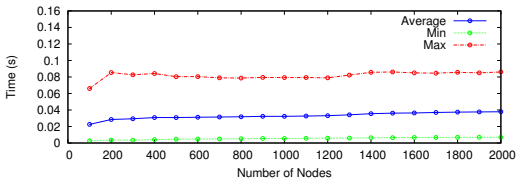


(c) Node to Root QoS in Soft-QoS NICE with 256 QoS Classes

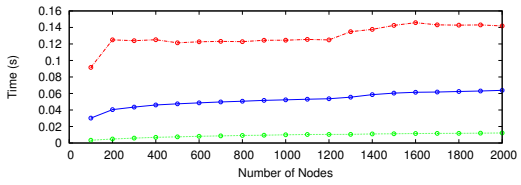
Figure 4.8: Fulfilled Node to Root QoS

This shows that our implementation of the protocol-dependent QoS approaches supports QoS as presented in Section 2.4.

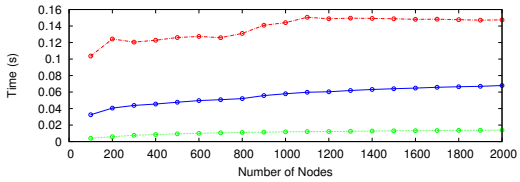
Figure 4.9 depicts how the RTT from any node to the root node behaves. The average is only slightly higher in the QoS mode than in the native NICE. In these QoS modes only the cluster leader selection is modified. Therefore, the NICE tree is built according to the value of the QoS classes. This means that a joining node still selects the cluster to join regarding the locality of the cluster in the underlaying network. Since cluster leaders are in this case selected according to the QoS class, the difference to the native NICE is visible by the maximum curve. In native NICE, the cluster leader selection is based on the graph-theoretic center calculation in terms of RTT, while in the QoS modes, the top cluster leaders are not necessarily close to the root node in terms of RTT and generally not close to each other on a certain layer.



(a) Node to Root RTT in Native NICE



(b) Node to Root RTT in Hard-QoS NICE with 256 QoS Classes



(c) Node to Root RTT in Soft-QoS NICE with 256 QoS Classes

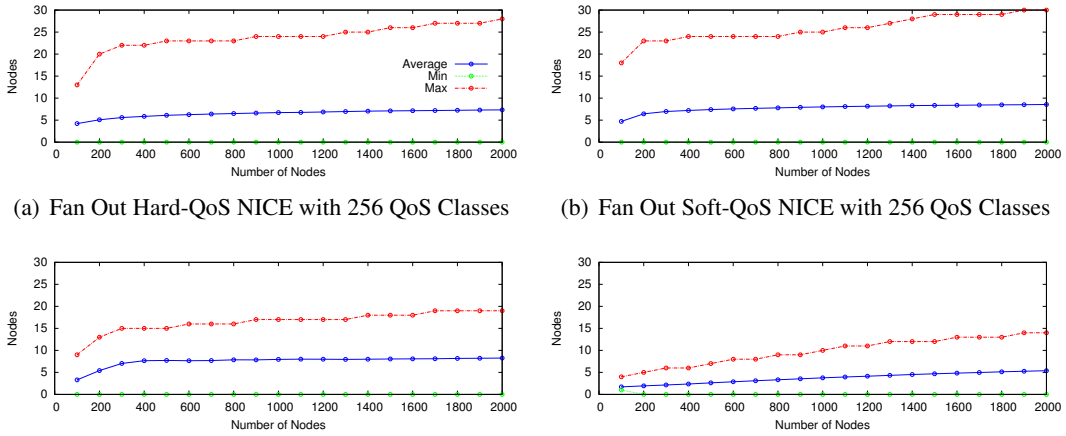
Figure 4.9: Node to Root RTT

Protocol-Independent OM-QoS Approach: Layered OM-QoS Framework

For this evaluation, we used the Layered OM-QoS Framework as proposed in [17, 16]. This framework provides a protocol-independent support for QoS. In general, with a constant number of nodes, we can say that the NICE networks in each framework layer become smaller, the more QoS classes are being used. The goal of the Layered OM-QoS Framework is to support QoS with no modification of the Overlay P2P protocol in the framework layers. One can see that the use of the Layered OM-QoS Framework does not influence NICE in a negative way.

As shown in Fig. 4.10, the fan out can be reduced using more QoS classes. Since for each QoS class, a dedicated NICE network is built, the number of nodes per NICE network is reduced. Therefore, we have smaller NICE networks with less clusters. This is the reason why the fan out for any node is dramatically reduced using more QoS classes.

