University of Twente
Enschede - The Netherlands

Master of Science Thesis
University of Twente
Design and Analysis of Communication Systems

Using NSIS (Next Steps in Signaling) for support of QoS aware multimedia services

Ruud Klaver

Februari 9, 2007

Committee:
Dr. ir. Georgios Karagiannis (UT/DACS)
Dr. ir. Pieter-Tjerk de Boer (UT/DACS)
Dr. ir. Geert Heijenk (UT/DACS)
Dr. Attila Bader (Ericsson)

List of Abbreviations

API Application Programming Interface

DACS Design and Analysis of Communication Systems

DCCP Datagram Congestion Control Protocol

DiffServ Differentiated Services

DoS Denial of Service

DSCP DiffServ Code Point

ECN Explicit Congestion Notification

EF Expedited Forwarding

FIFO First In First Out

GHC GIST Hop Count

GIST General Internet Signaling Transport

HTB Hierarchical Token Bucket

ICMP Internet Control Message Protocol

IETF Internet Engineering Task Force

IntServ Integrated Services

IP Internet Protocol

ISCL IntServ Controlled Load

LAN Local Area Network

MA Messaging Association

MRI Message Routing Information

MRM Message Routing Method

MTU Maximum Transmission Unit

NAT Network Address Translation

NLI Network Layer Information

NSIS Next Steps In Signaling

NSLP NSIS Signaling Layer Protocol

NTLP NSIS Transport Layer Protocol

OO Object Oriented

OS Operating System

PHB Per Hop Behaviour

qdisc Queueing Discipline

QNE QoS NSIS Entity

QNI QoS NSIS Initiator

QNR QoS NSIS Receiver

QoSM QoS Model

QoS Quality of Service

QSPEC QoS Specification

RAO Router Alert Options

RFC Request For Comment

RII Request Identification Information

RMD Resource Management in DiffServ

RMF Resource Management Function

RSN Reservation Sequence Number

RSVP Resource ReSerVation Protocol

RTT Round Trip Time

SCTP Stream Control Transmission Protocol

SFQ Stochastic Fairness Queueing

SID Session Identifier

SII Source Identification Information

SLA Service Level Agreement

TCP Transmission Control Protocol

TLS Transport Layer Security

TLV Type Length Value

TTL Time To Live

UDP User Datagram Protocol

UML User-Mode Linux

UT University of Twente

VM Virtual Machine

VoIP Voice over IP

Contents

1. Introduction **8**
1.1. Goals . . . 9
1.2. Implementation Requirements . . . 9
1.3. Thesis Structure . . . 10

2. NSIS Overview **11**
2.1. NSIS Introduction . . . 11
2.2. NSIS Components . . . 11
2.2.1. NSIS Transport Layer Protocol . . . 12
2.2.1.1. GIST API . . . 13
2.2.1.2. Message Routing Information . . . 16
2.2.1.3. GIST Service Example . . . 17
2.2.1.4. GIST Transmission Modes . . . 18
2.2.1.5. GIST Messages . . . 19
2.2.1.6. GIST Message Exchange Example . . . 22
2.2.1.7. GIST Advanced Features . . . 24
2.2.2. QoS NSIS Signaling Layer Protocol . . . 25
2.2.2.1. QoS-NSLP Components . . . 25
2.2.2.2. QoS-NSLP Message Processing . . . 25
2.2.2.3. QoS-NSLP Example . . . 27
2.2.2.4. QoS-NSLP Layering . . . 29
2.2.3. QoS-NSLP QoS Models . . . 30
2.2.3.1. IntServ Controlled Load QoS Model . . . 30
2.2.3.2. Resource Management in Diffserv QoS Model . . . 31

3. GIST Design and Implementation **36**
3.1. Previous Work . . . 36
3.2. Development Environment . . . 38
3.3. High-level Program Structure Overview . . . 39
3.3.1. Threading Model . . . 40
3.4. Network Interaction . . . 41
3.4.1. Query Mode . . . 41
3.4.2. Datagram Mode . . . 42
3.4.3. Connection Mode . . . 42
3.5. Message Processing . . . 42
3.5.1. Message Objects . . . 43

3.5.2. Message Object Interface . . . 44
3.5.3. Exceptions . . . 45
3.6. State Machines . . . 45
3.6.1. Routing State . . . 46
3.6.1.1. Query Node State Machine . . . 46
3.6.1.2. Responder Node State Machine . . . 47
3.6.2. Message Association State Machine . . . 48
3.7. Program Flow . . . 49
3.7.1. Example Program Flow . . . 50
3.8. Miscellaneous Features . . . 51
3.8.1. Logging . . . 51
3.8.2. ICMP Monitoring . . . 52
3.8.3. Source Identification Information Handle . . . 52
3.8.4. Network Interface Management . . . 53

4. QoS-NSLP Design and Implementation 59
4.1. Previous Work . . . 59
4.2. QoS-NSLP and IntServ Controlled Load QoS Model Implementation . . . 60
4.2.1. Program Structure . . . 61
4.2.2. Message and Exception Processing . . . 63
4.2.3. Application API . . . 63
4.2.3.1. VLC Application API Implementation . . . 66
4.2.4. Linux Traffic Control Subsystem . . . 67
4.3. RMD Linux Traffic Control . . . 68
4.3.1. The REMARKFIFO Queueing Discipline . . . 68
4.3.2. Linux Traffic Control Subsystem . . . 71

5. Functional Experiments 75
5.1. User-Mode Linux Setup . . . 75
5.2. Early Tests . . . 75
5.3. Lab Testing . . . 77
5.3.1. Successful Reservation and Teardown . . . 78
5.3.2. Successful Reservation With NSIS-Unaware Nodes . . . 78
5.3.3. Unsuccessful Reservation . . . 79
5.3.4. Final Demonstration . . . 80
5.4. Interoperability Testing . . . 82

6. Conclusion and Future Work 84
6.1. Conclusions and evaluation . . . 84
6.2. Future Work . . . 85

A. GIST and QoS-NSLP Logging Example 89

Next Steps In Signaling (NSIS) is a newly designed protocol suite for flow signalling on IP networks. One of its intended applications is to provide a dynamic end-to-end QoS reservation protocol. The main goal of the assignment described in this thesis is to update and expand and existing NSIS implementation for use by Quality of Service (QoS) aware multimedia applications. In doing this, it serves in part as a verification of the applicability and implementability of these new protocols.

1. Introduction

Transporting packet based network traffic, such as Internet Protocol [35], has traditionally been a best-effort service. This means that each Internet Protocol (IP) packet receives the same treatment from routers and that any router it passes will process and transmit the packet as fast as possible, depending on the load on the router. However, in a world of IP convergence, demand has risen for different levels of services to be applied to different packets, so-called Quality of Service. In particular real-time streaming traffic such as multimedia services place stricter requirements on such parameters as packet loss and jitter. Over the years a number of solutions have emerged to implement QoS in IP networks in various ways, e.g. Integrated Services (IntServ) [21] and Differentiated Services (DiffServ) [20]. Each QoS solution defines its own set of service types and means of attributing these different service levels to different IP packets. Most of these QoS systems are based on static reservations within the own domain, used for example for Service Level Agreement (SLA)s. Several protocols, such as Resource ReSerVation Protocol (RSVP) [22], were developed to provide dynamic soft-state end-to-end reservations, i.e. reservations for a flow from a sender to a receiver that may traverse several QoS domains and will expire after a certain period of time if not refreshed. This flow could be for example a Voice over IP (VoIP) session or a on-demand video stream. A conceptual depiction of this can be seen in Figure 1.1.

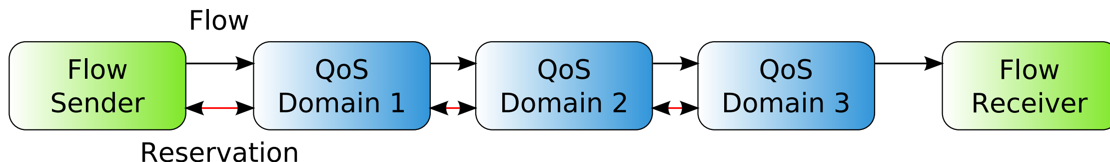

Figure 1.1.: Diagram of a dynamic reservation for a flow traversing several QoS network domains.

However, these reservation signalling protocols suffer from a number of issues and limitations [32]. For this reason, the Internet Engineering Task Force (IETF) is proposing a suite of protocols under the name of Next Steps In Signaling [28]. The goal of NSIS even goes beyond providing QoS signalling, it defines a generalised end-to-end signalling protocol for flows and several signalling applications, one of them being QoS. Other signalling applications, such as metering and Network Address Translation (NAT) traversal are also to be defined. The QoS signalling application in particular is designed in such a way that it is flexible in the types of QoS it can support, not restricting itself to any particular QoS model. In this way, NSIS should be ideally suited to perform dynamic end-to-end soft-state reservations for multimedia flows.

1.1. Goals

The main goal of this assignment is to optimise and expand an existing implementation of NSIS, developed at the Design and Analysis of Communication Systems (DACS) group [42, 44], in such a a manner that multimedia applications may use it to perform dynamic QoS reservations. One of the adaptations that need to be made towards this end is the development of a Application Programming Interface (API) that applications can use to request QoS from the NSIS implementation.

In particular the objectives for the assignment are:

- Studying of the current specifications and implementation of the NSIS protocol suite and in particular the QoS-NSLP protocol.
- Optimising and expanding an existing Linux prototype implementation of NSIS.
- Designing and implementing a QoS application API that can be used between NSIS and applications providing real-time streaming multimedia
- Setting up a demonstration of a QoS aware multimedia streaming application using NSIS.

As NSIS is designed to perform exactly the task set to it here, i.e. allowing dynamic QoS reservations for real-time streaming multimedia applications, and it is comprised of a set of protocols specifications that are at the time of writing still under development, this thesis is in part a proof-of-concept of the applicability and implementability of those parts of NSIS that are used.

1.2. Implementation Requirements

To achieve the goals set in the previous section, said implementation should adhere to a number of requirements:

- To test the NSIS specifications thoroughly, the implementation should follow these specification as strictly as possible. When the protocol specifications allow different choices or optional features, those options most logical to the achievement of the goals should be chosen.
- Because of the nature of this assignment the design and implementation should be a that of an investigative prototype. What this means is that it should not be designed to operate in a production environment, rather be aimed as much as possible at studying the inner workings of the programs and the protocols they implement. A prototype implementation should for example, as opposed to a production implementation, focus less on efficiency and more on disclosure of details.
- The design and implementation should be structured in such a way that future changes and extensions to the protocol and even unimplemented functionality can be easily incorporated at a later time. As the NSIS protocol specifications are still

drafts, at least at the time of writing, updates and additions to these protocols are released over time, prompted by feedback and discussion. If the implementation is to be used in future work, a modular program structure should aid in this.

- The implementation should be both portable and configurable. What this means is that, although a single Operating System (OS) with additional software dependencies may be chosen to develop the implementation for, it should be able to run easily on other systems using the same OS and software packages. As this may require alteration of configuration parameters, the implementation should allow and facilitate this.
- Error situations should be handled gracefully. The specifications foresee a lot of different error conditions which must be dealt with appropriately and should still allow the implementation to continue functioning. Even those error conditions that are unforeseen, such as programming errors, should be handled in such a way that it is clear that something went wrong.

1.3. Thesis Structure

The rest of this thesis is structured in the following manner; chapter 2 will provide an overview of the protocols in the NSIS protocol suite and their details relevant to this thesis. Chapter 3 will describe the General Internet Signaling Transport (GIST) implementation, which provides the lower layer functionality of NSIS. Chapter 4 will describe the QoS-NSIS Signaling Layer Protocol (NSLP) implementation, the upper layer QoS functionality of NSIS. Chapter 5 will provide a description of the experiments performed to evaluate the functionality of the implementations. Finally, chapter 6 will contain the conclusions of this thesis and evaluate if the goals set in section 1.1 have been reached.

2. NSIS Overview

This chapter will attempt to provide a basic understanding of the concepts and components of NSIS. By no means should this be considered to be a complete and exhaustive description of the subject matter at hand, rather an informative summary that will allow the reader to understand further chapters of this report. For further details or design motivations readers are advised to consult the respective normative documents referenced.

At the time of writing, most of the NSIS specifications were still in the draft stage and so had not yet reached the status of final document. The draft versions of the specifications on which the descriptions and work in this thesis are based are indicated in the respective entries in the references section.

2.1. NSIS Introduction

The abstract of the document describing the NSIS framework [28] states the following:

> The Next Steps in Signaling (NSIS) working group is considering protocols for signaling information about a data flow along its path in the network. The NSIS suite of protocols is envisioned to support various signaling applications that need to install and/or manipulate such state in the network.

This accurately describes the goal of the NSIS suite as a whole. Alternatively, it can be said to be a collection of protocols that enable end-to-end signalling pertaining to a flow or a collection of flows across heterogeneous IP based networks. In practice this means that any IP-based router within the path of a flow can communicate with other routers about this flow, and possibly install state about it. The fact that it is end-to-end also means that NSIS is meant to operate across different network domains, each with their own characterstics such as QoS provisioning, owned by different providers.

The nature of the signalling in this definition is intentionally unspecified. This is because NSIS, owing to its modular concept, decouples the signalling application from the signalling service. Examples of envisioned signalling applications, some of which are already defined, are QoS provisioning [33], NAT and firewall traversal and simple metering. Particularly the QoS application is considered to be an important part of NSIS and is the main application featured in this report.

2.2. NSIS Components

As already mentioned, NSIS is conceptually divided into two layers. The lower layer is known as the NSIS Transport Layer Protocol (NTLP), the upper as the NSLP. As

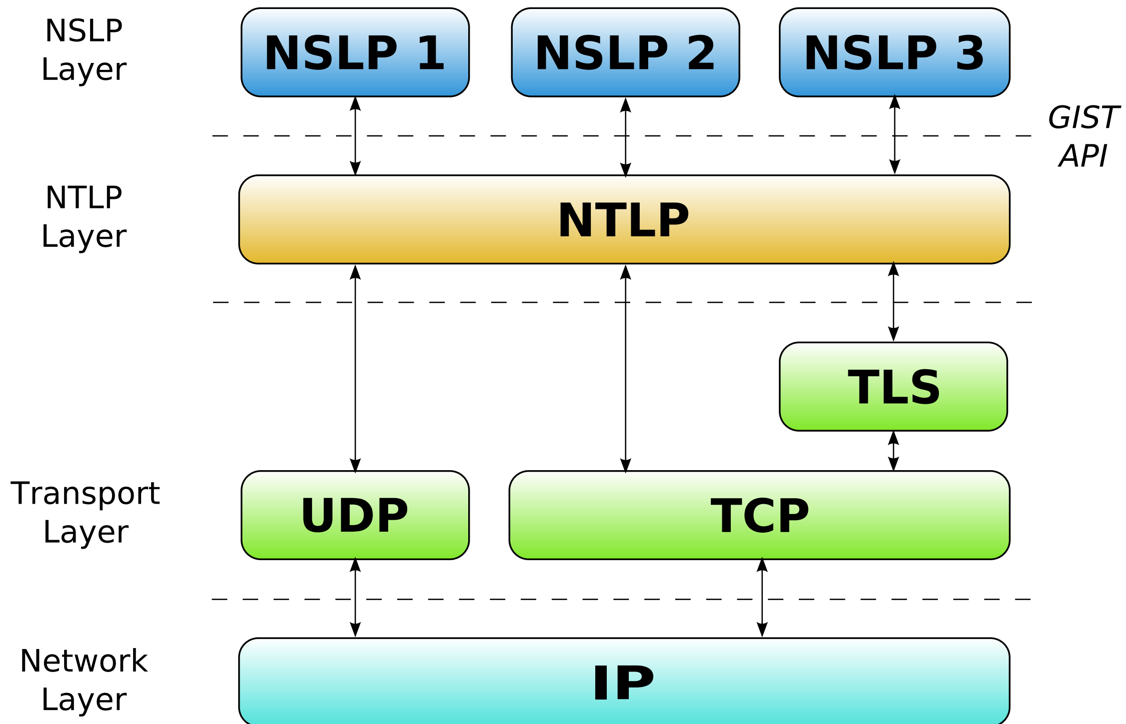

Figure 2.1.: Conceptual NSIS protocol stack.

the name would suggest, the NTLP provides a transport service for sending messages up and down a communication path, allowing routers within this path to communicate about flows. A NSIS node can have one or more NSLP applications running, each designated their own NSLP-ID. Any of these applications can make use of the NTLP service, which operates independently from the contents and nature of the NSLP data it carries. The relationship between these protocols, and their relationships with other well known network protocols, is illustrated in the protocol stack in Figure 2.1. As can be seen, the NTLP can use either User Datagram Protocol (UDP) [34] or Transmission Control Protocol (TCP) [37] for its message transport, the latter optionally making use of Transport Layer Security (TLS) [26] for security. The NTLP is not limited to the use of these protocols and can also use others, e.g. Stream Control Transmission Protocol (SCTP) [41] or Datagram Congestion Control Protocol (DCCP) [31], but because these are the only protocols used in the implementation described in this report they are the only ones pictured. Communication between the NSLP applications and the NTLP is performed through a predefined API.

2.2.1. NSIS Transport Layer Protocol

The NSIS working group specifies GIST [39] as the protocol that operates as NTLP. To understand what type of service GIST provides, the API specified within GIST will be described first, which allows NSLP applications to use its services. After this the inner

workings of GIST can be illustrated.

2.2.1.1. GIST API

GIST defines its API in terms of abstract service primitives with parameters. This means that no byte level specification is given and that, although the concept of the API remains the same, its exact implementation is undefined.

The GIST API consists of six service primitives, three from the NSLP application towards GIST, three in the opposite direction. The service primitives directed from the signalling application towards GIST are:

- SendMessage
- SetStateLifetime
- InvalidateRoutingState

The service primitives travelling in the opposite direction, from GIST to the NSLP application, are:

- RecvMessage
- MessageStates
- NetworkNotification

SendMessage
The SendMessage service primitive allows the NSLP application to send messages to its upstream or downstream peer, depending on the Message Routing Information (MRI). Its parameters are the following:

> SendMessage (NSLP-Data, NSLP-Data-Size, NSLP-Message-Handle, NSLPID, Session-ID, MRI, SII-Handle, Transfer-Attributes, Timeout, IP-TTL, GIST-Hop-Count)

NSLP-Data is the payload the NSLP application wants to transmit.

NSLP-Data-Size is the length of the NSLP-Data.

NSLP-Message-Handle is a handle that refers to this particular message. It allows GIST to refer back to it when it issues the MessageStatus service primitive.

NSLPID is 1a 6-bit unsigned integer identifying the NSLP application.

Session-ID is a 16-byte identifier unique to this session.

MRI is the Message Routing Information, which describes the flow to which the signalling pertains.

The following arguments are optional:

SII-Handle is the Source Identification Information Handle, which can be used to bypass state stored in GIST and directly address a node.

Transfer-Attributes allows the application to convey desired transfer properties of the message. Among others this can contain wether or not the message should be sent reliably and wether or not is should be sent securely.

Timeout is the time for which GIST should attempt to keep sending the message.

IP-TTL is the Time To Live (TTL) value GIST should put in the IP header.

GIST-Hop-Count is the initial value for the GIST Hop Count (GHC) when GIST transmits the message.

SetStateLifetime
The SetStateLifetime service primitive allows the NSLP application to control for how long the state retained within GIST for a particular session is valid. Its parameters are the following:

SetStateLifetime (NSLPID, MRI, State-Lifetime)

NSLPID is a 16-bit unsigned integer identifying the NSLP application.

MRI is the Message Routing Information, which describes the flow to which the signalling pertains.

State-Liftetime is the amount of time for which the application wishes the state to remain active in GIST.

InvalidateRoutingState
The InvalidateRoutingState service primitive allows the NSLP application to explicitly request GIST to remove any state associated with a particular session. Its parameters are the following:

InvalidateRoutingState (NSLPID, MRI, Status, Urgent)

NSLPID is a 16-bit unsigned integer identifying the NSLP application.

MRI is the Message Routing Information, which describes the flow to which the signalling pertains.

Status is a boolean which indicates how definite the routing state invalidation should be.

Urgent is a boolean which indicates wether state recovery should proceed immediately.

RecvMessage
The RecvMessage service primitive allows GIST to deliver incoming messages to a NSLP application. Its parameters are the following:

> RecvMessage (NSLP-Data, NSLP-Data-Size, NSLPID, Session-ID, MRI, Routing-State-Check, SII-Handle, Transfer-Attributes, IP-TTL, IP-Distance, GIST-Hop-Count, Inbound-Interface)

NSLP-Data is the NSLP payload of the message.

NSLP-Data-Size is the length of the NSLP-Data.

NSLPID is 1a 6-bit unsigned integer identifying the NSLP application.

Session-ID is a 16-byte identifier unique to this session.

MRI is the Message Routing Information, which describes the flow to which the signalling pertains.

Routing-State-Check is a boolean indicating that GIST is asking the NSLP application wether or not to set up state with this peer. If it is set, the application should reply to this primitive with:

- A boolean of wether it wants to set up state or if it wants the query to be propagated further downstream.
- Optionally a payload that the application wants GIST to include in the response to the querying peer or in the propagating query in case it does not want to setup state.

SII-Handle is the Source Identification Information Handle of the transmitting node, which can be used to bypass state stored in GIST and directly address this node.

Transfer-Attributes are the transfer properties with which the message was transmitted, such as reliability and security.

IP-TTL is the TTL value of the IP header of the received message.

IP-Distance is the calculated distance in IP hops between this node and the sender of the message.

GIST-Hop-Count is the value of the GHC in the received GIST message.

Inbound-Interface provides information about the physical interface on which the message was received.

MessageStatus
The MessageStatus primitive allows GIST to indicate to a NSLP application if a message was sent correctly and, if so, what transfer properties were used. Its parameters are the following:

MessageStatus (NSLP-Message-Handle, Transfer-Attributes, Error-Type)

NSLP-Message-Handle is a reference to the message that was generated earlier by the application and set using the SendMessage service primitive.

Transfer-Attributes are the transfer properties with which the message was transmitted, such as reliability and security.

Error-Type indicates, if the message could not be delivered, the reason of failure, the most important one being that this node is the last NSIS-aware node in the path.

NetworkNotification

The MessageStatus primitive allows GIST to indicate to a NSLP application any changes in network status. Its parameters are the following:

NetworkNotification (NSLPID, MRI, Network-Notification-Type)

NSLPID is 1a 6-bit unsigned integer identifying the NSLP application.

MRI is the Message Routing Information, which describes the flow to which the signalling pertains.

Network-Notification-Type indicates the type of network status change, such as a change in routing state.

2.2.1.2. Message Routing Information

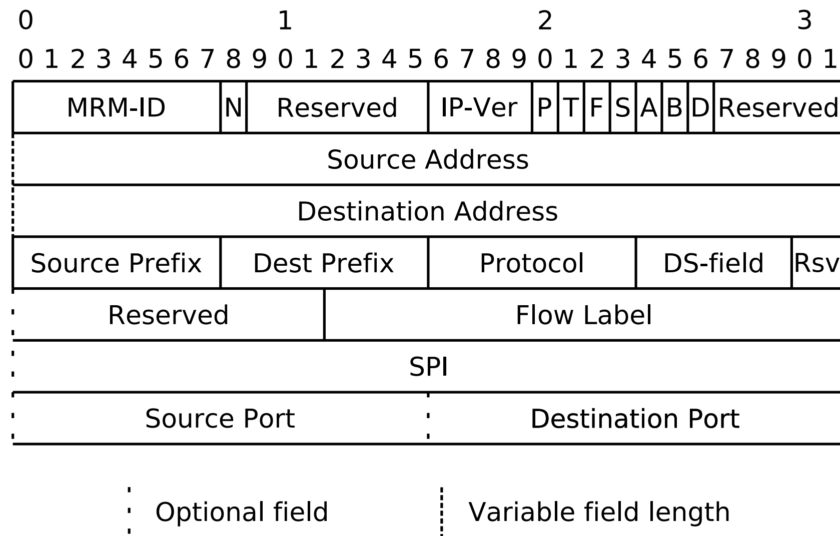

Figure 2.2.: MRI with Path-coupled MRM.

A central concept to GIST and its API is the MRI, which is used to describe a flow or a set of flows. A flow can be described in different ways by the so-called Message Routing Method (MRM), the default of which is the path-coupled MRM. The bitformat of the path-coupled MRM is illustrated in Figure 2.2. As can be seen the path-coupled MRM is based on the traditional 5-tuple of protocol, source address, source port, destination address and source port with some additional characteristics. The use of these characteristics can be controlled using the flags at the start of the MRI. The NSLP applications use the MRI to inform GIST about the characteristics of the flow. Another important flag is the D flag, which indicates the direction of the signalling in respect to the flow. If it is set to 0 the direction of the signalling is in the same direction of the flow, if set to 1 it is the opposite. With this a NSLP application can indicate if it wants to send a message to its downstream peer, i.e. in the same direction as the flow or towards the flow receiver, or to its upstream peer, i.e. in the opposite direction of the flow or towards the flow sender.

2.2.1.3. GIST Service Example

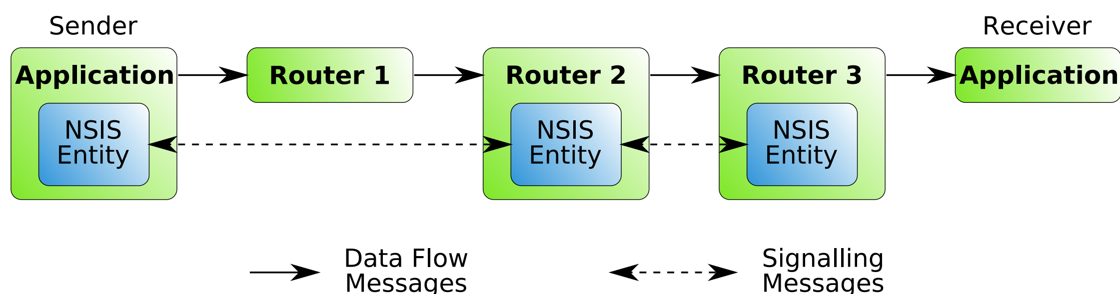

Figure 2.3.: Signalling and data flows.

We can now construct a complete picture of the service that GIST provides to the NSLP applications. These applications use the SendMessage service primitive to send messages about a flow described in the MRI to its peers. A typical example of this is illustrated in Figure 2.3 with a flow going from its sender to the receiver. Note that, for GIST, flows are always unidirectional. Any bidirectional operation requires coupling of two unidirectional flows at the NSLP level. In this example not every node on the path is NSIS-aware, i.e. has GIST software and one or more NSLP applications running. Say that the flow sender wants to do some signalling about the flow, e.g. a QoS reservation. Its QoS-NSLP application will construct a MRI about the flow, create a new Session Identifier (SID) and issue a SendMessage service primitive to GIST, including the QoS-NSLP payload to be sent and indicating what transfer properties the message should have. This can be for example that the message should be sent reliably but need not be sent securely. GIST will then check its internal state to see if knows the downstream peer, keyed by the combination of the NSLP-ID, MRI and SID. If this state exists it will use it to find out if this node has a Messaging Association (MA) with the downstream peer that has the same properties that the application requested. If the state does not

exist it will attempt to discover the downstream peer. Peer discovery is performed with a UDP datagram with a preset destination port addressed towards the flow destination, gleaned from the MRI, with the Router Alert Options (RAO) [30] in the IP header set to a value corresponding to the NSLP-ID. NSIS-aware routers along the path to the flow destination actively listen out for any UDP datagrams with the preset port and the RAO set and intercept these datagrams. In this way the next downstream peer can be found and a routing state between the peers can be set up, in our example between the flow sender and router 2. What follows is a negotiation between the two nodes to either re-use an existing MA or set up a new MA with the desired properties, e.g. a TCP connection. Once this MA is found or set up, the NSLP message is transferred over it and received by router 1. Its GIST will issue a RecvMessage service primitive to the QoS-NSLP application. This application will probably decide to issue a message further downstream to complete the reservation along the path, again performing downstream peer discovery.

This process will continue until the last node on the path is found, in this case router 3. Note that, although a node may have GIST software running, it may not have the NSLP application identified by the NSLP-ID running, or the application may even choose not the participate using the response to the RecvMessage service primitive. Now if for example router 3 wants to issue a message upstream, this could be a message containing information about the success or failure of the QoS reservation, it will do this with the D flag of the MRI set to 1. Now GIST in router 3 still has state stored about its upstream peer, since that peer has just performed downstream discovery with it. It can use this to send the message upstream.

All states stored in GIST are soft. This means that after a certain period of disuse states will expire automatically.

2.2.1.4. GIST Transmission Modes

GIST has three different modes to transfer messages to its peers, each one of which will now be illustrated.

Query Mode

This mode has already been mentioned in section 2.2.1.3 and is used to discover downstream peers. Although GIST specifies that it can also be used to discover upstream peers, this is not advised. This means that downstream messages should usually precede upstream messages, so that state can be installed during downstream peer discovery.

In Query Mode a UDP datagram will be sent towards the MRI destination at a predefined destination port. The UDP source port is set to the port on which the sending GIST node will accept messages in Datagram Mode, as can be read in the relevant following section. The IP header of a message in Query Mode should include a RAO, which is a IP option that can be used to notify routers along a path and includes a 16-bit unsigned integer. This integer should be set to a value corresponding with the NSLP-ID of the message so that GIST can decide at IP level the relevant NSLP application is running and if it should intercept the message. The source IP address of

the UDP datagram can be either set to the flow source, to make sure that the message is submitted to exactly the same routing as the flow, or it can be set to the signalling source, i.e. the node sending in Query Mode, to allow interception of returned Internet Control Message Protocol (ICMP) [36] messages. The latter is important because, if the Query Mode message is not intercepted and arrives at the MRI destination and this host is not NSIS-aware, a ICMP "Port Unreachable" error is returned that can be used to determine that there are no more NSIS-aware nodes on the path.

Datagram Mode
In Datagram Mode messages are simply sent as UDP datagrams, addressed directly to the GIST node one wants to reach. The IP destination address of a node can be learnt from a message it sent in Query Mode, as in this case it should always include its IP address. Note that this may not be the same as the source IP address used for the Query Mode message. The destination UDP port should be equal to the source UDP port used in a previous Query or Datagram mode message sent from the destination node. Conversely, that means that the Datagram Mode sending node should select the source UDP port to be that port on which it is accepting Datagram Mode messages itself.

Connection Mode
In Connection Mode a Messaging Association between two nodes is used to transmit the message. A MA is a connection between two nodes using a particular connection-oriented protocol or a stack of protocols with a set of properties, such as security and reliability. In this report only TCP, for reliable connections, and TLS over TCP, for reliable and secure connections, will be used. For a particular protocol or stack of protocols GIST will maintain at most one connection between two nodes. This MA will be set up dynamically if it is not present and can be used by different sessions once present. A MA is also stored in soft state, meaning that after a certain period of disuse the connection will automatically be torn down.

2.2.1.5. GIST Messages

GIST defines four different message types, each one of which consists of a set of GIST objects. For the sake of brevity these objects will only briefly be described.

Query
As the name would suggest, this message should be used for initial peer node discovery. Once state is installed in the nodes, this message should still be sent periodically to detect changes in routing topology. A Query message can only be sent using Query Mode. The objects included in a GIST Query message are the following:

Query = Common-Header
[NAT-Traversal-Object]
Message-Routing-Information
Session-Identification
Network-Layer-Information
Query-Cookie
[Stack-Proposal Stack-Configuration-Data]
[NSLP-Data]

The Common-Header is an object that is included in all GIST messages and includes information such as the type of the message, the NSLP-ID and the GIST Hop Count. The GHC prevents messages from looping infinitely, much like the TTL at IP level. The NAT-Traversal-Object will not be discussed in this report because it was not implemented. As can be seen, the MRI and SID are included in the Query message. Together with the NSLP-ID from the Common-Header this can be used to match state within the GIST node. The Network Layer Information object contains among others a token uniquely identifying the node sending the Query message and its IP address so that other nodes can reach it in Datagram and Connection mode. The Query-Cookie is a security measure to prevent certain Denial of Service (DoS) attacks and contains random data. The Stack-Proposal and Stack-Configuration-Data contain a list of MA protocol stacks and corresponding options, such as port numbers, that the querying node can use based on the transfer attributes submitted to its API by the NSLP application. The responding node should only use this information in case it is considering reusing an existing MA. Finally, the querying node can decide to piggyback NSLP payload onto a Query message, provided that the transfer attributes of the message allow this, i.e. it may be sent unreliably and insecurely.

Response

A GIST Response message should only be sent in reply to a received Query message, so will only be used in initial peer discovery and during refreshes. If the responding node already has a MA with the querying node, the protocol stack is in the stack proposal included in the Query message, the responding node should send its response over this MA to indicate that it will reuse it. Otherwise, the Response message should be sent in Datagram Mode. The objects included in a GIST Response message are the following:

Response = Common-Header
[NAT-Traversal-Object]
Message-Routing-Information
Session-Identification
[Network-Layer-Information]
Query-Cookie
[Responder-Cookie
[Stack-Proposal Stack-Configuration-Data]]
[NSLP-Data]

The first four objects are the same as in the Query message. The Network Layer Information (NLI) should only be included if the message is sent in Datagram mode, i.e. if a MA is not being reused. The Query-Cookie object sent in the Query message should be echoed in the response. Additionally, the responding node should include its own cookie and protocol stacks when a new MA needs to be set up. The responding node should always include all protocol stacks it supports regardless of the protocol stacks included in the Query message. Again, NSLP payload can be piggybacked if the transfer attributes submitted to the API allow this.

Confirm
A GIST Confirm message can be sent in reply to a Response message. When a new MA is setup, a Confirm message must be sent as the first message on this connection, otherwise transmission of a Confirm message is controlled by a flag known as the R flag in the common header of the response message. The Confirm message must be sent in Connection mode when a MA is reused or setup and in Datagram mode otherwise. The objects included in a GIST Confirm message are the following:

```
Response = Common-Header
           Message-Routing-Information
           Session-Identification
           Network-Layer-Information
           [ Responder-Cookie
              [ Stack-Proposal
                 [ Stack-Configuration-Data ] ] ]
           [ NSLP-Data ]
```

Again, the common header, MRI, SID and NLI are included. The Responder-Cookie should always be echoed if the Response message carried that cookie. The Stack-Proposal should only be included if the message was sent in Connection mode, i.e. over a new or reused MA. The Stack-Configuration-Data in an abbreviated form should only be included if this Confirm is the first message on a new MA. Again, NSLP payload can be piggybacked. Because the Confirm message should always have the desired transfer properties, NSLP data can be carried in any case.

Data
This message is simply used to transfer NSLP data between nodes. It can be sent in any of the three transfer modes, in Connection mode over a MA, unreliably and insecurely in Datagram mode and in special cases in Query mode, allowing NSLP data to be sent downstream without installing state within the nodes. The objects included in a GIST Data message are the following:

```
Data = Common-Header
       [ NAT-Traversal-Object ]
       Message-Routing-Information
       Session-Identification
       [ Network-Layer-Information ]
       NSLP-Data
```

The MRI and SID are included for state matching. The NLI should not be included in Connection mode, otherwise it should.

Error
Error messages are generated in response to an error condition and sent back to the node that sent the message that caused the error. GIST Error messages can be sent both in Datagram and Connection mode, depending on the transfer mode of the message that caused the error. The objects included in a GIST Error message are the following:

Error = Common-Header
[NAT-Traversal-Object]
[Network-Layer-Information]
GIST-Error-Data

The NLI should only be included in Datagram mode.

MA-Hello
The MA-Hello message is just used to refresh a MA, indicating that the sending node wants to keep the MA open. A GIST MA-Hello message only consists of a common header:

MA-Hello = Common-Header

In conclusion, table 2.1 lists which messages can be sent in which modes and for which situations.

Message	Query Mode	Datagram Mode	Connection Mode
Query	Always	Never	Never
Response	Never	Unless a MA is being reused	If a MA is being reused
Confirm	Never	Unless a MA has been set up or is being reused	If a MA has been set up or is being reused
Data	If no routing state exists and locally policy allows it	If no MA exists and transfer attributes allow it	if a MA exists
Error	Never	If the message causing the error was sent in this mode	If the message causing the error was sent in this mode
MA-Hello	Never	Never	Always

Table 2.1.: Combinations of GIST messages and transmission modes.

2.2.1.6. GIST Message Exchange Example

Now that the basic components of GIST are described, their usage can be illustrated by an example of a message exchange between two nodes, as can be seen in Figure 2.4.

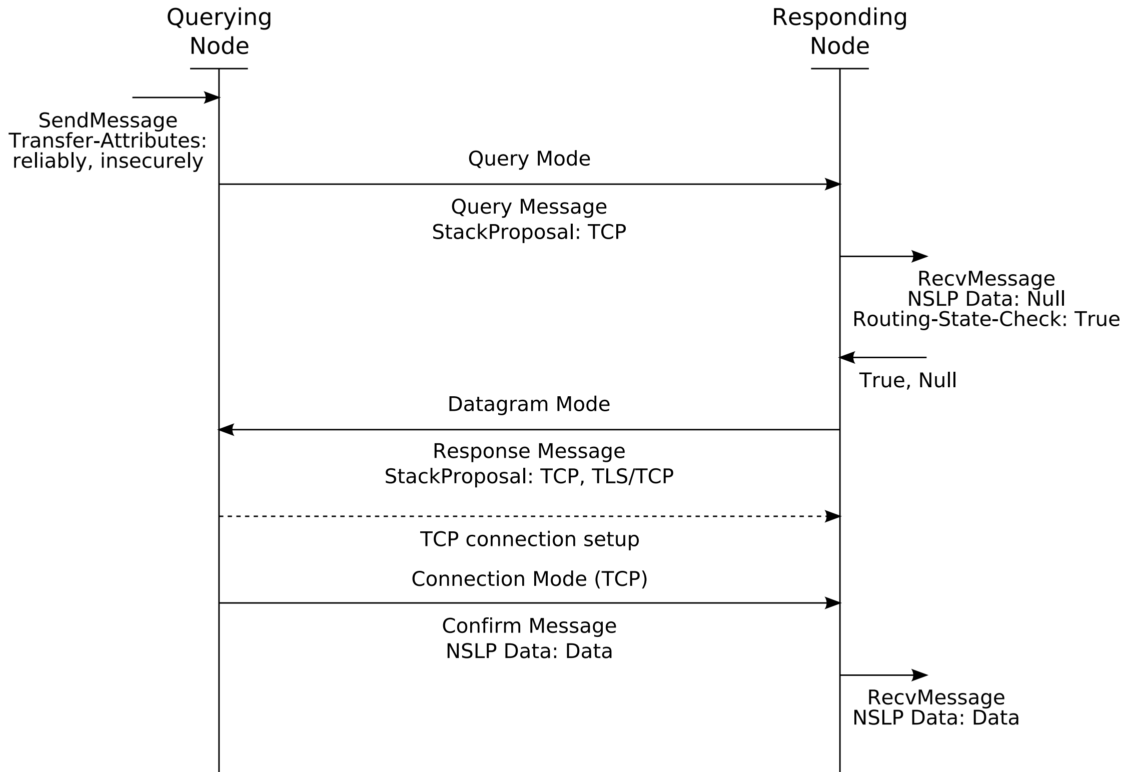

Figure 2.4.: Example of a GIST message exchange between two nodes.

The first thing that happens in the interaction between these two nodes is that a NSLP application located at the querying node wants to send some data to its downstream peer according to some MRI. This can be either an asynchronous message or a part of a chain of downstream messages. To indicate this desire, the application performs an API call to the GIST instance running at that node, including the NSLP payload, the NSLP-ID of the application, the MRI of the flow the signalling is about with the direction bit set to downstream and a SID value. Since in this case the NSLP application has not communicated downstream about this flow before, the SID is a newly generated value. This is because SID's are only of local significance between peers, i.e. not end-to-end. The application also indicates the desired transfer characteristics of the message, which can be seen in Figure 2.4. In the case of this example the application wants for the message to be sent reliably, but does not need it to be sent securely.

The GIST instance running at the querying node will process the API call and look up if there is any state installed, keyed by the NSLP-ID, MRI and SID. Because the SID is new there will be no state and GIST concludes that it will have to send a Query message. Because of the transfer attributes indicated by the NSLP application, GIST cannot include the payload in the Query message, but will include those protocols it supports that obey the desired transfer attributes in the protocol stack, in this case only TCP. The NSLP data will be queued internally. Once it has constructed the Query

message, it will send it towards the MRI destination with a RAO included in the IP header.

The first node directly downstream from the Querying node with respect to the MRI, which in this example will be the responding node, will intercept the Query message. The first thing the GIST instance at this node will do is look up if it has any state installed for this combination of NSLP-ID, MRI and SID. Once it has found that it does not, it will issue an API call to the relevant NSLP application with no NSLP data and with the Routing-State-Check parameter set to True. In this way GIST will ask the NSLP application if it wants to send up routing state. In this example the application answers that it does want to set up state and that it does not have any payload to include in the Response message. GIST then uses the NLI included in the Query message to determine the identity of the querying node and looks up if it has a MA installed to this node that uses any of the protocol stacks included in the Query message. Simply put, it checks to see if it already has a TCP connection with this node. Once it has determined that it does not, it will conclude that a new MA will need to be set up and include all protocol stacks it supports in the response message, including the port numbers on which it is listening. In this example it will include both TCP and TLS over TCP. It does this as a security measure, for details see section 8.6 of [39]. Because a MA is not being reused, the responding node will send the Response message in Datagram mode to the querying node, having learnt the destination IP address from the NLI in the Query message and the UDP destination port from the source UDP port of the Query message. GIST will also make sure to set the R flag of the common header of the Response message, as a Confirm message is required when a new MA is set up.

Once the querying node receives the Response message, it will match this with the state present within GIST and choose the MA it wants to use based on the Stack-Proposal of the Response message, in this case TCP. Based on the NLI and Stack-Configuration-Data from the Response message it will make a TCP connection to the responding node and send a Confirm message over this new MA. Included in the Confirm message is the NSLP payload that GIST had internally queued up for this routing state. The NSLP payload can then finally be delivered to the correct application at the responding node. What follows is probably that the NSLP application at the responding node processes the message and propagates it downstream, in which case the process is repeated with this node now acting as the querying node, or it will reply with an upstream message, in which case the routing state and MA now installed at the responding node will be reused.

2.2.1.7. GIST Advanced Features

This section has provided a description of the basic functionality of GIST. Further details of this functionality will be described in chapters concerning the design of the implementation.

Some advanced features exist, such as extensive rerouting detection and NAT traversal, but will not be described in this report because they do not appear in the implementation. For details on these features, see the specification document [39].

2.2.2. QoS NSIS Signaling Layer Protocol

As NSIS is meant to be at least a replacement for RSVP [22], which was designed to perform QoS reservations, the QoS-NSLP application is a core part of the suite. It uses the services provided by GIST to install QoS reservations along the entire path of a flow, informing every NSIS capable router on this path of the parameters of the QoS that is required. Like GIST, the QoS-NSLP uses soft-state to install these reservations, meaning that, unless they are refreshed, the reservations will expire after a predetermined amount of time. The reservations are performed end-to-end, meaning one reserve message is propagated along the entire path, while the refreshes are done peer-to-peer, with each pair of peers having their separate refresh and expiry timers. A reservation may also be removed explicitly by an end-to-end reservation teardown.

2.2.2.1. QoS-NSLP Components

Unlike RSVP, the QoS-NSLP is not bound to a particular type of QoS, rather it operates independently from this. It does this by decoupling the reservation processing from both the contents of the QoS parameters, called the QSPEC (QoS Specification) [18], and the addition and removal of actual reservations within the traffic control subsystem of the node, called the Resource Management Function (RMF). The idea is that a flow and thus a reservation can travel through several network domains, each implementing their own QoS Model (QoSM), e.g. IntServ [21] or DiffServ [20], and receive the required QoS in each of these domains, signalled by the QoS Specification (QSPEC) in the reserve message. This QSPEC attempts to provide a number of common QoS parameters that can be used across different QoSM domains.

This subdivision of the QoS-NSLP into a processing part and the RMF is illustrated in Figure 2.5. The former is responsible for communicating with GIST and any user applications requesting Quality of Service, processing messages and storing state about them and passing any reservation instructions to the RMF. Note that a concrete definition of any interaction with user applications is beyond the scope of the QoS-NSLP specifications. The latter is responsible for interpreting the QSPEC, which is opaque to the processing part, authorising and admitting reservations and actually performing them, interacting with the traffic control subsystem provided by the operating system of the host. This means that the functionality of the RMF is fully dependent upon the QoSM used, while the functionality of the message processing part remains constant. Two of these QoS models are described in sections 2.2.3.1 and 2.2.3.2.

2.2.2.2. QoS-NSLP Message Processing

The QoS-NSLP processing part as described in the last section is defined in [33], which dictates four different message types:

RESERVE This message creates, refreshes, modifies or even removes reservation state within the QoS-NSLP node. This is the only type of message that has any effect on QoS state within the node. Many objects included in this message are optional and

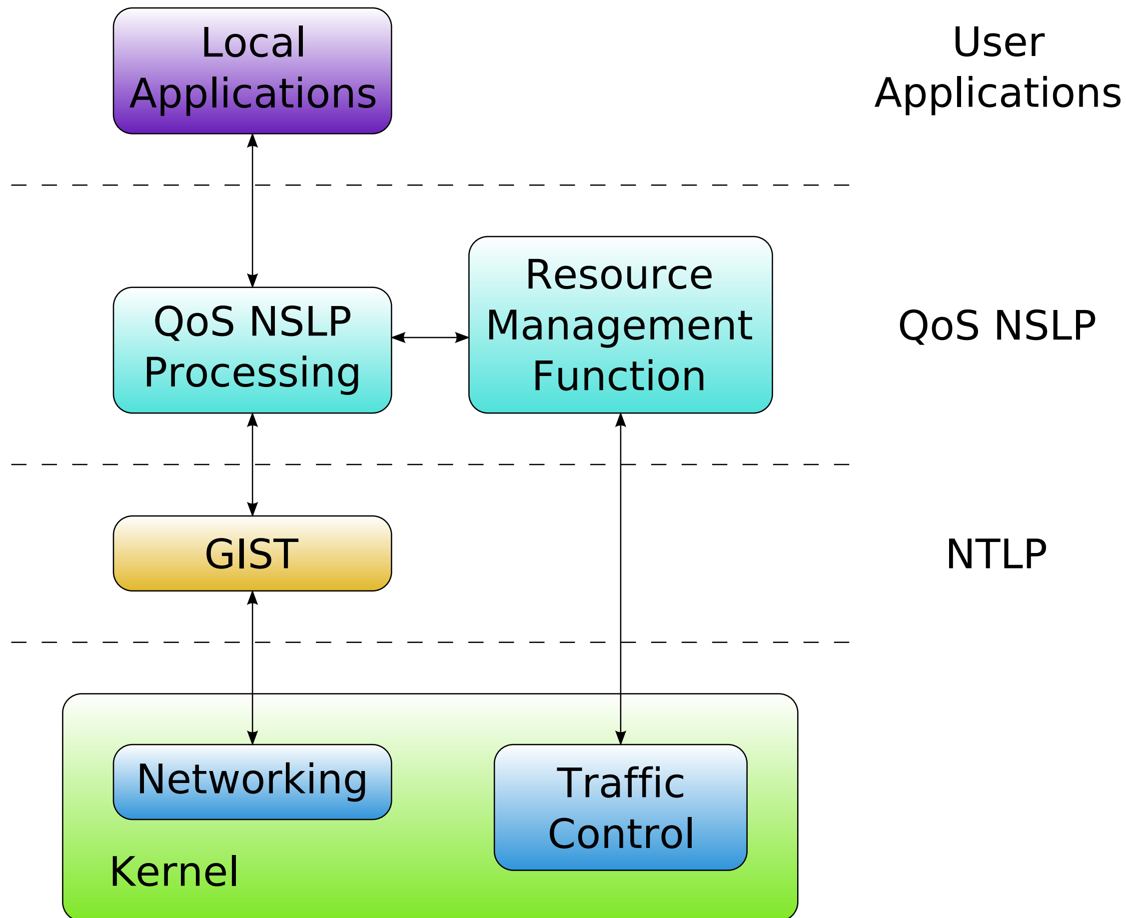

Figure 2.5.: Components within the QoS-NSLP system and their interactions.

depend on the situation in which the RESERVE is sent. Among the objects are two sequence numbers, one unique for the end-to-end reservation request, called the Request Identification Information (RII), the other unique in the sequence of message exchanges between two peers and having local significance only, called the Reservation Sequence Number (RSN). This last sequence number is the only mandatory object in the message. Also included are some refresh timer information in the REFRESH_PERIOD object, a PACKET_CLASSIFIER to mask out particular parameters of the MRI and an object which allows binding a different session to this one called BOUND_SESSION_ID, the function of which will be explained in section 2.2.2.4. Also included of course is a QSPEC describing the QoS parameters of the reservation in question. Besides this there can also be a second QSPEC that allows for layering, which will also be explained later.