If we compare the protocol-independent QoS approach with the protocol-dependent QoS approach regarding the fan out, we can say that the load in the protocol-dependent approach is much higher than using the protocol-independent QoS approach.



(a) Fan Out Hard-QoS NICE with 256 QoS Classes

(b) Fan Out Soft-QoS NICE with 256 QoS Classes

(c) Fan Out Layered-QoS with 32 QoS Classes

(d) Fan Out Layered-QoS with 256 QoS Classes

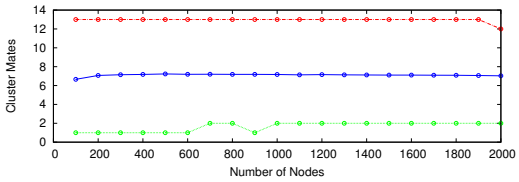
Figure 4.10: Fan Out of NICE Nodes in Different QoS Modes

An interesting issue is the linear increase of the average of the fan out using the Layered OM-QoS Framework with 256 QoS classes. It shows that due to the high number of NICE networks, the clusters in each dedicated NICE network do not reach the critical size where a split would be necessary. Since this leads to less clusters and therefore to less layers, the fan out is low. In case of 32 QoS classes in the Layered OM-QoS Framework, a similar linear increase can be observed until 400 nodes. Beyond this point, the average fan out is stable. This stability is explained by the stability of the size of the clusters. Having more than 400 nodes in the Layered OM-QoS Framework means that there are enough nodes per dedicated NICE network to build balanced clusters. Since there are more balanced clusters than in the previous scenario, the weight of the average fan out increases, which is the reason for the stability of the average fan out with more than 400 nodes in the scenario.

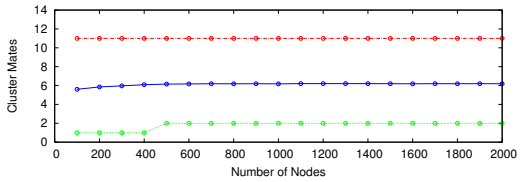
We do not evaluate what number of QoS classes is the most suitable, but it is clear that the number of nodes should be much higher than the number of QoS classes, since otherwise the NICE networks in the layers of the framework are too small.

The explanation for the fan out characteristics is also supported by the number of cluster mates as statistically depicted in 4.11. In the hard- and soft-QoS mode, all nodes are in the same NICE network. Therefore, the clusters are balanced concerning their size, which is shown by the stability of the average number of cluster mates in both of these cases. As already mentioned, the clusters in the Layered OM-QoS Framework using 256 QoS classes do not reach the critical cluster size, and therefore, no splits are necessary.

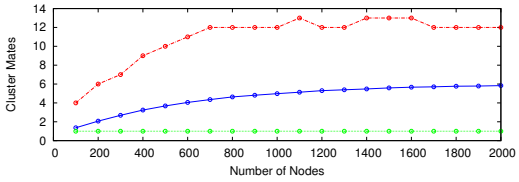
In case of the Layered OM-QoS Framework with 32 QoS classes, we see that there are cluster splits. The maximum curve gets more or less stable near the maximum of $3k-1$ with $k = 5$. This and the fact that the average number of cluster mates are more flattened are indicators that the cluster size gets more balanced. It is obvious that in hard- and soft-QoS scenarios, where we have only one single NICE network, the clusters are already balanced when using 100 nodes. This is because all nodes join to the same NICE network, and therefore the NICE protocol has



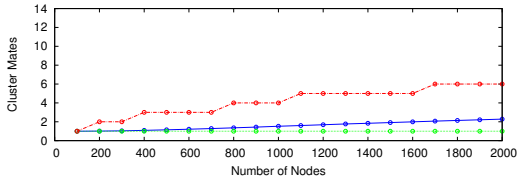
(a) Number of Cluster Mates Hard-QoS NICE with 256 QoS Classes



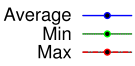
(b) Number of Cluster Mates Soft-QoS NICE with 256 QoS Classes