QUERY This message is similar to a RESERVE message in the respect that it also carries a QSPEC and generally travels down the entire path. The main difference

is that a QUERY message just gathers information and does not modify reservation state in any way. It can be used to probe the availability of QoS parameters along the path, such as bandwidth available within a certain traffic class. It also provides a means to perform receiver-initiated reservations, where the flow receiver first sends a QUERY message towards the sender, which in turn sends the required RESERVE message. A QUERY message may include RII, BOUND_SESSION_ID, and PACKET_CLASSIFIER objects, as well as one or two QSPEC objects. The first QSPEC is the only mandatory object in this message type.

RESPONSE This message is sent in reply to either a RESERVE or QUERY message to convey the results of the requested action. The most important and only mandatory object of this message type is the INFO_SPEC object, which contains this result by means of and error class and code, some identification information about the node that generated the RESPONSE and optionally additional information. Note that this object need not represent and error condition, as there is also an error class indicating success for different types of actions, e.g. successful reservation setup or teardown. Other objects that may be included are either a RII or RSN and one or two QSPEC objects. In case of an error these QSPECs can contain information about which parameter caused this error.

NOTIFY This message is similar to the RESPONSE message, with the main difference that it occurs asynchronously, i.e. it is not sent in response to any previous message. It typically conveys error information in the INFO_SPEC object, which is again mandatory. Apart from this it can also contain one or two QSPEC objects, which can convey additional information about the error that occurred.

2.2.2.3. QoS-NSLP Example

How the message types described in the previous section are used to provide the functionality of the QoS-NSLP is best illustrated in an example. The message flow for the example described here can be seen in Figure 2.6. Suppose that some application on a NSIS capable host, in this example called the QoS NSIS Initiator (QNI), is sending a flow to another host, called the QoS NSIS Receiver (QNR), and that this application wants the flow to receive a certain QoS along the path to its destination. Note that in this example both the flow sender and flow receiver are NSIS capable. It will signal this to the QoS-NSLP application running at this host, via some undefined local API. The QoS-NSLP application will construct a QSPEC which contains the QoS parameters the QNI wants the flow to receive. Ideally these parameters can be interpreted for any QoSM that the flow may traverse. Before creating and sending a RESERVE message, the host will inform its own RMF about the desired QoS, so that it can approve and perform this reservation using its own QoSM on the output interface of the flow. If the RMF indicates that it does not have enough resources the RESERVE message need not even be sent and an error condition is signalled back to the application requesting the QoS. If the RMF is successful in its reservation, the QoS-NSLP will create a new reservation state, referencing it by a newly created SID. It will use this state, which among

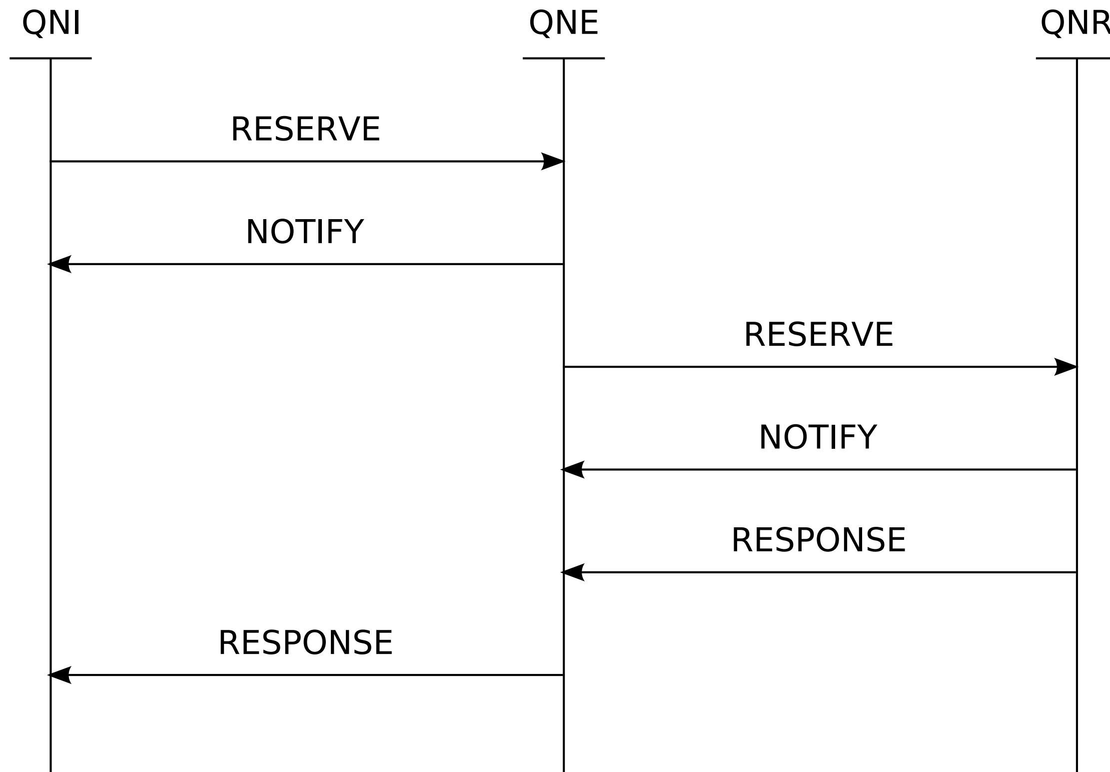

Figure 2.6.: Example of a sender-initiated QoS-NSLP reservation.

other things contains the RSN for the downstream peer, to generate the downstream RESERVE message, which it will pass to GIST. Included in this RESERVE is a RII which is unique to this reservation request. The QoS-NSLP will record this RII value to watch for in RESPONSE messages.

The first downstream peer for this particular flow, which in this case will be called the QoS NSIS Entity (QNE), will receive the RESERVE message and check to see if it has state installed for this SID. In this example it does not, but if it had state installed it would have done some checks on the RSN of the received message, to prevent message duplication and re-ordering, in case unreliable transport is used in GIST. It then send the QSPEC to the RMF, which indicates either that it does not have enough resources for the QoS requested, or that it has performed the reservation successfully. In this example the reservation is successful and the QoS-NSLP creates state for this session, records the RII of the RESERVE to monitor the response, inserts its own RSN into it and send it along to the next downstream node. Additionally, in response to a flag set in the RESERVE message header which requests support for reduced refresh messages, it will send a NOTIFY message back upstream with an INFO_SPEC value that indicates wether or not it supports this. The concept of reduced refreshes will be explained later.

This chain of events continues until, eventually, the RESERVE message reaches the QNR. If the reservation is also successful at this node, it will generate a RESPONSE

message indicating this success in the INFO_SPEC. It will also include the RII of the reserve, so that nodes receiving the RESPONSE know which reservation this pertains, and a QSPEC returned from the RMF. The latter is used to indicated what was actually reserved. This response message is send back upstream and each node will forward it until it reaches the QNI. This node determines that it was the originator for this RII and will stop forwarding it.

If the reservation had failed anywhere along the path, a similar RESPONSE would have been sent from the node where the failure occurred, but with a INFO_SPEC object indicating this failure. In this case the QNI would receive this RESPONSE and send a tearing RESERVE message to remove the state from those nodes that already have it installed.

The reservation is then in place along the entire path. Because this reservation is soft state it will need to be refreshed periodically, which happens asynchronously between peers. What this means is that, unless its own state expires, every node will send refreshing RESERVE messages to its downstream peer, which will not be propagated further downstream. Each set of peer has its own timers governing refresh intervals. If the downstream node has indicated that it supports reduced refreshes, the upstream peer for this node need only include the RSN of the last full RESERVE message in the refreshing RESERVE. The downstream node will see that the RSN is the same as the last RESERVE message and will conclude that the same QoS parameters still apply.

Two other features can be highlighted at this point. Suppose that the flow is in the opposite direction, i.e. from the QNR to the QNI, and that the QNI wants a certain QoS for this flow to be applied. In this case, which is a receiver-initiated reservation instead of the sender-initiated reservation of the example, the QNI can send a QUERY message with a specific flag requesting the QNR to send a RESERVE message. The QNR originated RESERVE message will then behave in much the same way as the one in the example.

Another possibility is a bidirectional reservation. Although reservations in the QoS-NSLP are always unidirectional, session binding between two separate session can be performed, one for the downstream flow and one for the upstream flow. Each RESERVE or QUERY belonging to one of the two sessions contains a reference to the other session through the use of the BOUND_SESSION_ID object. This allows for events such as errors pertaining to one of the sessions to be signalled to the other session as well.

2.2.2.4. QoS-NSLP Layering

To support local QoSM domains, a number of facilities exist within QoS-NSLP that allow use of different QSPECs or transfer properties. Although they are not part of the implementation described in chapter 4, they will briefly be described here.

QSPEC Stacking

As already described in section 2.2.2.2, messages carrying QSPEC information allow the inclusion of a second QSPEC. This allows for a QoSM-specific QSPEC to be carried in these messages for traversal of a specific domain. At the entry node for this domain the

second QSPEC can be "stacked" on the end-to-end QSPEC, while at the exit point it can be removed.

Session Tunnelling
As an alternative to the previous approach of QSPEC section, a whole new session can be started at the entry point of a domain. This tunnelling session is then bound to the end-to-end session using a BOUND_SESSION_ID object. The end-to-end session generally skips over the domain in this case using a different NSLP-ID value.

Aggregate Sessions
This is actually a special case of Session Tunnelling. A local QoSM may want to aggregate several reservations into one local session, to reduce load on the local domain. Again, the local session is bound to the end-to-end session using BOUND_SESSION_ID. This case is used for example in Resource Management in DiffServ (RMD).

2.2.3. QoS-NSLP QoS Models

As already indicated in section 2.2.2.1, the RMF conceptually performs all QoSM specific operations. Each particular QoSM has its own specification document, which perscribes which QoS-NSLP features specifically are used, how the QSPEC should be formated and interpreted, and how these parameters should be mapped onto the specific properties inherent to the QoSM. Two such QoSMs will be described here, one for IntServ and one for DiffServ.

2.2.3.1. IntServ Controlled Load QoS Model

Integrated Services [21] is a QoS architecture that allows per-flow reservations to be in place in routers, with flow characteristics being described using a so-called token bucket. Wikipedia describes a token bucket in the following way [16]:

> The idea is that there is a token bucket which slowly fills up with tokens, arriving at a constant rate. Every packet which is sent requires a token, and if there are no tokens, then it cannot be sent. Thus, the rate at which tokens arrive dictates the average rate of traffic flow, while the depth of the bucket dictates how 'bursty' the traffic is allowed to be.

IntServ defines several types of QoS that can be applied to a flow, of which the "Controlled load" [43] is the only one that will be discussed here. "Controlled load" means that a flow receives the equivalent treatment of a best-effort flow in a lightly loaded network, i.e. a network were there may be other traffic, but no heavy congestion. This definition is intentionally vague in that it does not define hard parameters in terms of packet loss or latency.

The IntServ QoSM [29] as defined for NSIS is actually a very brief definition of RMF functionality. It uses only two QSPEC parameters, one of which is optional. The first and mandatory one is the token bucket parameter, which is used as in the description of IntServ above, i.e. describing the characteristics of the flow on which QoS is requested.

The second parameter, which is optional, prescribes what what action should be taken when a flow exceeds the given parameters, e.g. dropping or reclassifying, the so-called "Excess treatment" parameter. Both of these parameters are defined in the QSPEC document [18]. Note that any interaction with the traffic control subsystem of the host is beyond the scope of the QoSM document.

2.2.3.2. Resource Management in Diffserv QoS Model

Flow-based QoS systems such as IntServ have scalability issues, as each router needs to keep track of one reservation per flow. In response to this Differentiated Services [20] was created, which is a class-based system. DiffServ assumes that a domain of routers, called a DiffServ cloud, is under control of the same entity. Upon entry into this domain flows are classified into a certain predefined Per Hop Behaviour (PHB), a QoS class, and marked accordingly in the DiffServ Code Point (DSCP) field of the IP header. The core routers of them domain need then only concern themselves with the different traffic classes, while the burden of per-flow processing is shifted towards the edge routers exclusively. The only PHB discussed here is the Expedited Forwarding (EF) PHB. This PHB guarantees that flows receive a certain minimum rate over a period of time.

RMD [19] defines a QoSM that adapts the QoS-NSLP of NSIS to this concept, allowing dynamic reservations for DiffServ with a minimum number of resources required in the core routers. As few of the activities performed and described in this report involve RMD, this section provides only a very brief description.

RMD makes use of either the session tunnelling or aggregate session layering mentioned in section 2.2.2.4. This means that for each flow there are two sessions, the end-to-end session initiated by the QNI, which will skip the DiffServ domain and go directly from the ingress to the egress node when a reservation is performed, and a related intra-domain session that travels from the ingress to the egress. In case of aggregate reservations there is one intra-domain session per PHB per ingress/egress pair, otherwise there is one intra-domain session for each end-to-end session. The intra-domain session carries its own QSPEC with contents specific to RMD, derived from the parameters in the end-to-end QSPEC. The interior nodes in the DiffServ domain maintain a minimum of state, operating statelessly at the GIST level by refusing to install state in response to a RecvMessage API call as described in section 2.2.1.1 and by sending data messages in Query mode as described in section 2.2.1.5, and optionally maintaining per-PHB state at the NSLP level. In this way the DiffServ paradigm of shifting the burden of processing to the edges of the domain only is carried through for its QoS-NSLP adaptation.

An typical example of this message flow can be found in Figure 2.7. As the general QoS-NSLP example of Figure 2.6, it displays a sender-initiated reservation from the QNI to the QNR, neither of which are shown. This example concentrates on the RMD domain, with the end-to-end RESERVE message arriving from the direction of the QNI relative to the flow in question. It arrives at the ingress node of the RMD domain, again relative to the direction of the flow given in the MRI, which propagates the message as would normally be the case, with the exception that the NSLP-ID is changed to a value that corresponds to a RAO value that is known to bypass the interior nodes of the RMD

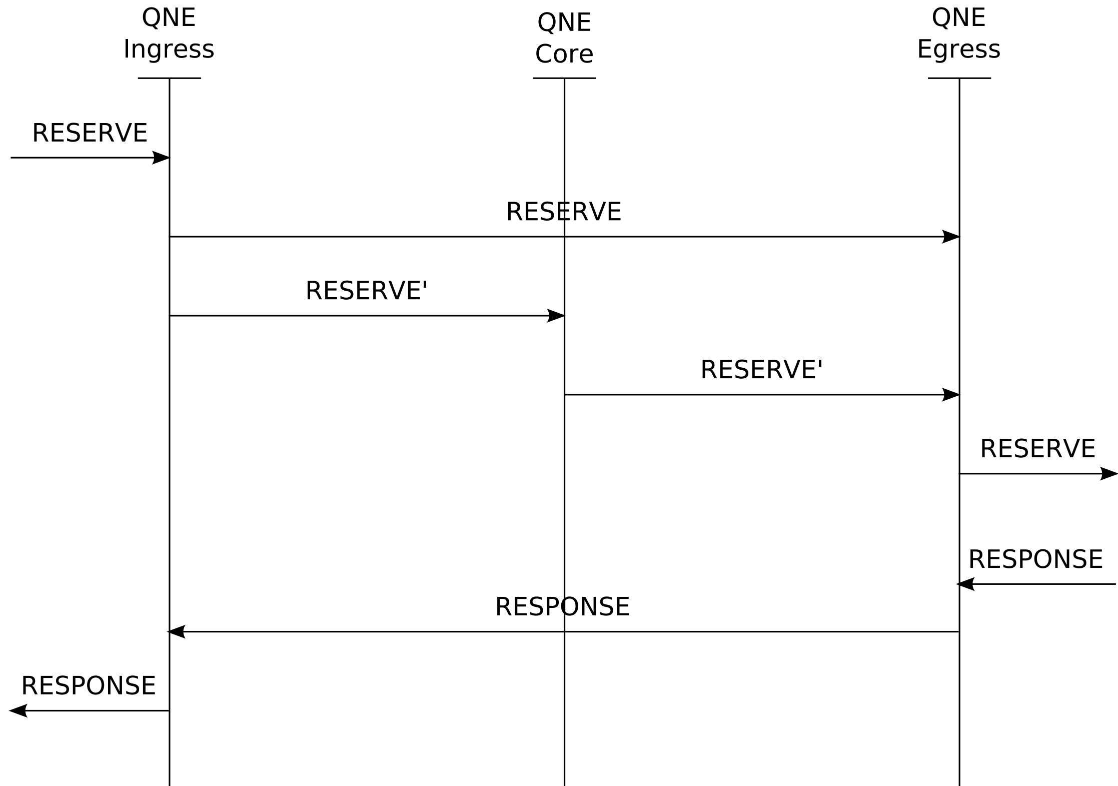

Figure 2.7.: Example of a sender-initiated RMD QoS-NSLP reservation.

domain. The egress node is configured to receive this RAO value and thus receives the RESERVE message, but before propagating it with a restored NSLP-ID value it has to wait for the intra-domain session to complete. In this intra domain session, which as explained can either be a session particular to the end-to-end session or an aggregate session, the ingress node starts by sending a RESERVE message towards the egress node, in which it includes the special RMD QSPEC. Note that RESERVE messages are the only intra-domain message supported in RMD and because of the stateless nature, messages are only allowed to be sent downstream with respect to the MRI, i.e. from the ingress towards the egress node. This is because at the NTLP level messages are either sent piggybacked on GIST Query messages or sent as GIST Data messages in Query mode, allowing the interior nodes to forego maintaining any routing state. Any interior node along the path receives the RESERVE message, if it stores per-PHB state checks if the reservation can be approved, updates the message accordingly and propagates it. Eventually this intra-domain RESERVE also arrives at the eggres, which couples it with the end-to-end RESERVE through the BOUND_SESSION_ID object. If the reservation was unsuccessful it signals this to the ingress node through a RESPONSE message, which will be propagated back towards the QNI, otherwise it will propagate the end-to-end RESERVE towards the QNR as normal.

RMD Features Overview

In summary, RMD has a number of features, some of which have already been mentioned:

- In addition to the successful and unsuccessful reservation described above, RMD also supports reservation refreshes and explicit reservation teardown
- RMD also has its own means of providing bi-directional reservations.
- As already highlighted, the edge nodes of the RMD domain have the choice of either maintaining per-flow inter-domain state or maintaining aggregate inter-domain flow state. Note that end-to-end state is always maintained per-flow.
- In contrast to the example of the previous section, the interior RMD nodes can be allowed to operate fully statelessly. The first method of reservation admission control, as used in the example, is called the reservation-based method, because the interior nodes maintain per-PHB reservation state. The stateless method is called the measurement-based method, as admission is based on continually performed measurements of traffic volume passing through the node, which can be further subdivided into two methods. In the NSIS-aware measurement-based admission control the interior nodes operate much like those in the reservation-based method. They intercept and process the RMD intra-domain RESERVE message, but instead of consulting reservation state, reservations are admitted based on traffic volume measurements. In the probing measurement-based method however, the interior nodes need not be NSIS-aware, i.e. they do not have a GIST and QoS-NSLP application running. Instead, they are configured to remark traffic within one PHB exceeding a certain preset rate, giving all packets within that PHB a different DSCP value. This will be noticed by the egress node, which will inform the ingress node about the failed reservation.
- In case a link or node within the RMD domain fails and traffic gets rerouted and overloads another node, a form of severe congestion notification can be used, also making use of the remarking of packets. This feature is expanded upon in the next section.

RMD Severe Congestion Handling

As some of the work performed involves the Severe Congestion Handling mechanism of RMD in particular, this feature will be described in more detail here. As already described, interior nodes should have some means to inform the egress node of severe congestion, as these interior nodes have minimal or no reservation state installed and are not in the position to decide which flows to terminate. Instead they allow the egress node to make an informed decision about which flows are experiencing congestion and possibly teardown some reservations end-to-end. Like the congestion notification already discussed, interior nodes can do this by remarking the data packets it forwards, i.e. changing their DSCP value to some other predefined value. Combined with the with congestion notification for the probing measurement discussed in the previous section,

this means that per PHB a total number of three additional DSCP values used for marking may be defined:

Notified DSCP This is the DSCP value used for the probing measurement-based admission control described previously. As other methods of admission control may be used, this DSCP remarking value is optional.

Encoded DSCP This value is used to notify the egress node of the amount of congestion experienced in the congested interior node, by using this marking proportionally. This means that for a certain preset time period the amount of traffic is monitored and during the next time period the number of bytes that were over the preset limit and thus dropped are marked in the packets leaving the node according to a certain preset ratio. This means that for example if there were 64 kilobytes overlimit in the last period and the ratio is two-to-one, in the next period 32 kilobytes worth of packets will be marked. Note that for this and the previous marking the same DSCP value can be used, if the egress node can properly determine if the interior node is experiencing severe congestion, possibly with aid of the next DSCP value.

Affected DSCP This DSCP value is optional and can be used in the situation where the node is congested, but has already transmitted enough packets with the *Encoded DSCP* marking to convery the level of congestion. This is allows the egress node to determine which flows passed congested nodes, ever if packets in those flows were not marked with the *Encoded DSCP* value.

Note that each PHB will need its own set of additional DSCP values, as the interior nodes processing marked packets still need to be able to determine the PHB of the traffic.

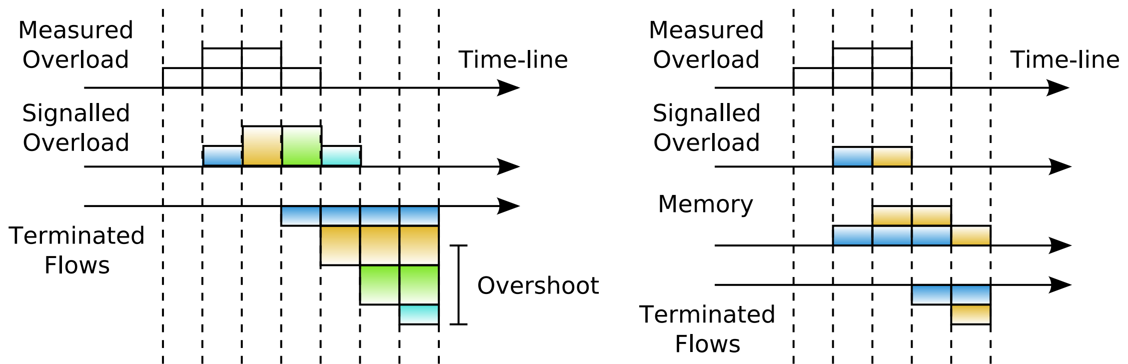

Figure 2.8.: Example of RMD Severe Congestion handling. In the left figure no sliding window memory is applied, in the right figure it is applied.

The termination of the relevant flows causing congestion may take some time and to prevent overshoot in the amount of flows terminated, the interior nodes maintain a sliding window memory of the amount of congested bytes signalled for several time periods, as proposed in [23] When calculating how many packets to remark, the node

will subtract any marked bytes from previous time periods in its memory from the total number of bytes over the congestion limit. This will give the egress node enough time to be notified of the congestion and terminate the flow reservation, causing the traffic volume to be under the congestion limit, before the sliding window memory runs out and the congestion is signalled again. This is illustrated in Figure 2.8, which is taken from [24].