(c) Number of Cluster Mates Layered-QoS with 32 QoS Classes



(d) Number of Cluster Mates Layered-QoS with 256 QoS Classes

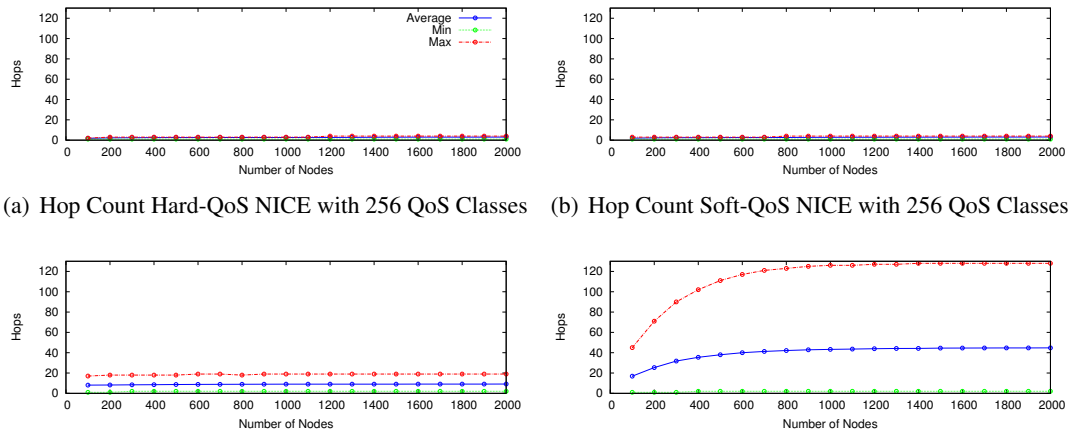


(e) Legend

Figure 4.11: Number of Cluster Mates of NICE in Different QoS Modes

no problems to build balanced clusters by invoking refinement operations.

The Layered OM-QoS Framework has one severe disadvantage, which is shown in Fig. 4.12. The hop count increases dramatically when using more dedicated NICE networks. This can be interpreted as the overhead of the Layered OM-QoS Framework. The reason for this is that the layers with the dedicated networks are linked together via the gateway nodes using a chain, where each root node is a gateway node. This implies that the multicast traffic coming from the sender has to be routed along this chain to all the dedicated networks and additionally to every node in the dedicated network. To limit the number of hops in the Layered OM-QoS Framework, additional gateway links have been introduced. The sender sends the traffic not only to one gateway node, but also to a gateway node which is further away down or up the chain. Obviously, each gateway node adds one hop to the path. Therefore, the hop count depends on the number of layers in the Layered OM-QoS Framework, which depends on the number of QoS classes used. Due to this fact, it is no surprise that the hop count of the Layered OM-QoS Framework evaluation with 256 QoS classes is much higher than the hop count in the same evaluation with 32 QoS classes. The additional gateway links in the Layered OM-QoS Framework explain why the maximum hop count is not as high as the number of QoS classes used in the framework.



(a) Hop Count Hard-QoS NICE with 256 QoS Classes

(b) Hop Count Soft-QoS NICE with 256 QoS Classes

(c) Hop Count Layered-QoS with 32 QoS Classes

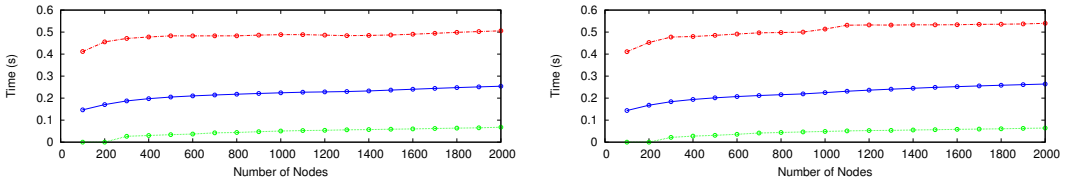
(d) Hop Count Layered-QoS with 256 QoS Classes

Figure 4.12: Hop Count of NICE Nodes in Different QoS Modes

Figure 4.13 shows the overview of the join duration for all QoS-enabled NICE modes. The observation is that the smaller the NICE networks are, the shorter the joining process is. This Figure also shows that the Layered OM-QoS Framework does not delay the joining process. On the contrary, the effect of having small dedicated NICE networks, makes the nodes join faster. From the implementation view of the protocol-independent QoS approach, there is no difference to the protocol-dependent QoS approach concerning the joining procedure of nodes. Since in both approaches the bootstrap node knows which node is root node, respectively which nodes are root nodes in case of the Layered OM-QoS Framework, the appropriate root node can be reported to the joining node. This fact wipes out any direct influence of the Layered OM-QoS Framework to the join duration. The only effect the Layered OM-QoS Framework has is that the dedicated NICE networks become smaller.