It is also important that if nodes downstream from the congested nodes need are also congested and need to drop packets, they prevent dropping of packets that are already marked with the *Encoded DSCP*, so that no signalling information about the amount of congestion is lost. Note that this only pertains to dropping priority and not scheduling priority, i.e. the packet ordering should not be changed when transmitting the packets further downstream.

3. GIST Design and Implementation

This chapter will attempt to describe and motivate the design choices made during the implementation of GIST and illustrate the process and structure of this implementation. Before this, the work performed by others at the start of this assignment will be discussed.

3.1. Previous Work

As already mentioned in the introduction in chapter 1, the work done in this master's assignment is a continuation of work performed by previous students at the DACS group of the University of Twente (UT). The part of the assignment that involved GIST consisted of bringing up-to-date, both in terms of features and specification draft version, the GIST implementation started by Mayi Zoumaro-Djayoon [44], to achieve the goals of the assignment set out in section 1.1. She made her implementation conforming to draft version 5 of the specification [39] and, partly because of time constraints and partly because of the incomplete draft specification, only included basic functionality.

The first objective of the assignment, after reading all of the relevant specifications, was inspecting the implementation that was delivered at the end of Mayi's assignment. During this stage it became apparent that this implementation, apart from the known omissions, contained a number of severe deficiencies and poor design choices which conflicted with the requirements set in section 1.2, a number of which will be expanded on here:

- It proved no trivial task to migrate the implementation from the workstation it was developed on to a different one. Development was performed on a Debian Linux [1] system in the Python [10] programming language. The program made a number of assumptions about its environment, such as being able to resolve network interface names to IP addresses, e.g. *eth0*, using the system's */etc/hosts* file, which is not the usual way. Although a lot of parameters of the implementation could be set using a configuration file, several of them were still hardcoded in different places in the code, including parameters such as network interfaces and IP addresses used. These issues resulted in having to modify the program quite extensively to even be able to run it on a different computer than the one it was written on.

- Python is a class-based language and the design of the implementation intended to make use of this. The class-based paradigm however was not utilised properly, in that a lot of object were only ever used at the moment of their creation, thus exhibiting function-like behaviour. Quite often global variables were used for inter-class communication, disallowing the use of separate instances of single classes.

- Related to the previous point of using global variables, the program was written without no regard for thread-safety, although the program quite rightly made uses of threads. There was no guarantee that another class might not use the global variables used for communication. The only means implemented to avoid race conditions was setting a static delay in one of two threads known to occur simultaneously.

- The implementation made no clear distinction between network packet contents and network packet logic. What is meant by network packet contents in this case is the actual bit-level contents of packets of the packets sent or received. Packet logic is about which packet should be sent in response to some event. The separation of the two is needed so that, when the bit-level format is changed, such as in the case of new specifications, only one part of the code needs updating, which is part of the requirements set. It also prevents the need for using the same code several times over, since in this implementation packet generation and decoding was mostly done in-place, i.e. in the location of the code where the packet logic needed it. In this case it also meant that decoding of all possible packet contents dictated by the specifications was not implemented, i.e. the implementation only expected the type of packet contents it itself could generate. This could prove a big problem for interoperability.

- It contained little error checking and exception handling, which is an important requirement. This is somewhat related to the previous item, as most error checking and error message generation should be done on message reception. To add a decent level of error checking the code would have to undergo a complete overhaul, as error checking is tightly coupled with such parts of the program. Those errors that were caught were often not handled gracefully, with the implementation continuing an exchange of known erroneous messages.

- Several functionalities that should have been implemented appeared to have been hardly implemented at all, such as MA functionality, which just consisted simply of setting up a TCP connection.

- Such debugging output that was available was cryptic and sometimes even in binary format, which tended to cause the terminal window to which it was output to malfunction. This made it hard to determine what exactly the implementation was doing during runtime and afterwards.

For these reasons it was considered that the amount of time it would cost to correct all of these deficiencies so that the implementation adhered to the set requirements and fully update the implementation to the latest specifications would be more than or equal to the amount of time involved in starting an implementation from scratch according to the requirements, re-using the concepts of the previous implementation and learning from its defects.

3.2. Development Environment

Like the previous implementation, this implementation of GIST was made using the Python [10] programming language on a Linux operating system, partly so that the concepts in this previous implementation could easily be re-used.

Linux was chosen because, combined with the large number of applications available, it provides an ideal testbed for developing new network applications and protocols. Because of its open nature, any adjustments that may need to be made to allow GIST to operate can be made. For example, the highly flexible *iptables* [9] framework that Linux provides was used extensively to influence network traffic in different ways. A feature of Linux that was primarily used for the QoS-NSLP was the traffic control subsystem, which allows shaping of traffic flows. Another reason for using Linux is the User-Mode Linux [14] testing setup, which will be described in chapter 5. The particular Linux distribution that was used was Gentoo Linux [3], because of the amount of past experience in using it.

Initially there was no choice but to use Python, as the previous implementation was also done in this language. When it became apparent that the implementation would have to be made from scratch however, Python proved to be an excellent choice for the development of a prototype implementation of GIST, as it can potentially fulfil a lot of the requirements set in section 1.1. Wikipedia has the following to say about Python [17]:

> Python is a programming language created by Guido van Rossum in 1990. Python has a fully dynamic type system and uses automatic memory management; (...) Python is notable amongst current popular high-level languages for having a philosophy that emphasises the importance of programmer effort over that of computers and for rejecting more arcane language features, readability having a higher priority than speed or expressiveness.

and:

> In contrast to some lower-level languages, Python's design does not emphasise runtime speed. While Python implementations are generally comparable to that of other bytecode interpreted languages, a concern for clarity of code always comes first in Python's design.

This philosophy is well suited to the requirements, both in code readability and extensibility and in the fact that it should serve as a prototype, as a prototype implementation should prioritise clarity in the workings of the implementation over performance.

Furthermore, Python is an interpreted language, compiling source files to bytecode on execution. The dynamic type system and memory management make sure that the programmer can spend his or her time more efficiently, having more time available to work on the structure of the program. These properties should aid in rapid development and adjustment of the GIST implementation.

3.3. High-level Program Structure Overview

The GIST implementation uses the Object Oriented (OO) programming paradigm features offered by Python, as this provides support for a modular architecture, which is inherent to the requirements. The main server class is named *gistServer.Server*, one instance of which should me made to create a running GIST server. This server class contains instances of several other classes, which work as threads and provide certain types of services to the server. The threads of the service classes monitor their specific domain and notify the GIST server of these incoming events. This model matches the conceptual model of a server, in the sense that any action is reactionary, i.e. it is caused by a certain external or internal event, with each type of event monitored by a thread. The server can also use the service classes for outbound communication that should use the domain of the service class. Although this is not strictly necessary, it provides a grouping of functionality within the same class. As can be seen in the diagram in Figure 3.2, the service classes are:

gistAPI.APIService This class provides communication to and from the NSLP applications running locally. The GIST server can perform the outbound service primitives described in section 2.2.1.1 on the API service as function calls, and the API service in turn can perform the inbound service primitives on the server. The API service is responsible from maintaining a list of running NSLP applications and delivering the service primitives received to the right application. This can be either another Python module, in which case the API service is responsible for running the application, or an external application.

gistRaw.RawService This class is responsible for sending and receiving message in Query Mode. Its name is derived from the method of sending, i.e. using raw sockets.

gistUDP.UDPService This class is responsible for sending and receiving messages in Datagram Mode, i.e. using normal UDP encapsulation.

gistICMP.ICMPService This class monitors ICMP traffic sent to the node. As explained in section 2.2.1.4, a certain ICMP message in response to a Query message allow a node to deduce that it is the last NSIS capable node on the path.

gistTCP.TCPService This class is responsible for handling incoming and outgoing new TCP connections, one of the Connection Mode protocols used in this implementation. Note that every active TCP connection is managed by its own *gistTCP.TCPConnection* class, which also has its own thread and relays incoming data to the server object.

gistTCP.TLSService This class is responsible for handling incoming and outgoing new TLS over TCP connections, the other Connection Mode protocol used in this implementation. TLS connections are also managed by their own class, *gistTLS.TLSService*.

Although the server can communicate with the network service classes directly, in typical cases this is performed through certain state machines that the server contains. These state machines handle state changes and set timers, but also store state information. The rules of the state machines are dictated by the GIST specification and are further described in section 3.6. There are two types of states stored by the GIST server, routing state in the form of either a *gistQuerySM.QuerySM* or a *gistResponderSM.ResponderSM* instance, which as stated in 2.2.1.3 is keyed by the pair of NSLP-ID, MRI and SID, and MA state in the form of a *gistMASM.MASM* instance. The routing state state machine used depends on the role of the node during the initial discovery procedure and subsequent refreshes. In the case of the Query state machine it contains a timer to perform refreshes and both contain state expiry timers, implementing the soft state principle. As a timer expiry is an event they are also implemented as threads, notifying the GIST server class of the relevant event. There is one MA state machine per MA, which also contains refresh and expiry timers.

The implementation also contains a message decoding and encoding subsystem, decoupled from the service classes and state machines. This subsystem allows any part of the program to treat messages as hierarchical objects and read, write and modify messages independently from their coding structure. This allows for flexibility on changes to the bit-format and makes sure that there is only one place in the program where messages are encoded and decoded. Tightly coupled with the message subsystem is strict error checking on decoding, providing a certain amount of robustness. GIST defines a number of error messages that can be sent in response to a malformed or erroneous message, possibly including information about the error. Python provides exceptions and every GIST error is mapped to its own exception class. If an error occurs on decoding, the message subsystem raises the correct error class, instantiated with parameters containing details about the error. The GIST server can catch this exception, use it to produce a complete error message and send it back to the node that caused the error.

3.3.1. Threading Model

As already pointed out, there are two types of execution threads and thus two sources for events occurring that the server needs to respond to, i.e. external and internal events. External events are generated by the service classes pictured in Figure 3.2. These can be either received API service primitives or received network traffic. Additionally, each established TCP or TLS connection has its own object and corresponding thread, receiving network traffic for that connection. A single thread per connection is needed, as reading data from a connection on which nothing is received results in a so-called blocking call, halting program execution at that point. This could be handled from within the same thread by for example using the *select* system call, but since the Python *threading* module provides multi-threading in a high-level OO model, this fits better within the structure of this program. Internal events are generated by timers belonging to state machines, each timer executing in its own thread, also using the Python library [11] *threading.Timer* class.

Both of these types of threads need to perform operation that may change state main-

tained within the GIST server. To provide thread-safety, each thread will need to acquire a threading lock provided by the server, implemented using the Python *threading.Lock* object, before doing any such actions. Acquiring the lock guarantees that only one event at a time is handled. If another event is already being processed, acquiring the lock will block the calling event thread until the executing thread releases the lock. To be able to reach the lock any threading class within the GIST implementation maintains a reference to the main server object, an instance of *gistServer.Server*. Using a single lock to prevent several threads from accessing and possibly changing state stored within the GIST implementation greatly simplifies safe multi-threaded programming, allowing the code to access and change objects in the same way it would in a single-threaded program, as long as it has acquired the lock. Any alternative method, e.g. passing messages in a thread-safe way between different parts of the program, would require more complexity. The only requirement for using this single lock is that any operation performed while the lock is acquired is dealt with as quickly as possible and does not sleep or block in any way. This was taken into account while implementing the program.

Fatal programming errors are generally represented by Python exceptions. To prevent one thread from having a fatal exception while other threads continue to operate, all threads are executed in a *try* statement, in which any unexpected exceptions can be caught. If this occurs, the exceptions handler will make sure to append the exception in question to the log, see section 3.8.1, and terminate the program. This greatly helps debugging, as errors cannot be overlooked.

3.4. Network Interaction

The three modes of transmission used in GIST each have their own service class or classes, the implementation of which will be described in this section.

3.4.1. Query Mode

Query Mode transmission and reception is provided by the *gistRaw.RawService* class. Transmission is performed using so-called raw sockets, which allows the program to construct its own network and transport layer header, in this case a IP and UDP header. The reason for doing this is that, by default, Linux does not allow a program to transmit a IP packet with a source address that is not among the IP addresses of its own interfaces. While constructing the IP header, the *gistRaw.RawService* class also inserts the RAO, with as parameter the NSLP-ID. Note that, although the specifications state that there is no one to one relationship between the RAO parameter and the NSLP-ID, at this point in time there is no clear mapping mandated between the two. Using raw sockets in Linux is something that is allowed for the *root* user only. Because of this is the GIST server should only be run as *root*.

Reception of Query Mode messages is done with the *ip_queue* handler Linux provides. This allows *iptables* to filter out specific network messages and send them to a userland process, in this case the GIST server. The filter used is the *ipv4options* match extension provided as a netfilter kernel patch [9]. Natively Python does not provide means to

receive *ip_queue* messages, but this is provided by the external *ipqueue* module [5]. Upon reception, the *gistRaw.RawService* class checks if the RAO parameter value matches any of the NSLP-IDs of the applications running at this node. If there is a match, it will pass the packet along to the GIST server. Otherwise, it will tell *netfilter* to continue sending the packet along the MRI path.

3.4.2. Datagram Mode

The *gistUDP.UDPService* provides transmission and reception of Datagram Mode messages. Transmission of such messages is simply done using the basic UDP socket functionality that Python provides. Upon reception, the TTL value of the IP header is also needed. This is done using an external module called *eunuchs* [2], which wraps the Linux *recvmsg* function, as the Python library [11] does not natively supply this. This function allows meta-information to be received along with the payload of a network packet, such as the TTL value. Unfortunately this module needed a slight modification, as it did not relinquish the Python *Global Interpreter Lock* before performing a blocking read operation, i.e. waiting for data to come in on the UDP socket.

3.4.3. Connection Mode

The Connection Mode service classes are slightly more complicated. Both Connection Mode protocol stacks implemented, i.e. TCP and TLS over TCP, provide a service class with the same generalised interface. This generalisation allows for future addition of other protocol stacks. For making new connections they provide a *connect* method. Conversely, for registering new incoming connections they contain a *listen* method, which should only be called when GIST receives a Query message and includes a Protocol Stack in its response, and includes the querying node's NLI as parameter. When a new connection then arrives within the listening thread of the service class, it is checked against the registered NLI's. This provides a security measure of only accepting incoming connections from nodes with which this node is currently setting up routing state. An expiry timer is also provided within the service class to make sure that registered incoming connections are cleaned up after a certain period of inactivity.

When a new connection is setup, either through actively connecting or through passively accepting a new connection, the service class spawns an instance of the connection class belonging to the protocol stack. These connection objects also have the same interface, providing a listening thread for this connection and a sending method for actively sending data on the connection.

Implementation of the TCP connections is performed using the native socket provisions in Python. For TLS an external module is used, called *TLS Lite* [13].

3.5. Message Processing

As already mentioned, message encoding and decoding and subsequent error-checking is a separate part of the program, implemented in the *gistMsg* module. This allows the

rest of the program to abstract from the actual bit-level format of the messages and compose, read and change messages in a format that is actually quite close to the GIST specifications, in that it can deal with the message and parameter names as they are used within this specification. Message decoding and encoding consists of two layers, message objects which directly represent the bit-level contents of a message and can be arranged hierarchically in a tree structure, and the interface towards the rest of the program which abstracts slightly from this.

3.5.1. Message Objects

As the contents of GIST messages can be represented hierarchically, the most natural way to represent this in a class-based language is with a tree of objects. This allows for dynamic composition and re-compositing of messages by parts of the program that need to perform these operations on messages. Each GIST message starts with a common header and has a number of so-called Type Length Value (TLV) objects. This consists of a small sub-header listing the type and byte-length of the object, along with the object that actually describes the content, such as the MRI or SID. This object then contains the parameters relevant to that object type. Analogously to this, the root object for any GIST message is an instance of the *gistMsg.CommonHeader* class. It can contain several *gistMsg.TLV* objects, in the order in which they appear in the byte-level message, each of which has a *Value* parameter that contains the object in question. Several objects, such as *gistMsg.MRI* and *gistMsg.Error* contain even further message objects as children. An example of this hierarchical structure is illustrated on the right side of Figure 3.1.

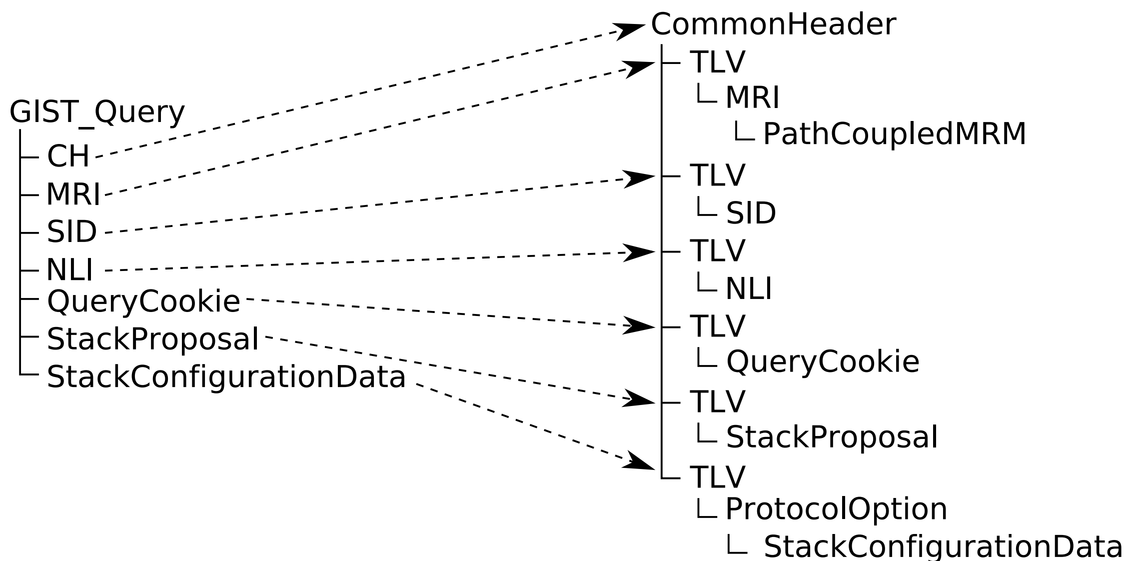

Figure 3.1.: Example of a Query message object tree on the right, with its interface representation on the left.

Each message class has the same interface, i.e. several class methods that are common

to this type of object. They are:

__str__ This is a special method for Python. It will be called every time that the object needs to be implicitly or explicitly converted to a string, e.g. when printing an object. In this implementation it is used extensively for logging purposes as will be described in section 3.8.1. For every object it will output a description of the parameters contained within that object, concatenated with with description strings from each of its children. For the root *CommonHeader* object this means that simply printing it to a file or standard output will give a complete description of the message and each of the child objects contained within that message, including their parameters.

__init__ This is the constructor method for any class in Python. Generally for message objects it is empty, as there are two ways to create a message objects. This means that, along with the constructor, one of the two following methods needs to be called.

create This method will initialise all of the parameters of the message object. It is the method that should be used when creating a new object.

decode This method takes as a parameter a byte string and attempts to decode this to the object based representation. Typically this is called on the *CommonHeader* object, which takes a full byte-level message, creates the relevant child objects and calls the *decode* method on them.

calcsize This method returns the size in 32-bit words of the byte-level message it represents. Calculation is done on the spot, based on the parameters of an object and any possible child objects it contains, calling *calcsize* on them if needed.

encode Finally this method gives the byte-level representation of what the object represents, appending its own byte code with the byte representation of any of its children. Again, this would typically be called on the *CommonHeader* object, giving a complete byte-level representation of the message for network transmission.

3.5.2. Message Object Interface

For ease of handling each type of GIST message is represented by its own class, which are subclasses of *gistMsg.GIST_Message*. Creation of these messages is simply performed using the constructor of the relevant message class. Objects can then be added with the revelant methods of this class, e.g. *add_QueryCookie*. Internally these classes hold the *CommonHeader* object of the message and several other objects. An example of a Query message can be seen on the left side of Figure 3.1. When the *encode* method is called, the message class composes the object that were added to it before in the right order, so that the part of the program using this class need not worry about the specific object order, and relays the *encode* call to the *CommonHeader* object. It also does some additional checking to verify that the message contains all of the mandatory objects for the message

type in question. This the main reason for this further layer of abstraction, as it adds and additional layer of message checking, only allowing messages to be created only in a way that conforms to the specification. It also means that none of the other parts of the program need worry about such things as the internal order of parameters within the message.

For message reception the function *gistMsg.get_message* is provided. This function takes as input a byte-string and returns the relevant message class as described in the previous paragraph, as well as doing some extensive checks on the validity of the decoded message. Once the message is returned, it can easily be modified in-place, either by calling any of the *add_* methods or by modifying the parameters of the objects in the message directly, and encoded again for down- or upstream propagation.

3.5.3. Exceptions

The GIST specifications [39] dictate a number of error types and subtypes for different error situations. The bulk of these errors are caused by erroneous messages that are received, e.g. because message decoding failed or because the message is somehow inappropriate to the situation. This causes the error checking and error message generation system to be tightly linked with message decoding. As Python provides the facility for raising and handling exceptions, this comes as a natural way for dealing with any error that may occur, skipping all further processing and dealing instantly with the response to this error. Each different error defined in GIST is mapped unto its own class, which can be raised as an exception. These error objects are instantiated with as argument the *CommonHeader* object of the message which caused the error and possibly some additional information, such as a debugging comment. All of these error classes are derived from *gistException.MessageError*, which has a method *get_error_message* that allows for instant generation of an error message relevant to the error that occurred. In this way the part of the program calling for message decoding can easily catch error situations and inform the node that caused the error about this.

3.6. State Machines

The GIST server contains two different kinds of state machines, one for storing per-session routing state and one for storing Messaging Association state. Both of these state machines are well defined in the GIST specifications [39]. In order to implement these specifications as accurately as possible, each state machine is directly represented as a Python class. Within this class the current state is stored, as well as any other information that needs to be stored. It also contains timers relevant to this state machine, represented as Python *threading.Timer* objects. The external events to which the state machines need to respond are defined as methods of the class, each with parameters relevant to the event. Two other methods of the state machine classes are a constructor, an explicit descructor and also the special Python *__str__* method, which outputs a neatly formatted string describing the state and parameters of the state machine and is used

for logging purposes, much like the message objects of section 3.5.1. The list of possible events is given in table 3.1, taken directly from the GIST specification.

Event Name	Meaning
rx_Query	A Query message relevant to this state has been received.
rx_Response	A Response message relevant to this state has been received.
rx_Confirm	A Confirm message relevant to this state has been received.
rx_Data	A Data message relevant to this state has been received.
rx_Message	*rx_Query* or *rx_Response* or *rx_Confirm* or *rx_Data*
rx_MA-Hello	A MA-Hello message relevant to this MA has been received.
tg_NSLPData	A NSLP application has requested data transfer using the SendMessage API call.
tg_Connected	A protocol stack for particular MA has finished connecting.
tg_RawData	GIST wants to send bytes over this MA.
er_NoRSM	A *No Routing State* error message relevant to this state has been received.
er_MAConnect	A protocol within the stack for a particular MA has failed to connect.
er_MAFailure	A MA has experienced a failure.

Table 3.1.: GIST state machine events.

3.6.1. Routing State

The state machines storing and handling routing state are subdivided in one for the querying role and one for the responding role. Routing state is stored in the *gistServer.Server* object, keyed by the combination of NSLP-ID, MRI and SID, as dictated by the specification. This storage is implemented using the dictionary object type native to Python, which allows a look-up by any Python object or combination of objects. This means that each session has its own routing state within the GIST node. If the node initiated peer discovery with a Query message this routing state will be represented with an instance of *gistQuery.QuerySM*. Generally this will be the case for the routing state towards the downstream peer with regard to the MRI. The node that intercepts the Query message and responds to it will have a *gistResponder.ResponderSM* object as routing state.

The responsibility of this routing state is to store information relevant to the session, such as which MA is to be used and what the NLI of the peer is, and the state of the peer discovery procedure. The state transition diagrams for this last function, for both the Query Node State Machine and Responder Node State Machine, is lifted directly from sections 6.2 and 6.3 of the GIST specifications [39].

3.6.1.1. Query Node State Machine

The state machine for the querying node is show in Figure 3.3. It governs the transmission of Query and Confirm messages and ultimate state expiry. Each query state

machine has three timers:

No_Response The amount of time to wait for a Response message, once a Query is transmitted.

Refresh_QNode The time between sending Query messages to refresh the routing state.

Inactive_QNode The amount of time that should pass before the state can be removed. During this time no data revelant to this routing state should be sent or received.

Every time an external event occurs in the state machine, one or more actions need to be taken. The following detailed list of these actions, given in pseudo-code, is referenced by the state transitions in Figure 3.3:

```
Action 1:  store the message for later transmission
Action 2:  if number of Queries sent has reached the threshold
               // nResp_reached is true
               indicate this as error to NSLP
               destroy self
           resend Query
           start No_Response timer with new value
Action 3:  // assume the confirm was lost
           resend Confirm
           restart Refresh_QNode and Inactive_QNode timers
Action 4:  if a new MA state machine is needed, create one
           if the R flag was set, send a response
           pass any NSLP data to the correct application
           send any stored data messages
           stop No_Response timer
           start Refresh_QNode and Inactive_QNode timers
Action 5:  send Data message
           restart Inactive_QNode timer
Action 6:  Terminate
Action 7:  pass any data to the NSLP
           (re)start Inactive_QNode timer
Action 8:  send Query
           start No_Response timer
           stop Refresh_QNode timer
```

3.6.1.2. Responder Node State Machine

The state machine for the responder node is show in Figure 3.4. The state machine represents the passive side of a routing state between two peers, in that it only responds to refreshing Queries and never actively tries to refresh the state itself. This means that the state machines governs the transmission of Response messages in reply to queries and state expiry if no Query is received for a certain amount of time. As can be seen,

the use of Confirm messages is optional, this is a matter of local policy. There are two timers within this state machine:

No_Confirm If a confirm is requested by this node, by means of setting the R flag in the Response message, this represents the amount of time the node should wait, after sending a Response, before either retransmitting this Response or letting the state expire.

Expire_RNode The amount of time that should pass before the state can be removed. This timer is reset every time a Query or Confirm message is received.

The action list, with action numbers referenced in 3.4, is the following:

```
Action 1:   // Confirm message is required
            send Response
            (re)start No_Confirm timer
Action 2:   pass any piggybacked data to the NSLP
            start Expire_RNode timer
Action 3:   send the Data message
Action 4:   pass data to the NSLP
Action 5:   // Confirm message is not required
            send Response
            start Expire_RNode timer
Action 6:   send No Routing State error message
Action 7:   store Data message for later transmission
Action 8:   pass any piggybacked data to the NSLP
            send any stored Data messages
            stop No_Confirm timer
            start Expire_RNode timer
Action 9:   if number of Responses has reached threshold
               // nConf_reached is true
               destroy self
            else
               send Response
               start No_Response timer
Action 10:  destroy self
```

3.6.2. Message Association State Machine

All instances of this state machine are also stored in a dictionary in the *gistServer.Server* object, keyed by the peer node's unique identity and IP address and the protocol stack for this MA. This means that, unlike the routing state, state is not stored per session. Whenever a new session is started that wants to use a particular stack of protocols, such as TLS over TCP, to a particular node, GIST first checks to see if a MA for this is already present in the server. If it is found, no connection setup need take place and the MA can be reused. In this way a MA can be used by several different sessions at once.

The MA state machine constains three timers:

SendHello When this timer expires a MA-Hello message should be sent to the other node.

NoHello This is the amount of time the node has to wait before closing the MA connection and letting the state expire. During this time no MA-Hello or other messages should be received, otherwise the timer is stopped.

NoActivity The amount of time the MA should be inactive, i.e. any messages other than MA-Hello messages are sent or received, before the state machine moves to the Idle state.

Note setting the R flag for MA-Hello messages is optional, indicating the desire for a MA-Hello to be sent in response. The graphic representation of the state machine is given in Figure 3.5. The list of actions corresponding to this state machine is the following:

```
Action 1:  pass message to transport layer
           (re)start NoActivity timer
           (re)start SendHello timer
Action 2:  (re)start NoActivity timer
Action 3:  if reply requested
               send MA-Hello
               restart SendHello timer
Action 4:  send MA-Hello message
           restart SendHello timer
Action 5:  queue message for later transmission
Action 6:  pass outstanding queued message to transport layer
           stop any timers controlling connection establishment
           start NoActivity timer
           start SendHello timer
Action 7:  stop NoActivity timer
           stop SendHello timer
           start NoHello timer
Action 8:  destroy self
Action 9:  if reply requested
               send MA-Hello
           restart NoHello timer
```

3.7. Program Flow

The central part of the GIST implementation is the *receive* method of the *gistServer.Server* class. All service threads that receive network messages pass the data to this method, which distributes it again to other parts of the program. The first thing that happens in this thread is a call to the *gistMsg.get_message* described in section 3.5.2, converting the received byte-string to an object based representation. As this function already does

an extensive amount of error checking, any possible *gistException.MsgError* needs to be caught at the end of the *receive* method. This also allows for the next step in this method, checking if the message contains the contents required for the transfer mode in which the message was received. If an error or omission is found within the message, the code can raise the appropriate exception itself. The exception handling code at the end of the method then produces a correct error message and sends it back to the node which originated the erroneous message. After this error checking stage the receive method can branch into the different actions that need to be taken depending on the message types. Specific Error messages that require an action, such as the *No Routing State* error, are sent to the relevant state machine, as are MA-Hello messages. For the other types of messages the processing is a little more complicated, but basically it also involves either looking up or creating the routing state that is relevant to the message received and calling the event method corresponding to the message on the state machine, as well as passing any NSLP payload to the relevant application.

3.7.1. Example Program Flow

To illustrate how the different parts of the GIST implementation described in this chapter interoperate, this section will describe an example program flow, taking as scenario the responding node in Figure 2.4. Extensive logging output of this scenario can be found in appendix A.

For this node the sequence of events starts with the reception of a Query message from the querying node. This is intercepted by the *gistRaw.RawService* object, which does some checks to see wether the RAO parameter matches any of the NSLP-IDs hosted at this node. Once it has established that the data received is valid for this node, it will inform the *netlink* system that it has accepted the packet, obtain the server threading lock and call the *gistServer.Server.receive* with as parameter the UDP payload of the received datagram. As described in section 3.7, this will use the message decoding system through the *gistMsg.get_message* function, which returns an object representation of the message that allows easy inspection of the contents. If the Query message passes all of the error checking, it will be handled in the Query section of the *receive* function. This will determine, based on the NSLP-ID, MRI and SID, that it does not have any routing state installed yet for this session. It will issue an API call through *gistAPI.APIService.RecvMessage*, giving as parameters an empty payload and *Routing-State-Check* set to true, as well as other information about the received Query message. If the relevant NSLP application wishes to set up routing state, it will return this. The GIST server then creates a *gistResponder.ResponderSM* as routing state for this session and passes the Query message to it. The constructor for the *gistResponder.ResponderSM* object will call *rx_Query* on itself. Within this method, this state machine then builds a Response message in reply to this Query, including all of the protocol stacks it supports, in this case TCP and TLS over TCP. As it does this, it calls the *gistServer.Server.get_ma* method, which inspects the MA dictionary of the GIST server. In this example it finds that this node does not have a MA with this particular node for either protocol stacks and will automatically create the state machines for them. The *gistResponder.ResponderSM*

object will store the returned references to these newly created MA state machines. Upon creation these MA state machines will be in the *Awaiting Connection* state and will inform the relevant service classes, i.e. *gistTCP.TCPService* and *gistTLS.TLSService*, that it will accept incoming connections from the Querying node, the NLI of which each MA state machine has stored. After this, the *gistResponder.ResponderSM* object shall transmit the Response message it composed using the *gistUDP.UDPService.send* method, which simply sends the message in Datagram mode to the Querying node.

The next event pertaining to this session that this node sees is a new TCP connection, which arrives at *gistTCP.TCPService*. This checks to see wether it has the source IP address of this connection registered. As the *gistResponder.ResponderSM* object performed this registration, it will accept the incoming connection and create a new *gistTCP.TCPConnection* to manage it. It will also assign this connection a temporary placeholder MA state machine, called *gistTCP.FakeMASM*. This is needed because the *gistTCP.TCPConnection* will simply perform *rx_Message* on its registered MA state machine, while this is not allowed by the real state machine the connection belongs to, as this is still in the *Awaiting Connection* state. This is exactly what happens as the Confirm message arrives over the newly created TCP connection, the receiving thread in *gistTCP.TCPConnection* calls *gistTCP.FakeMASM.rx_Message*. This in turn calls *gistServer.Server.receive*, giving as parameters the received data and the newly created *gistTCP.TCPConnection* class. The *receive* method again does the decoding and error checking, in particular checking that the first message on this new connection is a Confirm message. It then looks up the routing state, finds the *gistResponderSM.ResponderSM* object and calls *rx_Confirm* on it, again passing as parameters the received message and the newly created connection object. This checks all the contents of the confirm message, looks up the MA state machine in the server relevant to this session and with the protocol stack of the *gistTCP.TCPConnection* object and matches this against the MA state machines it had stored when generating the Response message. It will also perform *gistMASM.MASM.tg_Connected* on the MA state machine it has found and, if there is a match with the stored state machines, makes sure the other MA state machine is cleaned up, if it is still in the *Awaiting Connection* stage. The routing state, in the form of the *gistResponder.ResponderSM* object, then has one fully connected MA registered to it, ready for use. The Confirm message will also carry the NSLP payload, which the routing state machine will submit to the application through the *gistAPI.APIService.RecvMessage* method.

3.8. Miscellaneous Features

This section describes some smaller features implemented in GIST that did not seem to fit elsewhere.

3.8.1. Logging

The *gistServer.Server* object contains a set of logging objects, that can be specified in the configuration file *gistOptions.py*. These logging object provide the same interface, a *log*

method with as parameters a logging string and a three-tiered log level, so that several classes can be implemented that process and represent the logging data in different ways. On initialisation, which happens in said configuration file, the log level of the particular object can be specified. Only message of this log level and lower are processed. In this implementation two logging objects are provided, *gistLog.Print*, which simply outputs to the text console, and *gistLog.File*, which writes the output to a file.

All objects active within the GIST server maintain a reference to the *gistServer.Server* object and can call its *log* method, again using as parameters some text and a log level, which in turn performs this on all of its logging objects. Combined with the string generation functionality of message objects and state machines described in sections 3.5.1 and 3.6, this provides for a powerful and extensible logging framework which gives a clear picture of what the program is doing and allows for extensive debugging, helping to fulfil the requirements of section 1.2. An example of logging output of the most detailed level can be found in appendix A.

3.8.2. ICMP Monitoring

The function of intercepting ICMP replies and its rationale is described in section 2.2.1.4. Every time a query is sent by a *gistQuerySM.QuerySM*, it stores this in the *gistServer.Server.icmpdict* dictionary object, referencing itself. The service class *gistICMP.ICMPService* monitors incoming ICMP messages and only selects those with the code that represents *ICMP Port Unreachable* [36]. Because this ICMP reply possibly contains only part of the UDP payload of the original datagram that caused the reply, but it does contain the full original UDP header, Query messages need to be matched to the dictionary by a distinguishing feature from the header. This is done using the UDP checksum from the original Query message, which should provide sufficient uniqueness for different Query messages and is short enough to be used as an index. Even for retransmitted Query messages with the same content, the Query Cookie object needs to be newly generated, providing a different checksum for the whole message. Once the matching has taken place using the *gistServer.Server.icmpdict* object, the *gistICMP.ICMPService* object has found the Querying state machine that issued the Query message. It then kills the *gistQuerySM.QuerySM* object in question and informs the NSLP application that it is the last NSIS capable node on the path with a *MessageStatus* API call. If, on the other hand, a valid response is received in the *gistQuerySM.QuerySM* state machine, it will clean up the entry it made earlier in *gistServer.Server.icmpdict*.

Note that the reception of ICMP messages only works if a Query message is sent with as source IP address the IP address of the sending node. This will only occur as a re-try after an initial query with as source IP address the MRI source has timed out.

3.8.3. Source Identification Information Handle

The Source Identification Information is a parameter that needs to be passed to the NSLP application when issuing a *RecvMessage* service primitive. With this Source Identification Information (SII), which is fully opaque to the application, it can perform

direct addressing when issuing a *SendMessage* service primitive, bypassing routing state withing GIST. This is usefull for example in cases of routing changes, when the application wishes to send message to the node on the old routing path after the routing state within GIST has changed. The SII needs to contain all information on how to reach a node without consulting routing state. In this implementation of gist this is simply done by creating a byte-string with all the relevant information. It is represented by a *gistAPI.SII* object, which contains the following attributes that are normally stored within the routing state:

Transfer Attributes This contains the transfer attributes of the type of connection stored in this SII, i.e. security and/or reliability.

Peer Identity The unique peer identity string from the NLI object.

Peer Address The IP address of the peer in question, also from the NLI object.

Peer Port The UDP port on which the peer accepts incoming Datagram mode messages, obtained during peer discovery.

Protocol Stack If the transfer attributes have at least reliability, this contains the protocol stack used by the MA to this node. The GIST node can use this to look up the MA state machine to this node, which is independent from routing state, and use it for transmission.

The *gistAPI.SII* object also contains a *create*, *encode* and *decode* method, analogous to the message objects from section 3.5.1.

3.8.4. Network Interface Management

The GIST implementation also contains a module with functions that performs low-level interaction with the kernel to gather information about network interfaces and their addresses:

gistIface.get_ip This function takes as parameter the interface name, e.g. *eth0*, and returns the IP address associated with it.

gistIface.get_iface This performs exactly the opposite operation of the last function, i.e. it takes as argument a IP address and returns the interface name to which this address belongs.

gistIface.get_all_ips This will return a list of all IP addresses listed for the network interfaces of this host. It is mainly used to check if this node matches the MRI destination address and thus is the last node on the path.

gistIface.get_if_route This function takes as argument a IP address and checks the host's routing table to see what network interface would be used to transmit a package to this address.

gistIface.is_endhost This takes as argument a MRI object and uses the aforementioned *gistIface.get_all_ips* function to determine if this node is the end node.

gistIface.get_nli_address This function also take a MRI object as argument and uses a number of the other functions in this module to select on of the host's IP addresses that is sure to be on the path of the MRI. This address can then be included in a NLI that includes this MRI.

These functions are used from various locations in the program.

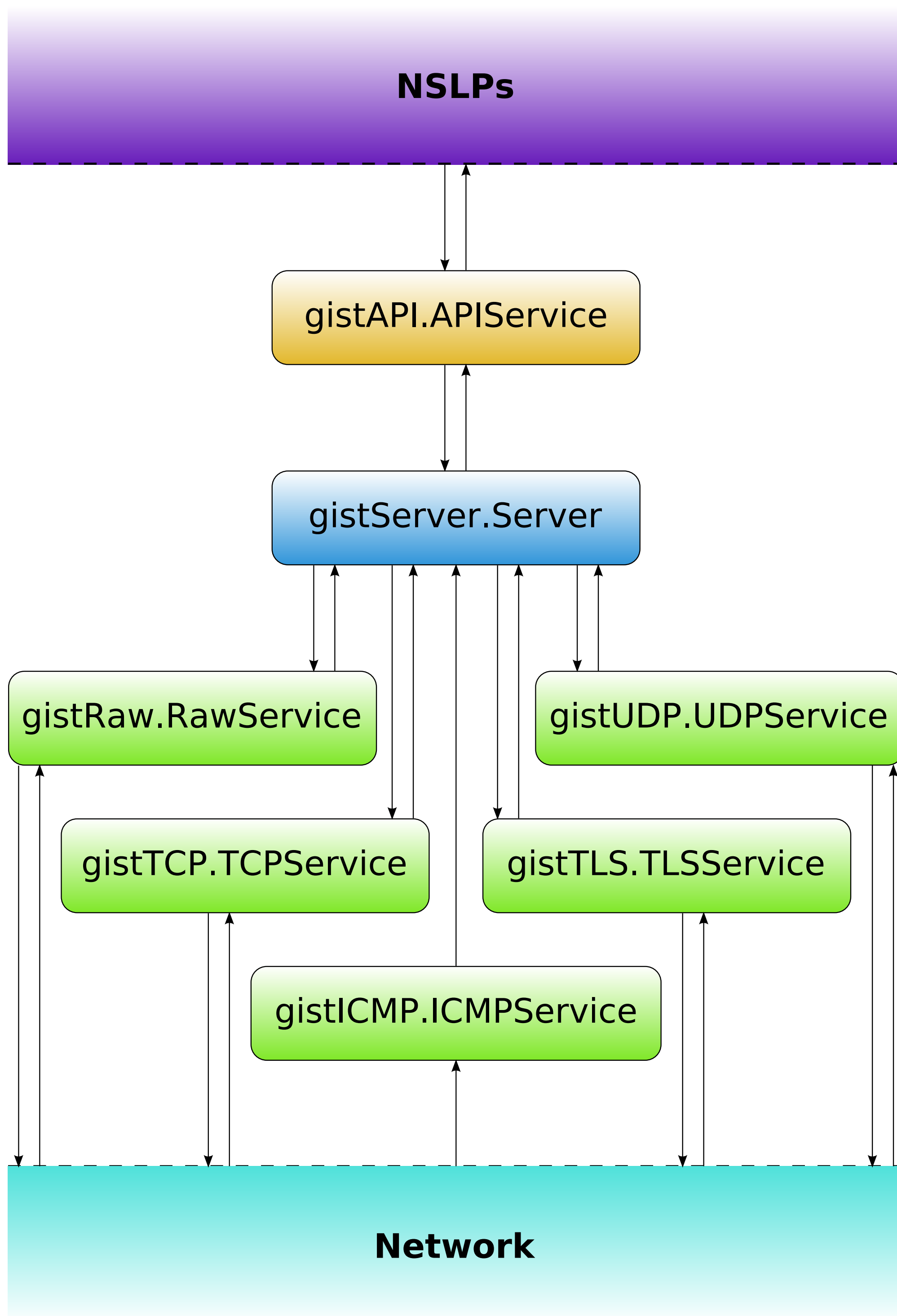

Figure 3.2.: High-level program structure of GIST implementation.

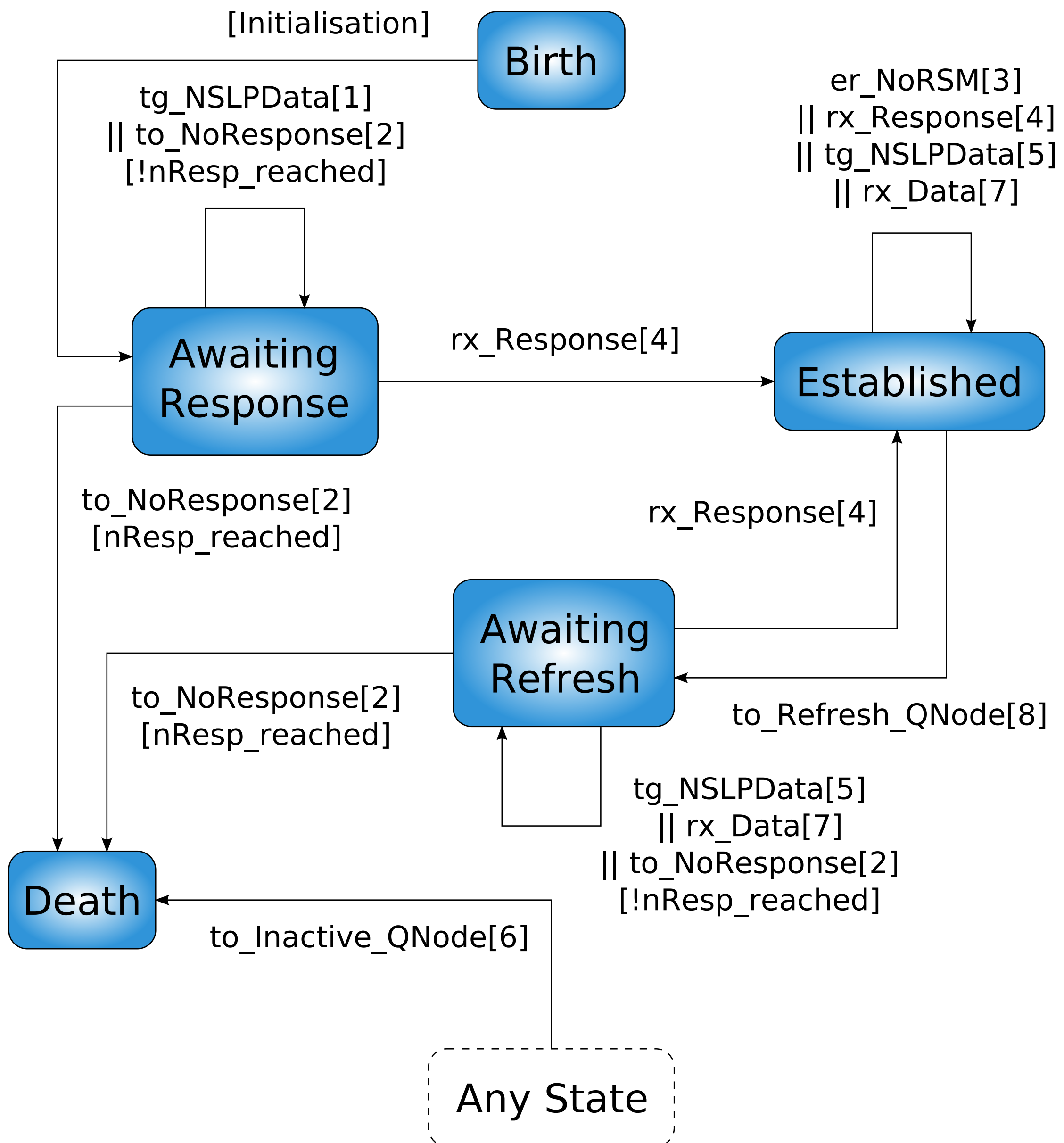

Figure 3.3.: Query State Machine.

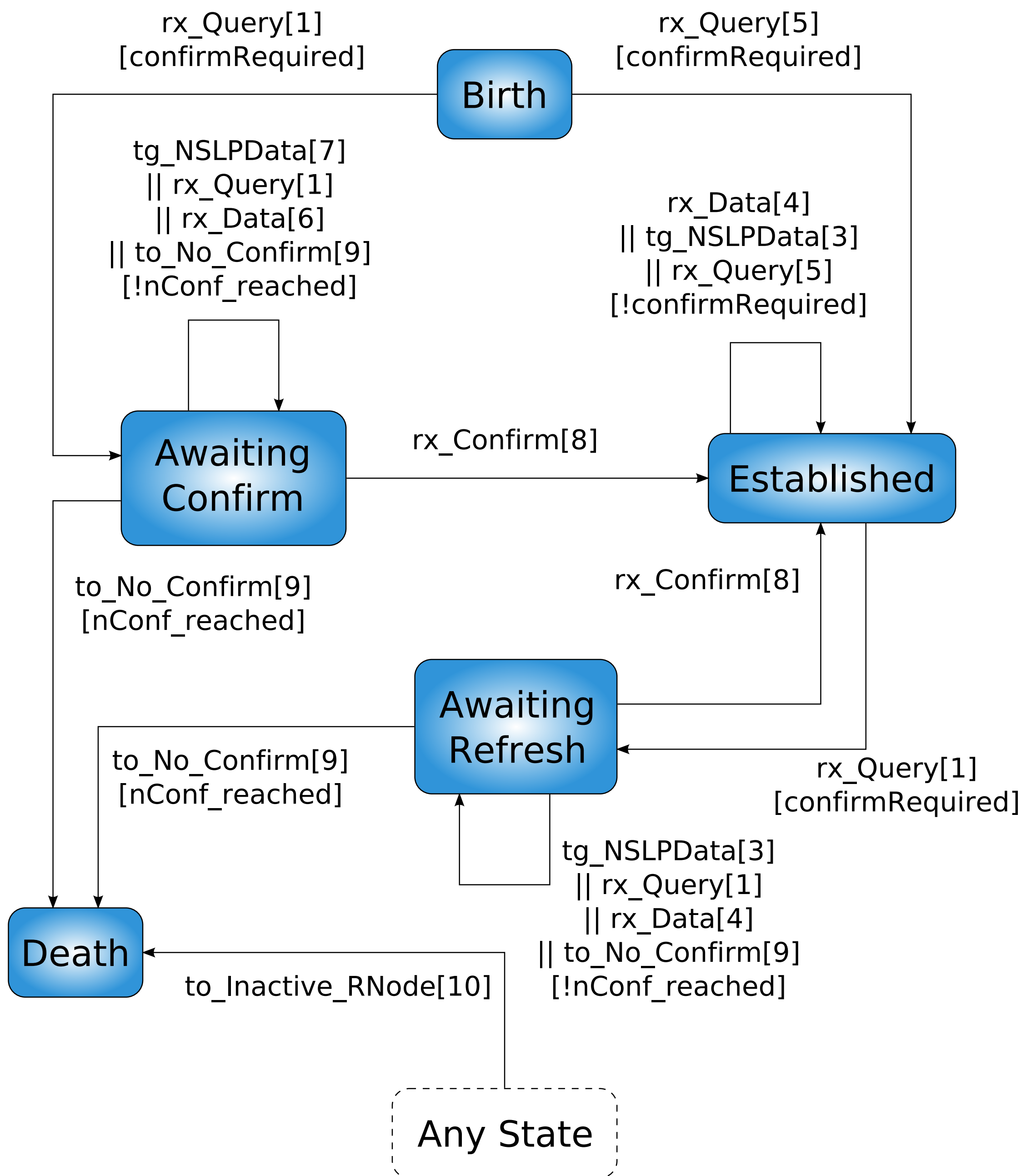

Figure 3.4.: Responder State Machine.