If we consider Fig. 4.14 showing the leaving process duration of the nodes, we can see the impact of small dedicated NICE networks once again. The less nodes the dedicated NICE network has, the faster the nodes can leave. This effect is apparent regarding the maximum curve. The maximum leave duration is the leave time of the root node. How fast the root leaves is depending on the number of layers the dedicated NICE network has, because the root has to leave a cluster in every layer. The more QoS classes are used in the Layered OM-QoS Framework, the less nodes are in a dedicated NICE network. The less nodes are in a NICE network, the less layers are in the network. Therefore, the leave duration in the Layered OM-QoS Framework using 32 QoS classes is smaller than the one using 256 QoS classes. In case of the Layered OM-QoS Framework with 32 QoS classes, there is a strong slope in the maximum curve between 400 and 600 nodes. This apparently arises from an additional layer in one of the dedicated NICE networks.

Figure 4.15 shows the RTT of the nodes to the root node. Since the Layered OM-QoS Framework adds many hops along the paths from the node to the root node as shown in 4.12, it is obvious that the RTT to the root node raises as well.

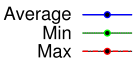


(a) Join Duration Hard-QoS NICE with 256 QoS Classes

(b) Join DurationSoft-QoS NICE with 256 QoS Classes

(c) Join Duration Layered-QoS with 32 QoS Classes

(d) Join Duration Layered-QoS with 256 QoS Classes



(e) Legend

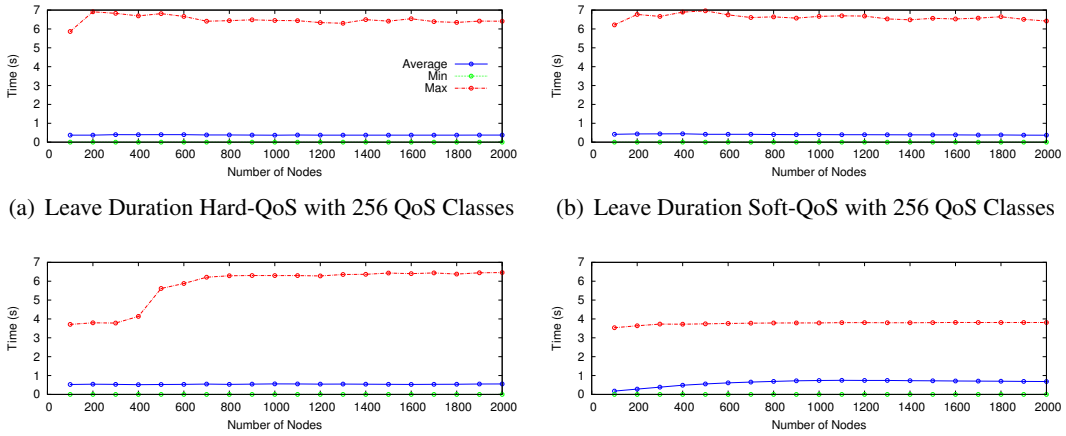
Figure 4.13: Join Duration of NICE Nodes in Different QoS Modes

As shown in Fig. 4.8 for the protocol-dependent QoS approach, also the protocol-independent QoS approach supports QoS through all paths as depicted in Fig. 4.16.

End-to-End Delay Optimization

For this evaluation we first had to determine, which RTT range (from any node to the root node) is the most tight. Therefore, we performed simulation runs with different RTT ranges. We encountered that the thightest range is from 25ms to 50ms. This means that using this range, we achieve that the E2E-Delay Optimization can fulfill almost 100% of the RTT to root constraints of every node. Simulation with higher range boundaries showed that 100% of the RTT to root constraints are already fulfilled without E2E-Delay-Optimization. On the other side, using a range below 25ms to 50ms decreased the percentage of RTT to root constraint satisfaction even with using the E2E-Delay-Optimization.

In NICE using the E2E-Delay Optimization, very node chooses a value from this range randomly, which is used as the maximal RTT constraint between itself and the root node. It is not surprising that this range is optimal. If we consider the average RTT between two nodes in Fig. 4.5 for $k = 5$, which is around 12ms to 14ms, we notice that the range of 25ms - 50ms matches with about two to four hops. As our measurement in 4.2 for $k = 5$ shows, the average hop count is two to four hops. Therefore, we can assume that using this range leads to almost 100% RTT to root constraint satisfaction when using the E2E-Delay Optimization in combination with the native NICE mode.