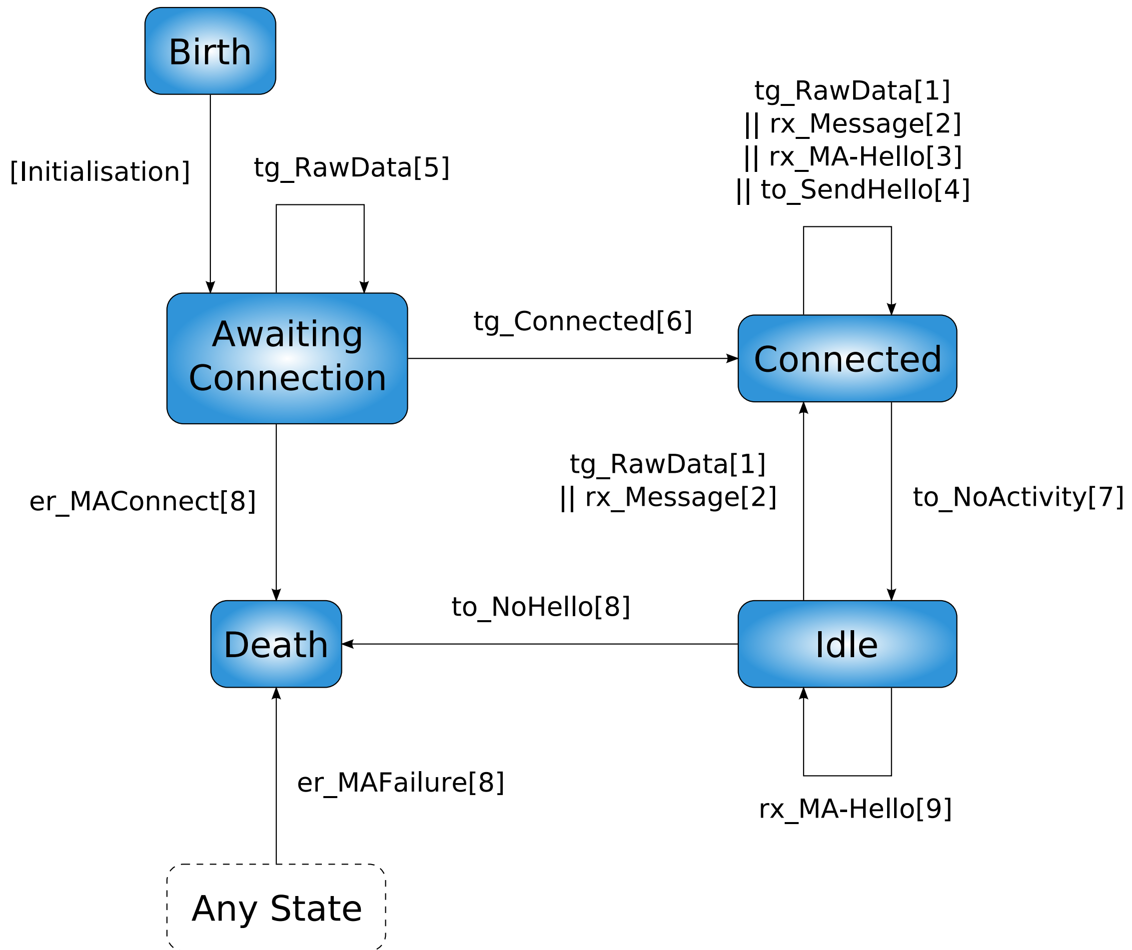

Figure 3.5.: Message Association State Machine.

4. QoS-NSLP Design and Implementation

This chapter provides a description of what was implemented at the QoS-NSLP of NSIS level within the scope of this assignment and how it was implemented, describing implementation structure and where needed motivating design choices. The implementation discussion can be subdivided into two parts, the first about the QoS-NSLP with IntServ Controlled Load (ISCL) QoSM and the second about RMD traffic control, as these are entirely separate implementations. Again, before this is done there will be a brief description of the work already performed by other people.

4.1. Previous Work

In conjunction with the GIST implementation assignment performed by Mayi Zoumaro-Djayoon described in the previous chapter, a RMD QoS-NSLP implementation was done by Martijn Swanink [42], also at the DACS group of the UT. Initially the intention was to update and expand this implementation to support the goals set out in section 1.1, as was the intention for the GIST implementation. For this reason, when this was changed to a re-implementation of the GIST layer, the choice was made to implement the same API of the previous implementation in the *gistAPI.APIService* class described in section 3.3. The bit-level format of this API can be found in appendices in the reports of both implementers [42, 44]. This allowed the new implementation of GIST to communicate with the RMD QoS-NSLP implementation. As this assignment and the master assignment covering the RMD implementation overlapped for a period of time, this was mutually beneficial, allowing testing of the new GIST implementation with the RMD QoS-NSLP on one hand and use of an improved and more reliable GIST implementation while debugging the RMD implementation. This is described in section 5.2.

However, as the GIST implementation was finished, it became clear that for the purposes outlined in chapter 1.1, it would be more beneficial to focus on implementing a more general QoS-NSLP incorporating the ISCL QoSM. The rationale behind this is the following:

- The RMD QoS-NSLP implementation focused wholly on the RMD QoSM. All QoS-NSLP applications need to adhere to behaviour outlined in [33], but because the RMD-QoSM has some quite radical differences the implementation only contained a minimum of QoS-NSLP functionality. The implementation in question served more as a proof of concept of the mechanisms used in the RMD QoSM than a fully fledged QoS-NSLP implementation. However, in performing end-to-end reservations by QoS aware multimedia applications, one also needs an end-to-end

per-flow QoSM, which RMD is not. For these reasons it makes much more sense to implement a generalised QoS-NSLP with the lightweight ISCL QoSM.

- The RMD implementation had only limited functionality and adhered to outdated specification drafts. Initially this was an argument to update and extend the implementation, but in light of the previous argument it makes more sense to first build a generalised QoS-NSLP for this assignment and construct it in such a way it can dynamically handle different QoSMs, including RMD. In this way the concepts of the RMD implementation can be integrated and transposed into the general QoS-NSLP implementation at a later time, whilst updating and extending the functionality. This also fits in with the modularity and extensibility requirement of section 1.2. The alternative, implementing a general QoS-NSLP into the RMD-specific implementation, would be illogical and overall bad programming practice, quite akin to fully re-implementing the whole program, at least in time needed to do this.
- The programming paradigm and concepts learnt during the GIST re-implementation could prove very usefull in the QoS-NSLP implementation. Although they are conceptually two separate entities that communicate through a common API, much of what was used for GIST could also be used for QoS-NSLP. Indeed, one could say that the preliminary phase of the QoS-NSLP implementation was already performed during the GIST implementation. Additionally, the GIST implementation boasted some features that could be advantageous to implement in a QoS-NSLP implementation with any QoSM, such as extensive logging, thread safety and of course the advantages of the Python programming language as demonstrated in section 3.2.

As there was still some co-operation with regard to RMD however, this report will describe some minor contributions made to this QoSM, in particular the part concerning the Linux Traffic Control subsystem, including Severe Congestion handling.

4.2. QoS-NSLP and IntServ Controlled Load QoS Model Implementation

As this implementation integrates seamlessly into the GIST implementation discussed in the last chapter, the implementation environment is exactly the same. For a discussion of this see section 3.2.

Due to time constraints the implementation discussed here is limited in its functionality. For now it will only support setting up new sender-initiated reservations, refreshing them and tearing them down, meaning this implementation precludes more advanced features such as receiver-initiated and bidirectional reservations, session binding and re-routing detection other than the one provided by GIST. However, the structure of the implementation allows for easy integration of these features at a later time, as per requirement. Also, the only QoSM implemented is that of IntServ Controlled Load.

This basic functionality provided by this implementation should however suffice for the purposes set out within the scope of this assignment, i.e. supporting reservations for QoS aware multimedia applications.

4.2.1. Program Structure

For a conceptual overview of the components within the QoS-NSLP application, please refer to Figure 2.5. To allow for modular use of different QoS Models, the implementation adheres to this model as much as possible, separating the general QoS-NSLP message handling from the QoSM specific functionalities, which are mostly those dealing with QSPEC processing and traffic control interaction. This is illustrated in Figure 4.1. The message handling component is represented by the *qosServer.QOSServer* class, while the Resource Management Function component can be represented by any class in the *qosRMF* module, e.g. qosRMF.ISCL for the ISCL QoSM. Which QoSM is used depends on configuration options set in *qosOptions*, which allow specifying the RMF per outgoing network interface. As an example of the use of the application API, a completely separate *qosConsole* module is a small console based application that can be used to instruct the QoS-NSLP to make and remove reservations. This is mostly used for testing purposes. Finally, the application API was also implemented in VLC [15], a media playback and streaming application. Section 4.2.3 will provide the bit-level syntax for this application API and section 4.2.3.1 will provide a discussion about the use of this API in VLC.

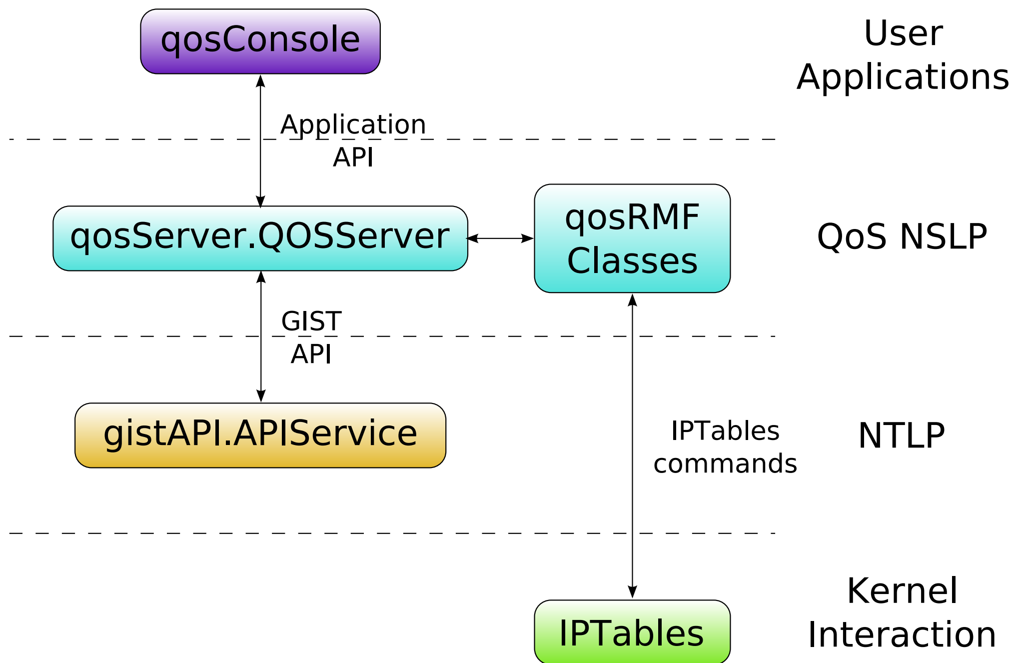

Figure 4.1.: The components of the QoS-NSLP implementation as applied to Figure 2.5.

As already mentioned, the QoS-NSLP implementation integrates seamlessly into the

GIST implementation. This is done by having the *qosServer.QOSServer* class operate as a service class of GIST, as described in section 3.3. This service class is started from the *gistAPI.APIService* class and anytime this class receives a API call from GIST with a NSLP-ID relevant to the QoS-NSLP, it will relay this API call to the *qosServer.QOSServer* class. From this it follows that the main methods of *qosServer.QOSServer* are the *RecvMessage*, *MessageStatus* and *NetworkNotification* incoming GIST API calls, the bulk of the processing of course being performed in the *RecvMessage* method, where QoS-NSLP messages are decoded and dealt with appropriately. Conversely, the *qosServer.QOSServer* has its own thread running in which it handles incoming application API calls from locally running applications that request QoS and, after processing, translates this into outgoing GIST API calls that it performs directly on the *gistServer.Server* object. Note that, like any GIST service thread, before initiating any action from its running thread as a result of an application API call, it must obtain the GIST threading lock as described in section 3.3.1.

The *qosServer.QOSServer* object also stores QoS-NSLP persistent session state, indexed per SID and represented by instances of the *qosState.PersistentState* class. This state contains information about sequence several up and downstream sequence numbers, both RSN and RII, the session's MRI and SII handle, reservations performed and also timers governing state expiry and refreshes. The RSN is represented by the special helper class *qosRSN.RSN*, which implements Request For Comment (RFC) 1982 [27] behaviour. As in GIST, these timers run in their own threads and obtain the threading lock before changing stored state.

For each QoSM that the QoS-NSLP node is configured to use, through options set in *qosOptions*, it maintains one instance of the relevant class in the *qoSRMF* module. These classes may maintain their own QoSM specific state and are required to support a common interface, i.e. they must implement the same methods for the *qosServer.QOSServer* instance to call. When any of these calls result in the installation or removal of reservation state, the relevant RMF class may interact with the Linux kernel through a *IPTables* [9] command, performed by the helper module *qosIPT*, which governs *IPTables* interaction. A detailed description of this is given in section 4.2.4. For the partial implementation currently in place, the interface methods are the following:

generate_reserve This instructs the RMF to construct an initiating reserve message, with objects and a QSPEC that is relevant to its QoSM. This method is generally called as a result of an incoming application API call.

process_reserve This is called when a RESERVE message is received. Depending on the QSPEC processing, this method can return an updated QSPEC to be propagated further downstream.

approve_response This method is called by *qosServer.QOSServer* at the node that initiated the RESERVE message upon reception of a RESPONSE message indicating success. This allows the RMF to inspect the QSPEC returned in the RESPONSE message and either approve or disapprove it. Upon disapproval a tearing RESERVE is sent.

generate_refresh This instructs the RMF to construct a refreshing reserve message for a particular session, taking into account information gathered about reduced refresh support of the downstream node. Because refreshes are generated asynchronously, i.e. separately by each node in the flow path, this method is called when a timer expires within the persistent state in each node.

generate_tear As the name suggests, this will have the RMF construct a tearing reserve for a particular session. This can be called either as a result of application API interaction or following an error condition.

remove_reservation This can be used to trigger actual reservation removal within the RMF upon receipt of a RESPONSE message indicating a successful teardown operation.

expire This is simply the method that is called from the persistent state timer as it expires. The RMF should take this as an indication that any reservation associated with this state should be removed.

Note that in a full implementation a number of interface methods may need to be added.

As the ISCL QoSM [29] adds little additional requirements, the *qosRMF.ISCL* class implements these methods in only a simple way, constructing a message with a QSPEC that is quite straightforward and only contains a Token Bucket parameter. For QoSMs such as RMD this is expected to be much more complex.

4.2.2. Message and Exception Processing

Analogous to the message decoding/encoding and exception handling described of GIST as described in section 3.5, the QoS-NSLP implementation has a *qosMsg* and a *qosException* module to perform this. The *qosMsg* module implements message decoding and encoding in exactly the same way as the *gistMsg* module, using an message objects in a tree structure. Please refer to sections 3.5.1 and 3.5.2 for a detailed description. As the QoS-NSLP specification [33] dictates error classes and codes in a similar fashion to those in GIST, again the concept of mapping each error code onto their own error class, derived from a base error class, can be used, as can be found in section 3.5.3.

The only location where message decoding takes place is at the start of the *RecvMessage* method of the *qosServer.QOSServer* class. Any exception caused by the decoding or subsequent processing of the message is caught at the end of this method, causing a RESPONSE method with a INFO_SPEC object relevant to the exception to be sent in reply to the message.

4.2.3. Application API

To enable user applications to request QoS from the QoS-NSLP application running locally and thereby achieve one of the main goal as described in section 1.1, an application API needs to be defined and implemented. Since this API is beyond the scope of the QoS-NSLP specifications [33], the syntax and semantics of it are implementation specific. The

design and bit-level syntax for the application API of this implementation is described in this chapter.

Note that the design of this API has a lot of similarities to the proposed RSVP API [12], notably the use of type-length-value byte-string encoded objects. The choice was made however to use new messages and objects, as those used for the RSVP API are very specific to that protocol. The proposed RSVP API also seems overly elaborate for the purposes of this thesis.

As the API will be designed from scratch, some requirements will need to be specified:

- The application API should be flexible in the amount of parameters that can be transmitted. Sometimes an application wants to specify more about a reservation that other times.
- It will need to be extensible, i.e. in future versions of the implementation it should be able to support new messages and parameters.
- The QoS-NSLP will need to be able to communicate with several user applications at once.

The application API is implemented as byte-string commands carried in UDP [34] datagrams, each command message possibly carrying parameters, either from the application towards the QoS-NSLP or vice versa. As the applications requesting QoS will logically reside on the same host, these UDP datagrams will travel over the loopback network interface of this host. The application will send the datagrams to the QoS-NSLP at a predetermined destination port and a semi-random source port, which will reply with datagrams that have the source and destination ports reversed. This allows the QoS-NSLP to communicate with several applications on the host, each using a different port number. UDP was chosen because it facilitates easy interchange of short delimited messages. A stream-oriented protocol such as TCP does not support message delimitation, which as we will illustrate is actually used in the message syntax. Alternatively, Unix domain sockets could be used, but these would have taken a little more effort to implement.

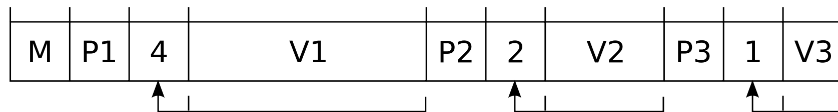

Figure 4.2.: Example of a QoS-NSLP application API message with three parameters.

Every UDP datagram contains exactly one message, optionally followed by a number of parameters. The message type is encoded as a single unsigned byte value. The possible message values are listed in table 4.1. Note that because one byte is used to specify messages, values of up to 255 may be used in a future implementation, supporting the extensibility requirement. If there are any parameters in the message, the unsigned byte type value of the first parameter is located directly after the message type byte. The values of these bytes can be found in table 4.2, which can again be extended in future.

Message	Value
Kill	0
List	1
Add	2
Del	3
Success	4
Failure	5
Item	6
End	7

Table 4.1.: QoS-NSLP Application API messages and their byte values.

Parameter	Value
SID	0
Source IP	1
Source Mask	2
Source Port	3
Destination IP	4
Destination Mask	5
Destination Port	6
Protocol	7
DSCP	8
Bandwidth	9

Table 4.2.: QoS-NSLP Application API parameters and their byte values.

In order to be able to skip parameters that the decoding side of the message does not recognise, because they were defined later, the byte after each parameter type value is its length. Because of this end decoding the message knows the size of each parameter as well as the total message size, so the order and number of parameters are irrelevant. Directly following the parameter size byte is the value of the parameter, which has a variable length depending on the parameter type. The byte-length and format of the parameters is described in table 4.3. This table also describes which parameters must, can and cannot be used in which messages. As can be seen in this table, some of the parameters are optional, supporting the flexibility requirement. Messages not listed in this table can never contain any parameters. Figure 4.2 illustrates a theoretical example of a message containing three parameters, the first four bytes long, the second two and the last one.

Some of the messages listed in table 4.1 are meant to be sent from the application towards the QoS-NSLP, others are sent in the opposite direction in response. These are the messages of the first category and the response messages they should elicit:

Kill This message instructs the QoS-NSLP to shutdown and discontinue operation. It takes no parameters and generates no response besides program termination.

Parameter	Length	Format	Add	Del	Item
SID	15	string	Not possible	Mandatory	Mandatory
Source IP	4	unsigned int	Mandatory	Not possible	Mandatory
Source Mask	1	unsigned int	Optional	Not possible	Optional
Source Port	2	unsigned int	Optional	Not possible	Optional
Destination IP	4	unsigned int	Mandatory	Not possible	Mandatory
Destination Mask	1	unsigned int	Optional	Not possible	Optional
Destination Port	2	unsigned int	Optional	Not possible	Optional
Protocol	1	unsigned int	Optional	Not possible	Optional
DSCP	1	unsigned int	Optional	Not possible	Optional
Bandwidth	4	float	Mandatory	Not possible	Mandatory

Table 4.3.: Use of QoS-NSLP Application API parameters in combination with messages.

List This message asks the QoS-NSLP to give a list of all reservations currently in place that were initiated by this node. For every reservation the QoS-NSLP generates a *Item* message, including as parameters the details of the reservation state. For an overview of which parameters are possible and mandatory, see table 4.3. The application will receive and parse these items, each being received in a separate UDP datagram. The QoS-NSLP will indicate the end of the list by sending a *End* message, which contains no parameters. After this the application stops waiting for incoming messages.

Add This instructs the QoS-NSLP to install a new reservation and initiate signalling for it. Again, table 4.3 lists the possible and mandatory parameters for this message. The QoS-NSLP send either a *Failure* or *Success* message in response, neither of which take any parameters. If the response is a *Success* message, a SID parameter needs to be included to indicate to the user application what SID the reservation state was stored under. This allows the application to later remove it.

Del This instructs the QoS-NSLP to remove state for a reservation it initiated and start an explicit teardown procedure for it. It takes as its only argument the SID of the relevant session, as listed in table 4.3. The QoS-NSLP send either a *Failure* or *Success* message in response, neither of which take any parameters. The *Failure* message is typically sent when the QoS-NSLP state does not contain a session it initiated with that SID.

4.2.3.1. VLC Application API Implementation

VLC is an open-source media playback and streaming application, supporting decoding of a large number of network protocols and video, audio and container formats. It can be used both to transmit and receive streaming video. To achieve the main goal set in the introduction in chapter 1, use of the application API described in the previous section was implemented in the UDP output module of VLC. As UDP is inherently

unreliable, it is an ideal candidate for a transport protocol that should receive QoS. The VLC modification allows specifying the IP address of the host on which the QoS-NSLP application is running, which will usually be the localhost, the UDP destination port on which to connect and the amount of bandwidth one wants to reserve. Note that this means that the peak bitrate of the media file being streamed has to be known in advanced. On initialisation of this module it will send a *Add* message to the QoS-NSLP at the specified address and port, including as parameters the specified bandwidth and the specifications of the flow, which it gathers automatically from other parameters of the program. If it does not receive *Success* as a reply, it will abort initialisation, generate an error message and prohibit sending the stream. On success the stream is sent and when the program terminates, a *Del* message is sent with as parameter the SID VLC received in the *Success* message.

4.2.4. Linux Traffic Control Subsystem

As described in section 2.2.3.1, packets belonging to a flow that is described in a QoS-NSLP reservation and is traversing a IntServ Controlled Load domain should receive "service closely equivalent to that provided to uncontrolled (best-effort) traffic under lightly loaded conditions" [43]. To provide this we have to implement the following behaviour in the packet scheduling of the outgoing interface of a node belonging to such a domain:

- When selecting packets to transmit, the packets receiving *Controlled Load* service should have priority over best effort packets.
- Each flow that receives *Controlled Load* service should experience as little interference from other flows receiving the same type of service.