(a) Leave Duration Hard-QoS with 256 QoS Classes

(b) Leave Duration Soft-QoS with 256 QoS Classes

(c) Leave Duration Layered-QoS 32 QoS Classes

(d) Leave Duration Layered-QoS 256 QoS Classes

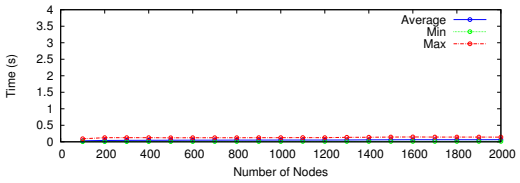
Figure 4.14: Leave Duration of NICE Nodes in Different QoS Modes

We evaluate the E2E-Delay Optimization using the native NICE mode. We also use E2E-Delay Optimization in combination with the hard-QoS mode. With this setup, we are able to optimize the QoS mode further to additionally support RTT constraints when joining the NICE network. In general, we can say that the E2E-Delay Optimization in combination with the native NICE mode leads to better results, since in native NICE mode, the paths are already RTT optimized while in any QoS mode, the QoS class is used to determine cluster leader and to build the NICE network topology.

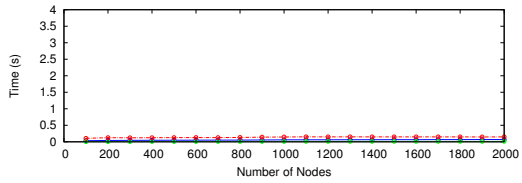
In Fig. 4.17 we can see that using our E2E-Delay Optimization and the given RTT range, we can achieve that almost 100% of the nodes join a cluster, which can satisfy the RTT constraints of the nodes to the root node. Without the E2E-Delay Optimization, the satisfaction is much lower, between 60% to 90%.

If we consider the nodes for which the NICE network could not fullfill the nodes' RTT constraints, the difference between the RTT constraint value and the effective RTT to the root node in the worst case are quite small for NICE with enabled E2E-Delay Optimization compared to the scenarios where the E2E-Delay Optimization is disabled. This is shown in Fig. 4.18. The difference in case of disabled E2E-Delay Optimization is between 29ms and 75ms in the worst case. Looking at the same value in the case of enabled E2E-Delay Optimization, we see that the worst failure of the constraint is between 18ms and 28ms. In general, the difference is smaller using the native NICE mode, since there the NICE topology is built regarding the RTT between the nodes, while in QoS mode, the QoS class is used. This fact allows the conclusion that the E2E-Delay Optimization is an improvement for NICE to support RTT requirements of a node to the root node in the NICE network.

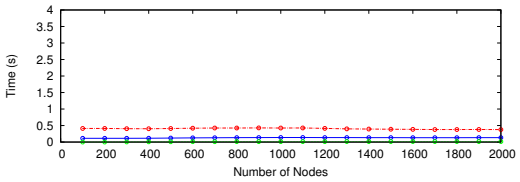
Since the E2E-Delay Optimization only affects the joining process as mentioned in Section 3.4, refinement operations can lead to a degradation of the degree of RTT to root satisfaction. In our implementation of the E2E-Delay Optimization we do not yet consider the E2E-Delay Optimization during cluster rebuilding processes.



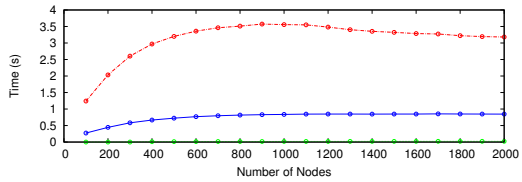
(a) Node to Root RTT Hard-QoS with 256 QoS Classes



(b) Node to Root RTT Soft-QoS with 256 QoS Classes

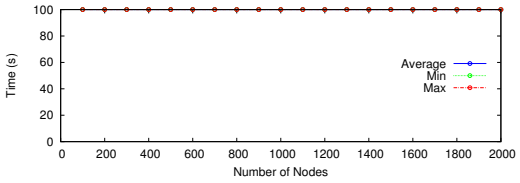


(c) Node to Root RTT Layered-QoS with 32 QoS Classes

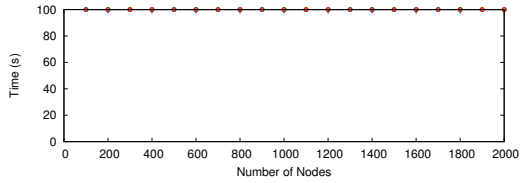


(d) Node to Root RTT Layered-QoS with 256 QoS Classes

Figure 4.15: Node to Root RTT of NICE Nodes in Different QoS Modes

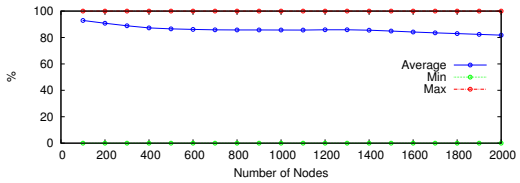


(a) Node to Root QoS Layered-QoS with 32 QoS Classes

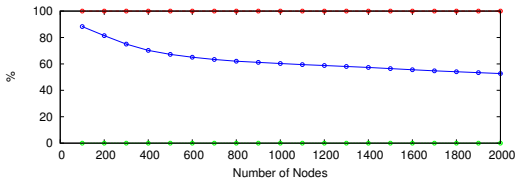


(b) Node to Root QoS Layered-QoS with 256 QoS Classes

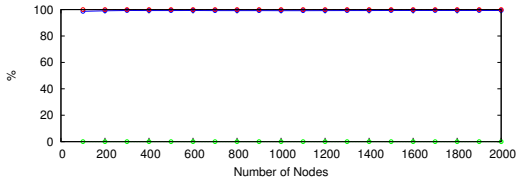
Figure 4.16: Node to Root QoS of NICE Nodes in Layered OM-QoS Framework



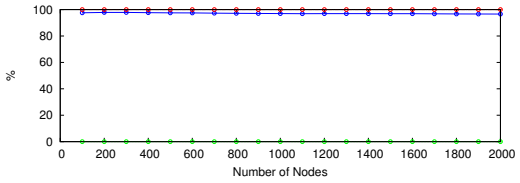
(a) RTT to Root Satisfaction with E2E-Delay Optimization off in Native NICE



(b) RTT to Root Satisfaction with E2E-Delay Optimization off in QoS-enabled NICE

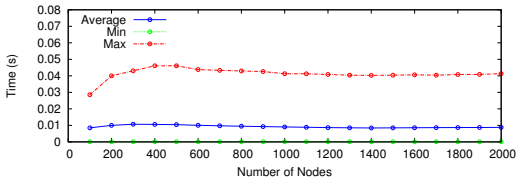


(c) RTT to Root Satisfaction with E2E-Delay Optimization on in Native NICE

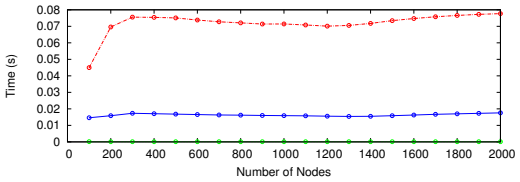


(d) RTT to Root Satisfaction with E2E-Delay Optimization on in QoS-enabled NICE

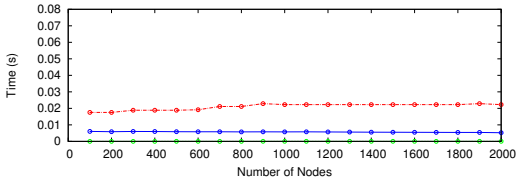
Figure 4.17: RTT to Root Satisfaction of Nodes after Initial Join



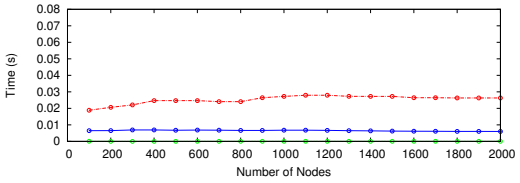
(a) RTT to Root Failed Difference with E2E-Delay Optimization off in Native NICE



(b) RTT to Root Failed Difference with E2E-Delay Optimization off in QoS-enabled NICE



(c) RTT to Root Failed Difference with E2E-Delay Optimization on in Native NICE



(d) RTT to Root Failed Difference with E2E-Delay Optimization on in QoS-enabled NICE

Figure 4.18: RTT to Root QoS Failed Difference of NICE Nodes after Initial Join

Chapter 5

Conclusion and Outlook

5.1 Conclusion

Within this Bachelor thesis we implement the NICE protocol in OMNeT++. The native NICE protocol uses RTT optimizations to build a NICE topology. We modified the native NICE protocol to support QoS. The way we achieved this is by implementing two different approaches. The first one is the protocol-dDependent QoS approach, which modifies the cluster leader selection of the NICE protocol. Instead of considering the RTT, to each node a QoS class is assigned. Based on the QoS class the cluster leaders are determined. In this approach the cluster mate with the highest QoS class is determined to be cluster leader. We divided this approach in a static and a dynamic part. For the static part we use the term hard-QoS and for the dynamic part the term soft-QoS. The soft-QoS mode simulates sudden QoS requirement violations, to which the NICE protocol has to react. A more general approach to support QoS is the Layered OM-QoS Framework. This framework does not need any protocol modification. That is why we call it protocol-independent QoS approach. In our scenario this approach uses dedicated NICE networks in dedicated layers. In each dedicated network, nodes of only one single QoS class are present. The dedicated NICE networks are interconnected via the root nodes, which are also called gateway nodes and build a chain.