To influence the behaviour of network interface packet scheduling in Linux one needs to use the traffic control services provided by the kernel [8]. Linux traffic control works with a concept called a Queueing Discipline, which governs the behaviour of packets arriving and leaving. Depending on the type of qdisc it can contain classes, which in turn can contain child Queueing Discipline (qdisc)s and so on. To satisfy the first of the two requirements outlined above, a classful priority qdisc needs to be used, which is illustrated in Figure 4.3. This qdisc, called PRIO in Linux, will always transmit *Controlled Load* packets first, which travel through the upper inner qdisc, when the interface is ready to send a packet. While best effort traffic uses a default PFIFO qdisc, which as the name suggests is just a First In First Out (FIFO) queue with a size limit in packets, the packets receiving *Controlled Load* service use the so-called Stochastic Fairness Queueing (SFQ) qdisc [40]. This is used to satisfy the second requirement, as SFQ gives packets from each flow their own queue, at least in most cases. Instead of indexing an unlimited number of queues, it hashes the flow identifiers and uses this as an index to a limited number of queues to save resources. This allows *Controlled Load* flows to experience a minimal amount of interference from eachother and, in combination with the PRIO qdisc, exhibits the behaviour of an unloaded or lightly loaded network from the point of view of the flow.

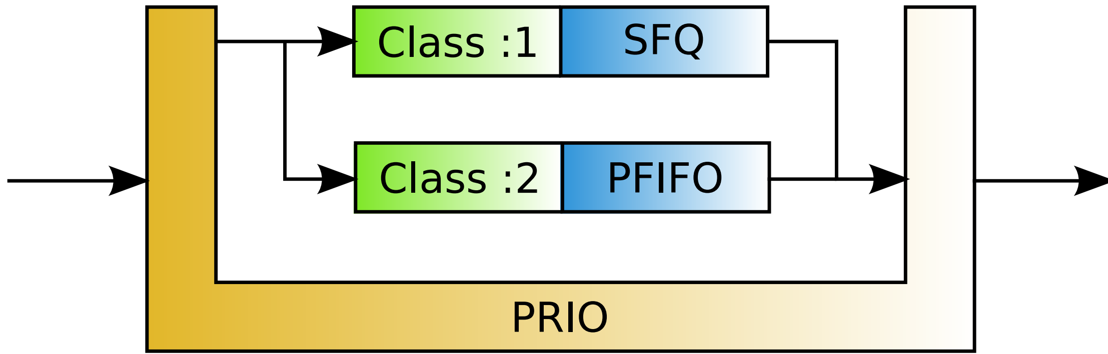

Figure 4.3.: Linux traffic control Queueing Disciplines used to implement IntServ Controlled Load.

The default class to send packets to in the *PRIO* qdisc is the one that has the PFIFO qdisc attached to it, i.e. the best effort queue. Filtering packets based on the MRI of a RESERVE message and instructing them to use the higher priority SFQ qdisc is done with *iptables* [9], using the CLASSIFY target. This target allows direct specification of the Linux traffic control class the packet should use. As packet filtering and classification can also take place in the Linux control system, it may seem illogical to use *iptables* for this purpose, but it has the advantage that filter construction is much easier compared to that of Linux traffic control, especially when filtering on multiple criteria, which may be needed if a MRI uses several options. The *qosRMF.ISCL* class uses the helper functions in *qosIPT* to convert the MRI of the flow in question to a *iptables* commandline and simply adds or removes this to or from the tables, specifying that packets belonging to that particular flow should traverse the SFQ qdisc and receive higher priority that best-effort packets.

4.3. RMD Linux Traffic Control

For the RMD QoSM, a linux traffic control framework similar to that of section 4.2.4 was implemented. To achieve the desired behaviour, outlined at the end of section 2.2.3.2, a new qdisc needed to be implemented. This will be described first.

4.3.1. The REMARKFIFO Queueing Discipline

To reiterate, to implement RMD severe congestion handling, a queue with the following properties is needed:

- At the end of every time interval, the total number of bytes that passed through the queue needs to be compared to the set thresholds. After this, the general state of the queue for the next time interval needs to be set, which can be either uncongested, within the congestion notification range or within the severe congestion range.

- Any time a packet leaves the queue an the state is set either to congestion notification or severe congestion, the packet needs to be marked accordingly and proportionately. Packets already marked may not be re-marked.
- Whenever a packets needs to be dropped because the buffers of the queue are full, unmarked packets are always dropped first.

These requirements were implemented in the REMARKFIFO qdisc as a Linux kernel module. As the name implies, it is based on the standard PFIFO and BFIFO modules already present in the kernel. Every time a packet arrives at this qdisc, the *remarkfifo_enqueue* function is called, which adds the packet size of the newly arrived packet to an internal byte counter. Although the external behaviour of this qdisc exhibits that of a single FIFO queue, internally two queues are used, already implemented by the kernel as linked lists. Upon arrival a packet is put at the tail end of either one of these queues, depending on if the packet is already marked or not. If adding the packet exceeds the byte limit of the overall virtual FIFO queue, either the packet is dropped instantly, or, if this packet is a marked packet, it removes the packet at the tail of the unmarked queue. This is exactly why the externally visible virtual queue is represented internally by two queues, to facilitate the dropping of unmarked packets. If a single queue was used, every time an unmarked packet needs to be dropped, the entire queue needs to be traversed in search for this packet. A graphic representation of the two internal queues and the externally observed virtual queue can be seen in Figure 4.4.

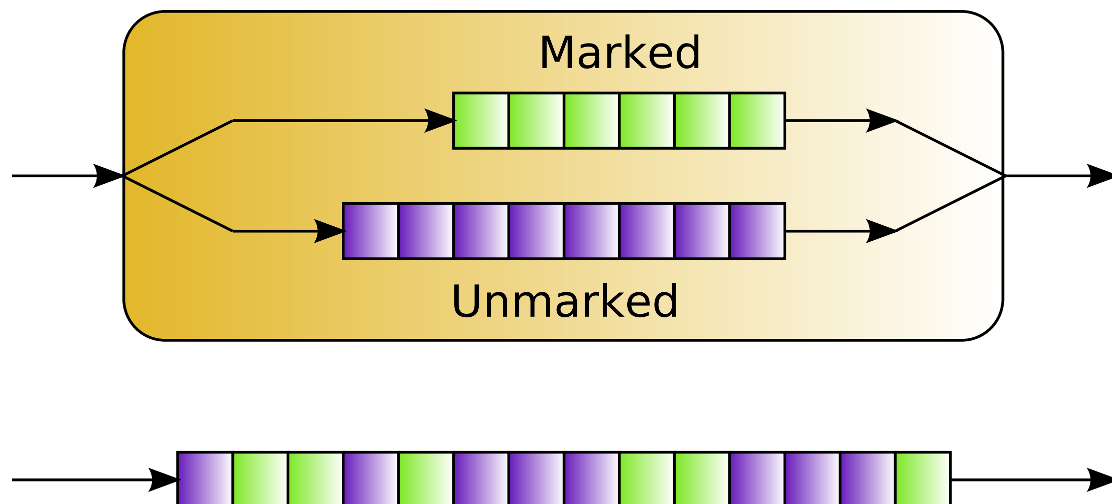

Figure 4.4.: Above the two internal queues of the REMARKFIFO queueing discipline is shown, one containing marked packets and the other containing unmarked packets. Below the externally observed virtual FIFO queue is show, with the marked and unmarked packets in their respective position in the queue, relative to the arrival timestamp it carries.

Whenever the system is ready to transmit a packet, it will call the *remarkfifo_dequeue* function on the REMARKFIFO qdisc. Note that this can be done either by the network interface that is ready to transmit a packet, or a classful parent qdisc, which may hold off calling this function to keep the data throughput within this PHB below a certain rate. In the latter case the incoming rate of the REMARKFIFO qdisc can still exceed this limiting rate, as the shaping is performed on the output of this qdisc. Within the *remarkfifo_dequeue* function two things happen. First a packet from one of the two internal queues is selected by means of the arrival timestamp associated with each of the two packets at the head of the queues, provided that both queues contain one or more packets. The packet with the lowest timestamp is selected and removed from its queue. This ensure that the external behaviour is that of a single virtual queue and that no packet re-ordering between marked and unmarked packets takes place. After this is done, the global state of the qdisc is checked and, if needed, the packet selected for transmission is marked. If this is done the REMARKFIFO qdisc decrements the byte counter for the number of bytes that need to be marked within this time period.

Its global state is updated every time interval by setting a kernel timer to call the *remarkfifo_update* function. As described in the requirements, this compares the byte counter that was incremented each time *remarkfifo_enqueue* was called to preset thresholds and sets the global state accordingly. It also sets the number of bytes that need to be remarked within the next time period, should this be needed, taking into account the number of bytes that were remarked in previous time intervals, using the sliding window mechanism that was described in section 2.2.3.2. It then also stores the resulting value in its sliding window history and resets the incoming byte counter.

To enable the use this new qdisc, the userland *tc* tool, part of the *iproute2* package [6], also needed to be modified. A module was written for this program to be able to instruct the kernel to use and configure it. The configuration parameters include:

limit Specifies the virtual FIFO size limit in bytes. By default the limit is equal to the MTU size.

interval Specifies the interval time in milliseconds. The default interval time is 50ms.

cellcount Specifies how many time intervals the sliding window mechanism should remember. The default value is 8.

multiplier Specifies the multiplier for proportional packet marking. This means that for N bytes over the preset limit, 1 byte leaving the qdisc is marked, where N is the multiplier. The default value is 2.

notified_dscp The DSCP value for packets marked as *notified*. Note that this are and all following DSCP value parameters are already left-shifted by two, taking into account the two least-significant ECN bits [38]. All DSCP value parameters are also mandatory and have no default value.

affected_dscp The DSCP value for packets marked as *affected*.

encoded_dscp The DSCP value for packets marked as *encoded*. Note that this can be the same value as *notified_dscp*.

cnd_rate Specifies the *congestion notification detection* threshold rate. If packets arrive at the qdisc at a rate that is above this rate, the global state is set to congestion notification. This and all following rates are mandatory parameters.

scd_rate Specifies the *severe congestion detection* threshold rate. If packets arrive at the qdisc at a rate that is above this rate, the global state is set to severe congestion. Note that this rate should always be higher than *cnd_rate*.

scr_rate Specifies the *severe congestion restoration* threshold rate. If the global state is set to severe congestion and packets arrive at the qdisc at a rate that is lower than this rate, the global state should be changed to no congestion. Note that this rate should always be smaller than or equal to *scd_rate*.

4.3.2. Linux Traffic Control Subsystem

Using the REMARKFIFO qdisc described in the previous section, a Linux traffic control framework was constructed, similar in nature to that implemented for the ISCL QoSM, described in section 4.2.4. Figure 4.5 displays the queueing disciplines used on any network interface on which outgoing packets traverse the DiffServ domain. As root qdisc Hierarchical Token Bucket (HTB) is used [4]. As the name would suggest, this qdisc allows the composition of a number of token buckets in a tree structure, with the leaf nodes of the tree representing different traffic classes. Each of these classes has its own token bucket, allowing a minimum guaranteed throughput on this interface to be specified. It also supports features such as sharing excess available bandwidth among sibling classes and priorities for both this excess free bandwidth sharing and packet scheduling. HTB is used to create three traffic classes, sharing the total link capacity specified in the parent class:

1. A traffic class is attributed to signalling traffic, i.e. NSIS communication. The specification dictates that signalling should always receive higher traffic priority [19] and creating a traffic class for this ensures that a certain amount of bandwidth is reserved for this. Assigning this class a higher priority in relation to the other two classes within HTB also ensures that signalling traffic receives excess available bandwidth first and gets dequeued first when the interface is ready to transmit a packet. This last property allows signalling traffic to be transmitted even in face of congestion. As there is no external method for distinguishing NSIS traffic, this traffic needs to be marked at the application layer with a predetermined DSCP. As child qdisc a simple BFIFO, a FIFO queue with a length limit in bytes, is chosen.

2. The only DiffServ PHB used within this project, Expedited Forwarding, has its own traffic class. If the RMD application uses the reservation-based admission control method, the minimum bandwidth specified for the token bucket of this traffic class should be taken as the maximum bandwidth that can be spent in reservations.

This ensures that traffic within this class does not exceed the total bandwidth assigned to it and should prevent EF traffic from experiencing congestion. Note that if there is excess free bandwidth on the interface that is not spent on signalling traffic, it is shared equally among this and the best effort class. The child qdisc for this class is of course the REMARKFIFO queue described in the previous section.

3. Best effort traffic is delegated to the last class, which again uses a simple BFIFO qdisc. As traffic within this class is not controlled by reservations, it may experience congestion. The minimum rates specified in the other traffic classes will ensure that only other best effort traffic is hindered by this congestion. This is configured as the default class for all traffic.

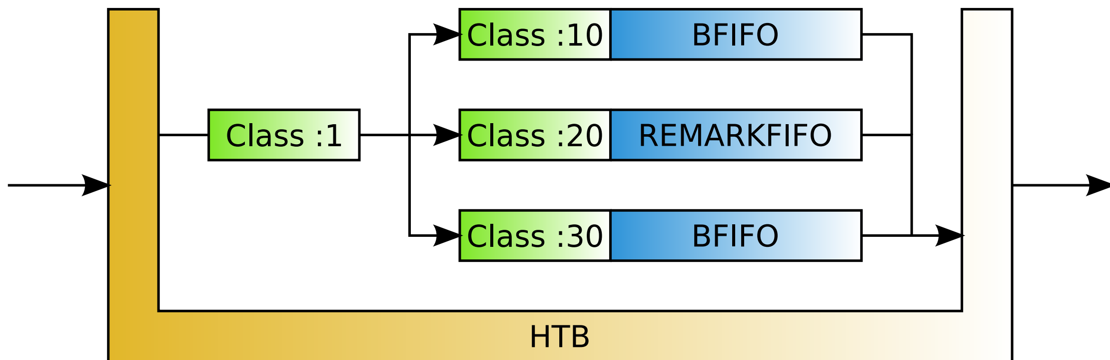

Figure 4.5.: Linux traffic control Queueing Disciplines used to implement RMD.

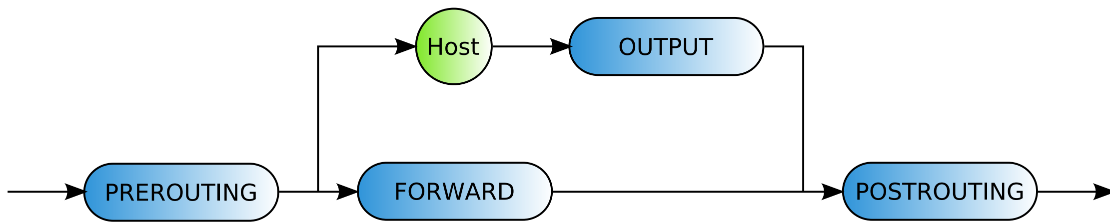

Figure 4.6.: Conceptual diagram of locations of some of the Linux netfilter tables.

For much of the same reasons as those outlined in section 4.2.4, classification of flows is performed in Linux *iptables*. An additional reason is that it the facilitates userspace-kernel interaction that is needed for keeping track of congestion marked DSCPs in the egress node, as ingress node, core node and egress node behaviour is combined in the *iptables* rules, including severe congestion handling. *iptables* uses a concept known as chains, which packets may traverse depending on their path through the kernel. Because the *iptables* setup is a little more complex as the one for the ISCL QoSM, a short description of the chains used may be in order. Note that, because of the nature of the operations performed on the packets, all chains are located within the *mangle* table of *iptables* [9]. Figure 4.6 displays those used chains and their conceptual location. Network

packets enter from the right, all packets passing through the PREROUTING chain. After this chain the destination of the packets are inspected, those that are destined for this host are sent up to the application layer, those that have their destination elsewhere and should be routed according to the system routing tables pass through the FORWARD chain. Packets that originate from this host first go through the OUTPUT chain, after which they join any other packet that leaves this host going through the POSTROUTING chain. Only after this packets are queued on the qdisc of the relevant network interface, so Figure 4.5 should be considered to be conceptually attached to the right end of Figure 4.6.

The rules and actions used in these chains can be respresented in the following pseudo-code:

```
PREROUTING:    if from external:
                  clear DSCP value                              (1)
               if a IPv4 option is present:
                  send the packet to the netlink queue          (2)
FORWARD:       if from external and to internal:
                  if this flow is reserved:
                     set DSCP value to EF                       (3)
               if from internal and to external:
                     go to the flow chain for this flow         (4)
OUTPUT:        if to internal:
                  clear DSCP value, except signalling DSCP      (5)
POSTROUTING:   if to internal:
                  if DSCP is signalling DSCP:
                     set tc class to :10                        (6)
                  if DSCP is in EF DSCPs:
                     set tc class to :20                        (6)
               if to external:
                  clear DSCP value                              (1)
flow chain:    if DSCP is notified DSCP
                  increase counter                              (7)
               if DSCP is affected DSCP
                  increase counter                              (7)
               if DSCP is encoded DSCP
                  increase counter                              (7)
```

An explanation of each of these actions follows:

1. DSCP values for packets entering and leaving the DiffServ domain should be cleared of their value to avoid interference with other domains that wish to use this field in the IP header.
2. This is the filter installed for GIST Query datagrams. It is described fully in section 3.4.1.

3. Because of the of this ruleset, this action will only occur at the ingress node. Note that, although it is listed as one rule in the pseudo-code, each reserved flow will have its own matching rule with a filter corresponding to the contents of the MRI.

4. This action will only occur at the egress node. Again, each reserved flow will be individually matched. Also, each reserved flow will need its own chain created upon reservation, providing the RMD application with a means to gather packet and byte-count for the different DSCP values per flow per time interval from the kernel.

5. Clear all DSCP values from locally generated packets, except the signalling DSCP, is an additional measure to make sure that traffic originating from the DiffServ domain does not interfere with the reservations made externally.

6. This where the actual classification takes place, packets with the signalling DSCP will go to the upper class pictured in Figure 4.5, packets with a EF DSCP or a marked EF DSCP will go to the middle EF class and any other unmatched packets will go to the lower best effort class.

7. As explained in action 4, each reserved flow will have to have its own chain created an assigned to it. In this chain the different marked DSCP values are matched upon. As *iptables* already does accounting for each rule in its ruleset, if configured in the kernel that is, this is simply implemented as a matching rule with no action. The RMD application should poll the kernel every time interval to read and reset these values and use them to calculate which flows should be terminated.

5. Functional Experiments

To determine if the implementation exhibited the desired behaviour, i.e. if it adheres to the specifications and to remove any bugs that may appear, a number of tests were performed. The setup used for performing these tests will be described first, after this there will be a brief discussion of early tests performed during implementation. The main section of this chapter will be comprised of the tests performed with the final implementation, which will evaluate if those goals set in chapter 1 that can be tested are reached, and finally there will be some discussion about interoperability testing.

5.1. User-Mode Linux Setup

While implementing both GIST and the different elements of the QoS-NSLP, a testing network with several routers was needed in order to perform several scenarios and test the implemented functionalities. However, because of resource limitations, it was not possible to have a real-life testing network. For this reason User-Mode Linux [14] was used to create five virtual hosts in a network setup on the same master host. User-Mode Linux (UML) allows just this, running a Linux Virtual Machine (VM) as a process on a Linux host system. Each of these VMs can have virtual network interfaces assigned to them that can either communicate with other VMs residing on the host system or with virtual network interfaces on the host system itself. The setup for this network and the location of the virtual hosts in it can be seen in Figure 5.1. Each host has one interface that connects it to a virtual Local Area Network (LAN) that is also connected to the host system. This allows each VM to communicate with the host system and, through NAT, reach the internet, which may be needed to perform system updates with the VM. The various interconnections between the virtual hosts form three separate LANs, and together these hosts and their connections for a chain of "routers", each host being configured to route traffic to the various domains through the relevant network interface. The names of the virtual hosts correspond to a RMD testing setup, in which the inner three hosts represent a DiffServ cloud, with *Edge1* and *Edge2* performing the role of the ingress and egress nodes. When testing for ISCL, all nodes can co-operate on an equal basis. Note that this setup was inherited from those who performed assignments previous to this one [42, 44], although it was adapted slightly.

5.2. Early Tests

During the implementation phase of GIST, testing had to be done with an already implemented NSLP application. The two NSLP implementations used were the *Ping*

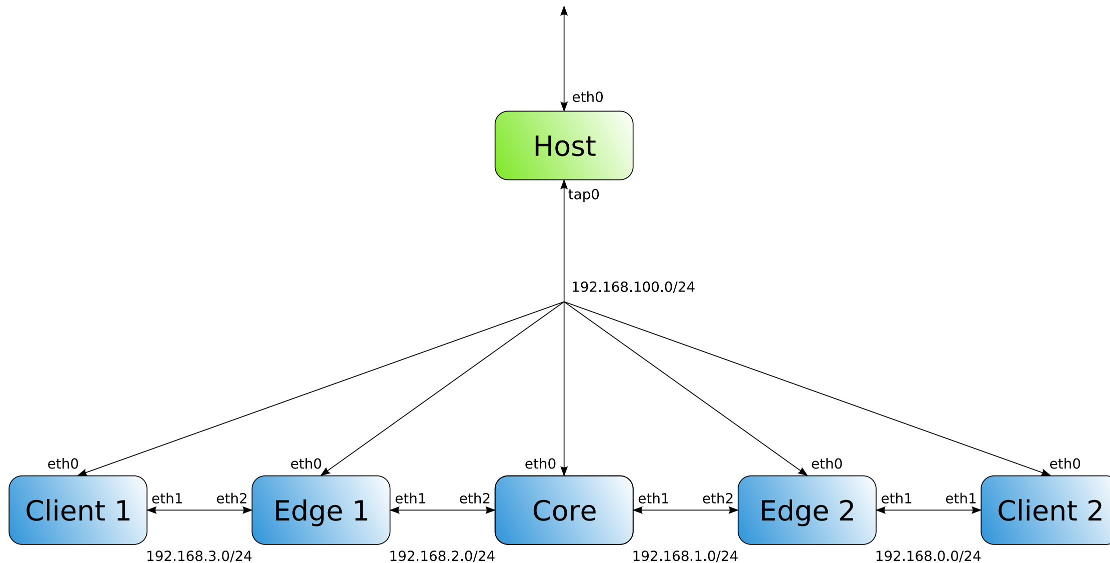

Figure 5.1.: Diagram of User-Mode Linux virtual machines in a testing network.

tool and the RMD implementation described in [42].

RMD Implementation

As mentioned in section 4.1, the GIST API was implemented to comply to the one specified in [42, 44] in order to allow the GIST implementation to seamlessly replace the already existing implementation in the cooperation with the RMD implementation. In this manner the functionality of the implementation could be tested, as it would need to exhibit the same behaviour as the previous GIST implementation [44], insofar as this functionality was present in this previous implementation. As the functionality of the RMD implementation was fairly linear, i.e. as already explained in section 4.1 it was very focused on providing certain parts of RMD functionality, the tests mainly consisted of making sure that GIST allowed this implementation to perform its tasks successfully. For more detailed scenarios, please refer to the revelant sections in the report on the RMD implementation [42]. As a result of these tests, adjustments were made both in the RMD and GIST implementations.

Ping Tool

Exactly for the purpose of testing GIST implementations, a *Ping* NSLP application was proposed [25] and implemented at the University of Göttingen. As all ping applications, the *Ping* NSLP can be used to measure the Round Trip Time (RTT) of a network, in this case a series of NSIS-aware hosts. A typical example is shown in Figure 5.2. A client signals the *Ping* NSLP that it wants to “ping” a certain host and which transfer attributes it wants to use. The NSLP then constructs a message and related MRI from itself to the designated host, including a timestamp in the message. The message then travels downstream towards the destination host, passing *Ping* NSLPs at other NSIS-aware hosts along the way that each record their own timestamps within the message.

By travelling downstream, routing state is setup between the GIST nodes. At the destination node the MRI is reversed and the message travels back upstream using the routing state established by the downstream message. Again, each *Ping* NSLP that is passed records its timestamp. When the message arrives back at the originating node, the NSLP at this node can report back to the client and calculate the timestamp differences.