We also implemented the End-to-End Delay Optimization as an additional mode to the NICE protocol, which tries to optimize the paths from any node to the root node during the joining process. The goal here is to guarantee certain RTT constraints from the root node to any node in the NICE network.

For the native NICE mode, our results show the impact of the cluster constant k, which is one of the central parameters for the NICE protocol. Many measured values such as the number of cluster mates, the hop count, the fan out, the join duration and leave duration depend on this value. Since a small cluster constant leads to smaller clusters, the chance to build more layers increases. Moreover, more layers means that the hop count is also getting higher. On the other side, the higher the cluster constant is, the higher the fan out is which indicates the load on the cluster leaders, which have to forward the multicast traffic.

The join and leave duration depends on the number of layers. Since a smaller cluster constant potentially leads to more clusters, and therefore to more layers, the join and leave duration are depending on the cluster constant as well.

A value, that is not influenced by the cluster constant, is the next hop RTT. This value shows the quality of the cluster building process in terms of RTT. The cluster building process depends only on the RTT between the nodes and not on the number of cluster mates, which is directly linked to the cluster constant. The next hop RTT measurements show that the RTT from the cluster mates to their cluster leader is lower than the average RTT between two nodes.

The first approach to support QoS for NICE called protocol-dependent QoS approach has two modes: The hard- and the soft-QoS mode. The results show that both modes fulfill the requirements of the paths to support QoS. We can also conclude that the average RTT between nodes and the root node is not bad compared to the RTT optimized native NICE.

The second approach uses the Layered OM-QoS Framework, which is a protocol-independent approach. The advantage of this approach is that the fan out is reduced dramatically, which means that the load on the multicast data forwarders is reduced. An important parameter in this approach is the number of QoS classes used, since this number is equal to the number of dedicated NICE networks. The more dedicated networks there are, the less nodes are in one single NICE network. This affects the cluster size, and therefore reduces the fan out, the join- and leave duration. The disadvantage of the Layered OM-QoS Framework is that the hop count increases dramatically, and due to this fact, the RTT between the node and the root node increases as well. Nevertheless, like the hard- and the soft-QoS mode, the Layered OM-QoS Framework is able to fulfill the requirements to support QoS along 100% of the paths from the leaf nodes to the root node.

In the last part, we evaluated the implementation of the E2E-Delay Optimization when nodes are joining. The RTT range from any node to the root node using our distance matrices is optimal in the interval of 25ms to 50ms. Our results show that we can guarantee a RTT to Root of 25ms to 50ms for almost every joining node. Even for those nodes for which the RTT constraints could not be fulfilled, the difference between the achieved RTT to root and the RTT to root constraint value is lower than having the E2E-Delay Optimization turned off. The results using the E2E-Delay Optimization on top of the native NICE implementation are slightly better in terms of RTT to root constraint satisfaction than using the hard-QoS approach.

As a conclusion, we can say that we successfully implemented, improved and extended the NICE protocol with our QoS approaches and the E2E-Delay Optimization when joining a NICE network. Our results show that NICE can be used to efficiently distribute multicast data in terms of low RTT constraints from nodes to the root node, which is the origin of the multicast data traffic. The evaluation of our developed modifications clarify that NICE can be used to support QoS along all the paths from the root node to any node. The abstraction of the QoS classes is a valid model for one or many QoS parameters like bandwith, computation time, uptime, etc., which are used in real-world environments. Therefore, the NICE protocol with our modifications and extensions is well suited for real-time multicast applications over network infrastructures such as the Internet, which have only limited multicast and QoS support by themself.

5.2 Outlook

The first point concerns the ungraceful leave of nodes. Currently, in our implementation nodes only invoke graceful leaves. The announcement of a leave of a node, especially of a node with a responsibility for other nodes, is required in our implementation to improve the network topology stability. In case of an ungraceful leave, such announcements would be absent. A possible method to allow ungraceful leaves would be to extend the Timeout Message reactions. In our implementation, the lack of announcements using Timeout Messages is problematic, since we need to precisely keep track of the states nodes are currently in, in order to be able to react properly to a Timeout Message. The extension of the Timeout Message reactions to also support ungraceful leaves would require a mechanism to check if a node really left the network, since if only one node reports that a certain node has left the network, it might be not trustworthy. But basically, our approach would also support ungraceful leaves up to a certain degree.

The second point looks at the implementation of a dynamic adaptation of the QoS class assigned to a node during the simulation. Such a mechanism would allow to simulate that a node adapts its QoS requirements during the use of a service provided by the network. This would require an appropriate reaction of the NICE protocol using the QoS approaches. The modifications to be made to the QoS approaches would include a rejoin mechanism, which allows a node to completely leave and rejoin the NICE network. In our implementation we did not consider a full rejoin to the NICE network.

Another point is the further extension and improvement of the E2E-Delay Optimization. The extension could contain the E2E-Delay Optimization being active all over the simulation time. This is beyond the scope of this Bachelor thesis, since the implementation of the E2E-Delay Optimization over the whole simulation time would require a full rejoin mechanism. Since for the E2E-Delay Optimization, information about the whole path is needed, a complete rejoin of the nodes may become necessary to meet the RTT constraints. Such a rejoin complicates the refinement process.

All these modifications, which were discussed in the outlook would bring the NICE protocol implementation even more close to a real-world implementation fully supporting all different QoS requirements during full life time of a node.

Bibliography

[1] 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, October 2002, pp. 205–217. [Online]. Available: http://portal.acm.org/citation.cfm?id=633045

[2] Wikipedia, "Peer-to-peer — Wikipedia, the free Encyklopaedia," http://en.wikipedia.org/wiki/Peer-to-peer, 2009, [Online; accessed 05-August-2008].

[3] 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, 2005. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1528337

[4] 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*, vol. 31, no. 4. ACM Press, October 2001, pp. 161–172. [Online]. Available: http://portal.acm.org/citation.cfm?id=383072

[5] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, F. M. 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, February 2003. [Online]. Available: http://portal.acm.org/citation.cfm?id=638336

[6] G. C. Plaxton, R. Rajaraman, and A. W. Richa, "Accessing nearby copies of replicated objects in a distributed environment," in *ACM Symposium on Parallel Algorithms and Architectures*, 1997, pp. 311–320. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.1850

[7] Wikipedia, "Flooding algorithm — Wikipedia, the free Encyklopaedia," http://en.wikipedia.org/wiki/Flooding_algorithm, 2008, [Online; accessed 05-August-2009].

[8] L. Strigeus and B. Inc., "μtorrent — the lightweight and efficient bittorrent client," http://www.utorrent.com/, 2009, [Online; accessed 05-August-2009].

[9] 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, 2007. [Online]. Available: http://dx.doi.org/10.1109/COMST.2007.4317616

[10] D. Milic, M. Brogle, and T. Braun, "Video broadcasting using overlay multicast," in *ISM '05: Proceedings of the Seventh IEEE International Symposium on Multimedia*. Washington, DC, USA: IEEE Computer Society, December 12–14 2005, pp. 515–522.

[11] Wikipedia, "Graph center — Wikipedia, the free Encyklopaedia," http://en.wikipedia.org/wiki/Graph_center, 2009, [Online; accessed 05-August-2009].

[12] 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, 2008, pp. 847–852.

[13] O. Community, "What is omnet++?" http://www.omnetpp.org/home/what-is-omnet, 2009, [Online; accessed 05-August-2009].

[14] A. Varga, "Omnet++ — manual (scalars)," http://www.omnetpp.org/doc/omnetpp33/manual/usman.html#sec358, [Online; accessed 05-August-2009].

[15] ——, "Omnet++ — manual (plove)," http://www.omnetpp.org/doc/omnetpp33/manual/usman.html#sec353, [Online; accessed 05-August-2009].

[16] L. Bettosini, "Quality of service for multicasting in content addressable networks," Master's thesis, University of Bern, Switzerland, 2009.

[17] 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, October 26 – 27 2009.

[18] A. Medina, A. Lakhina, I. Matta, and J. Byers, "Brite: an approach to universal topology generation," in *9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001*, August 2001, pp. 346–353, http://www.cs.bu.edu/brite/ [Online; accessed 7-July-2009]. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=948886

[19] A. Varga, "Omnet++ api reference 3.2," http://www.omnetpp.org/doc/api/classcMessage.html, 2009, [Online; accessed 05-August-2009].

[20] W. S. Voon, "Permutations in c++," http://www.codeguru.com/cpp/cpp/algorithms/article.php/c5123, 2006, [Online; accessed 05-August-2009].

[21] A. Rüttimann, "Quality of service, end to end delays and overlay multicast for structured p2p networks like chord," Master's thesis, University of Bern, Switzerland, 2009.

[22] A. Varga, "Omnet++ — manual (rng)," http://www.omnetpp.org/doc/omnetpp33/manual/usman.html#sec236, [Online; accessed 05-August-2009].

[23] D. A. Tran, K. A. Hua, and T. Do, "Zigzag: an efficient peer-to-peer scheme for media streaming," in *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*, vol. 2, March 30 – April 3 2003, pp. 1283–1292. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1208964