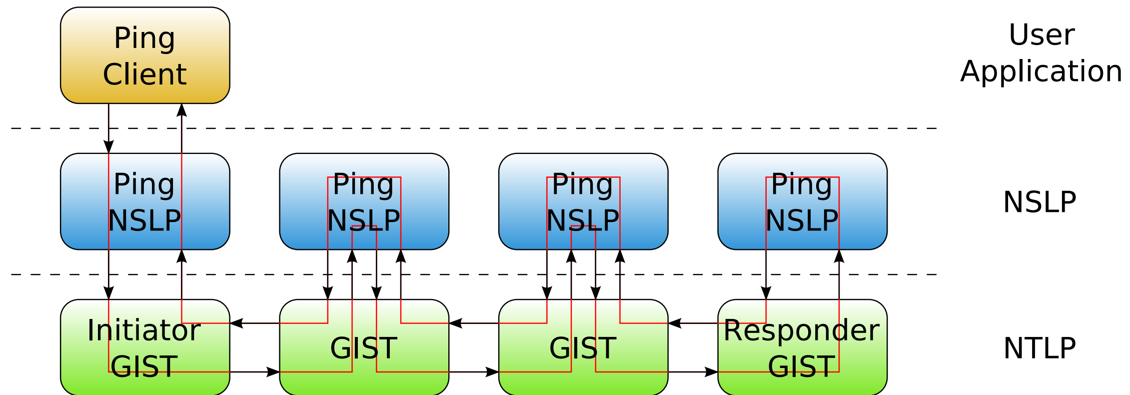

Figure 5.2.: Example message flow for the *Ping* NSLP.

An early implementation downloadable from the Göttingen University website was adapted to the API and used for testing. As this is a very simple NSLP protocol, it can easily be used to test basic GIST functionality in correctly setting up routing state using different protocols, depending on the transfer attributes. Typically the *Client1* node in the testing setup would perform the role of the initiating node and the *Client2* node the role of the responding node, traversing the whole virtual test network. In this way setting up routing state over UDP, TCP and TLS over TCP was tested. Other things that were tested was bypassing a node by not including the *Ping* NSLP-ID in the list of IDs of that node. During implementation this helped to verify if the functionality implemented behaved as expected.

5.3. Lab Testing

The aim of the functional experiments on the final implementation is to verify that the implementation functions in such a way that it can achieve the main goal, i.e. enabling QoS aware multimedia applications to perform dynamic reservations by using NSIS, and achieve it in such a way that it adheres to the requirements and criteria set in section 1.2. The main requirements that can be tested for are those that state that the implementation should follow the NSIS specifications and that it should gracefully handle error conditions. This can be done by performing a number of scenarios, both normal use scenarios and those with error conditions, and observing if the implementation exhibits the expected behaviour through the extensive logging facilities described in section 3.8.1 and in some cases actual preformed reservations. Although a number of scenarios are

described here, it is by no means an exhaustive list of verification tests, rather a few examples of tasks that the implementation should be able to perform correctly.

5.3.1. Successful Reservation and Teardown

The first scenario is that of a successful reservation through five ISCL QoS-NSLP nodes, similar to the example described in section 2.2.2.3. The QoS-NSLP level message exchange can be seen in Figure 5.6. *Client1* is instructed over the application API to perform a reservation of some flow from *Client1* to *Client2*, using the *qosConsole* application. As a result the logs show that *Client1* first performs the QoS reservation locally, registering it with the local traffic control subsystem through *iptables*. A RESERVE message is sent and propagated downstream towards the destination node *Client2*. As the RESERVE messages contain the Q flag, each node along the path directly sends a NOTIFY message in return indicating that it supports the reduced refresh mechanism, whereby refreshes are done with as only reference the RSN. Each node along the path checks if it has enough resources to perform the reservation, calls *iptables* to actually perform it and then propagates the RESERVE message. Once the RESERVE message reaches *Client2* this directly responds with a RESPONSE message indicating a successful reservation through the INFO_SPEC object. Note that in this implementation the destination node of a flow reservation does not store any QoS-NSLP state, as it does not have to maintain any reservations on an outgoing network interface. This message is propagated back upstream, using the GIST routing state now established, towards *Client1*. This has the RII it included in the initial RESERVE message stored and matched the RII included in the RESPONSE to it. After this, the reservation is fully established and refreshes take place asynchronously between the nodes. Note that this is not pictured in Figure 5.6.

Some time later the command is issued to the QoS-NSLP instance of *Client1* to remove the reservation, again using *qosConsole*. The logs indicate that a RESERVE message with the tear flag set is sent and propagated downstream. In case of an error during reservation removal, each node delays the actual removal of the reservation to the point in time where it receives a RESPONSE indicating a successful teardown. It does this by recording the RII used in the tearing RESERVE message. Once this message reaches *Client2*, it indicates this successful removal. Note that although *Client2* has no QoS-NSLP state installed, it still reports a successful teardown as default behaviour. The RESPONSE message being propagated upstream causes each node it reaches to remove state, until it reaches the originating node.

The behaviour as observed from the log files for each of the nodes conforms to the specifications, as was required.

5.3.2. Successful Reservation With NSIS-Unaware Nodes

This scenario is similar to the previous one without the explicit teardown, with two differences. The first is that the *Core* node is configured not to be interested in GIST messages that carry the NSLP-ID of the QoS-NSLP, through removing this value from

the NSLP-ID set in *gistOptions*. The second is that *Client2* doesn't have any GIST and QoS-NSLP application running at all. This situation is depicted in Figure 5.3. As the logs indicate, the *Core* node intercepts the initial GIST Query sent by *Edge1*, but sends it on to *Edge2* as it is not interested in the NSLP-ID included in the Query message. Subsequently, a routing state is established between *Edge1* and *Edge2* for this particular combination of MRI, NSLP-ID and SID.

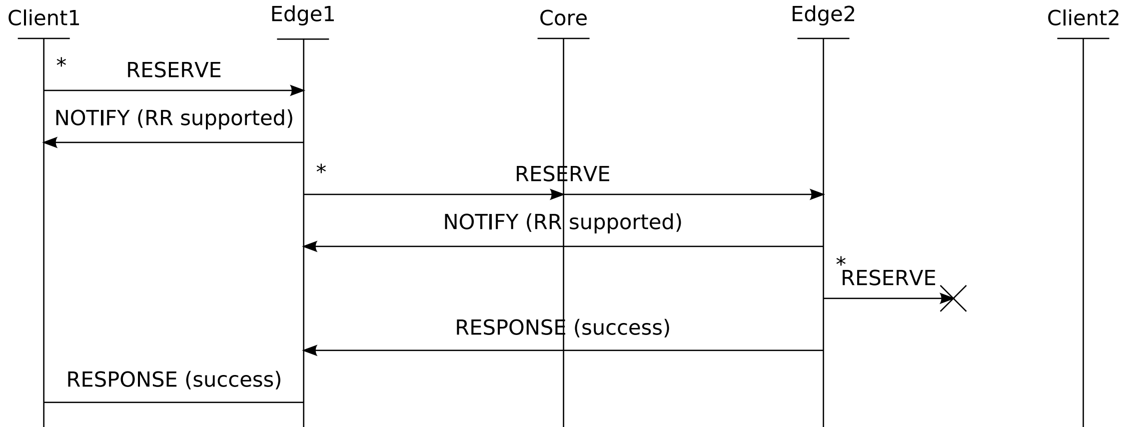

Figure 5.3.: QoS-NSLP level message diagram of a successful reservation with NSIS-unaware nodes. At the locations and times marked with a “*” the QoS-NSLP interacts with the kernel to actually add or remove a reservation.

Client2's system reports a ICMP “Port Unreachable” message in response to the GIST Query sent by *Edge2*, which catches that ICMP message and reports to its local QoS-NSLP instance that it is the last NSIS node on the path via the *MessageStatus* API call. In response to this *Edge2* initiates sending a RESPONSE message indicating a successful reservation.

Again, the behaviour exhibited is the desired behaviour and conforms to the specifications.

5.3.3. Unsuccessful Reservation

In this scenario, depicted in Figure 5.4, the *Core* node does not have sufficient bandwidth available to accept the reservation. This is achieved by setting the banwidth limit for ISCL quite low in *qosOptions* at the *Core* node an requesting more bandwidth than this at the *Client1* node using the *qosConsole* application. In the logs can be seen that the *Core* node sends a RESPONSE message back upstream indicating failure in reply to the RESERVE message requesting too many resources, eventually reaching the originator of the RESERVE message, *Client1*. At this point two nodes, *Client1* and *Edge1*, have a reservation in place that needs to be removed. To achieve this *Client1* sends an explicit teardown message in the form of a RESERVE message with the tear flag set. This operates as in the case for scenario 1, until it reaches the *Core* node, which has no reservation installed, nor does any node downstream from it. However,

the default behaviour in this case is to just forward the tearing RESERVE downstream until it reaches the destination node set in the MRI, *Client2*. Again, the default action to take when no state is installed is to reply that the teardown was successful in a RESPONSE message. As this travels upstream it eventually reaches the *Edge1* and *Client1* nodes, which use the RII to match it to the tearing RESERVE message and remove the reservation state.

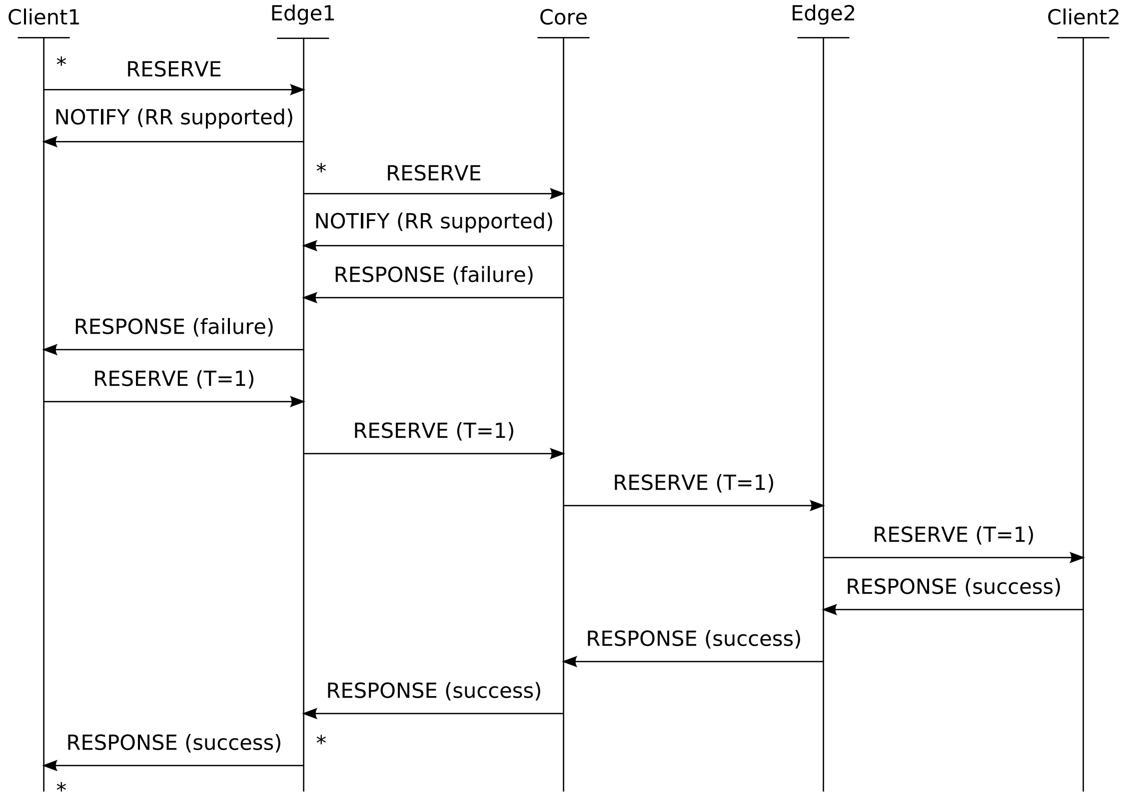

Figure 5.4.: QoS-NSLP level message diagram of an unsuccessful reservation. At the locations and times marked with a "*" the QoS-NSLP interacts with the kernel to actually add or remove a reservation.

The log files gained during this test show that this particular error condition can be handled gracefully.

5.3.4. Final Demonstration

As described in section 4.2.3.1, the developed application API was implemented in VLC. To provide a working demonstration of a QoS-aware multimedia application the setup depicted in Figure 5.5 was used. As the virtual UML hosts have only limited resources and can't access advanced video or audio hardware, VLC needs to run on the host system. Two instances of VLC are started, one for transmission of a video stream and one for reception of it. The transmitting VLC is configured to send the video stream

towards *Client2* and the routing tables of the host system are configured in such a way that it will use *Client1* to as a router for traffic towards the virtual subnet *Client2* is on. In this way the streaming video traffic, travelling in UDP datagrams, pass all the nodes as indicated by the arrow. *Client2* is configured to NAT this particular video stream back to the host system over the virtual interface it shares with it. In this way the stream can be decoded and shown by the receiving VLC instance, listening at a particular UDP port.

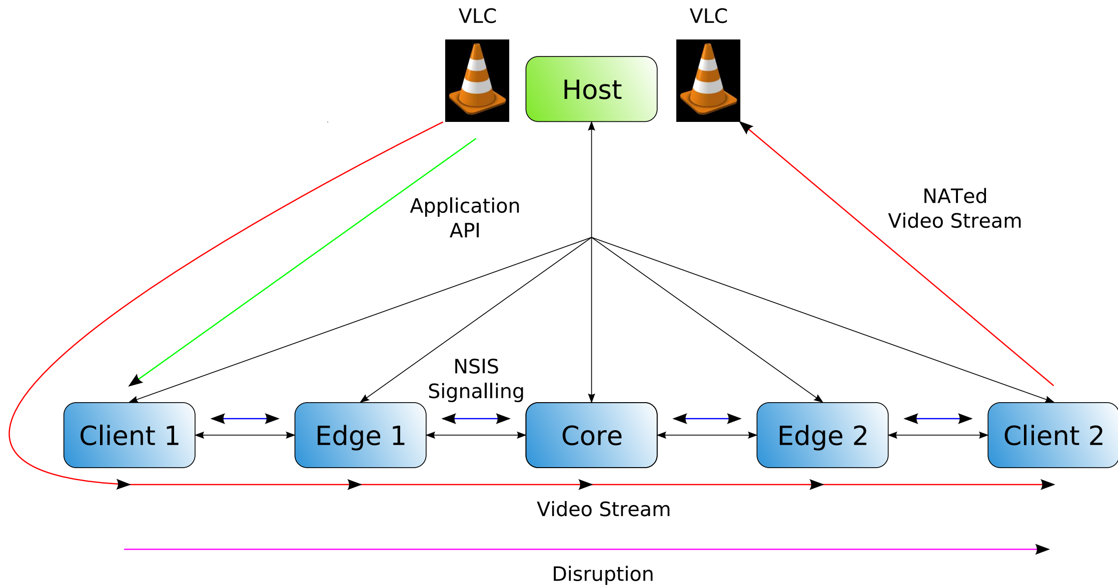

Figure 5.5.: The User-Mode Linux testing network as used for the final demonstration.

Instead of sending UDP datagrams on the loopback interface, the sending VLC instance running on the host system is configured to send application API message over its virtual network interface towards the QoS-NSLP instance running on *Client1*. Although this is not the way in which the API should normally be used, it is necessary for this virtual network test setup. The sending VLC will automatically request a reservation for the amount of bandwidth specified on the commandline for a flow with the MRI of the one it is about to transmit. When the program terminates, it sends the command that triggers an explicit reservation teardown over the API.

To test if the system works as a whole, a disruptive stream of UDP traffic is sent from *Client1* towards *Client2*. For this purpose the small traffic generating application *jtg* [7] is used. As each UML host has a maximum throughput speed on their virtual network interfaces, enough disruptive packets would cause the multimedia stream to experience packet loss. This can be shown by not instructing the sending VLC instance to perform the QoS reservation. In this case the video that is received is garbled. The test shows that this multimedia stream, through the reservation, is given Controlled-Load QoS by each of the nodes and any packets that are discarded by the nodes belong to the disruptive UDP stream.

5.4. Interoperability Testing

This GIST and QoS-NSLP implementation is only one of many. Implementations have been developed at several universities and companies. As they are developed according to the same protocol specifications, all of these implementations should be able to interoperate with one another, i.e. because the message encoding and processing behaviour is specified they should be able to communicate. Different implementations are regularly being tested in interoperability tests and the intention was to have this implementation participate in such tests as well. Unfortunately due to financial limitations it proved impossible to attend such a testing session. Some remote testing was done during a session, involving basic GIST handshakes, but the parties involved had very little time to attribute to it. Those tests that did take place revealed no errors or shortcomings in this implementation.

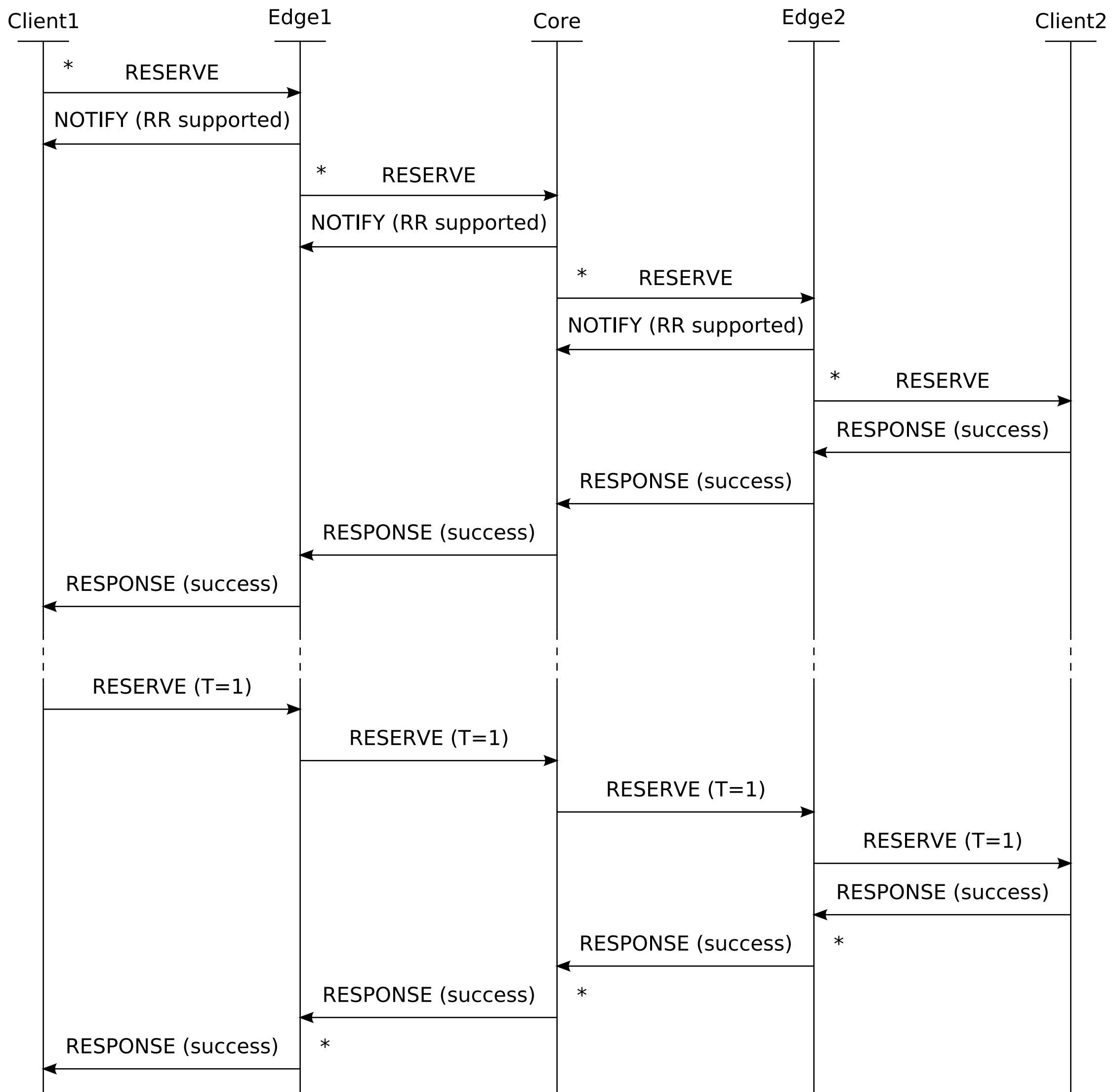

Figure 5.6.: QoS-NSLP level message diagram of a successful reservation and subsequent teardown. At the locations and times marked with a “*” the QoS-NSLP interacts with the kernel to actually add or remove a reservation.

6. Conclusion and Future Work

This thesis has attempted to provide a detailed description of the work involved in continuing development of an existing NSIS implementation in order to enable QoS aware multimedia applications to perform QoS reservations. The goals for this were set in chapter 1, an brief overview of the NSIS protocol suite was given in chapter 2, the design and structure of the implementation was described in chapters 3 and 4 and finally the functional experiments were discussed in chapter 5. This final chapter will evaluate the work performed within the scope of the master's assignment in terms of the set goals and reccommend future work that can be done.

6.1. Conclusions and evaluation

Referring to section 1.1, the achievement of the objectives will now be discussed one by one:

- Both the NSIS specifications and the already present implementations were studied. The set of NSIS specifications is very comprehensive and most specifications required several readings. In this light it may have been better to focus on one or two specifications in particular instead of the full spectrum of GIST [39], QoS-NSLP [33], QSPEC [18], RMD QoSM [19] and ISCL QoSM [29]. Also, as most of these specifications have different authors, they are not always in perfect harmony with one another. The study of both the GIST and RMD implementations led to divergences in the expected actions to be taken in both cases, as will be explained next.

- The initial intent of this assignment was to update and expand the existing implementations. For the GIST implementation this ended up being a complete redesign and re-implementation, for the reasons explained in section 3.1. The resulting re-implementation conforms to a far higher degree to all of the requirements set in 1.2. Some of these requirements were tested in chapter 5. In the case of the RMD implementation, a different path was taken to implement something that was better suited to achieving the main goal of the assignment. Although for both implementations a divergence was made from the original intent, this was done for good reasons that ultimately helped in achieving the main goal of the assignment

- An application API that allows applications to request QoS was designed and implemented, as described in section 4.2.3. A client-side implementation was made both in a testing application and in a multimedia streaming application.

- The GIST, QoS-NSLP and application API were used in a demonstration as described in section 5.3.4. This demonstration shows that the basic functionality of the implementation, such as performing and tearing down reservations, works as expected.

With this last item the achievement of the main goal of this assignment, use of the NSIS protocol suite in a QoS aware multimedia application, was demonstrated.

Additionally there should be some closing remarks on the role of verifying the implementability and applicability of the NSIS protocol suite. It is difficult to consider NSIS as a single body of work. Its specifications are made by a lot of different people from a lot of different institutions with disparate interests. The result is that NSIS has a tendency towards becoming overly complex, allowing for a lot of optional features that may or may not be implemented, and also at times ambiguous. This has a chance of hindering interoperability, certainly one of the goals of the NSIS specifications, and this may result in a hesitance of vendors to implement all or any of the protocols. In this light, the small amount of interoperability testing performed within the scope of this assignment is the least to say unfortunate.

Of course, this being said, the work in this thesis was performed while most of the specifications were still in a development stage, with discussion going on all the time. Some of the problems encountered during the work done for this thesis were a cause for participation in this discussion, hopefully contributing to the standardisation process.

6.2. Future Work

Reccommended future work that could be done is the following:

- Updating the implementation performed to the latest specifications. Although it was developed according to specifications that were current at the time, new iterations are being released periodically. Adherence to the extensibility requirement should facilitate in this updating.
- Implementing QoS-NSLP functionality fully and integrating the RMD QoSM, based on the existing RMD implementation, and complementing its functionality.
- Performance evaluation of the GIST and QoS-NSLP implementations in order to evaluate scalability.

Bibliography

[1] Debian linux operating system. URL http://www.debian.org/.

[2] eunuchs - trac. URL http://www.inoi.fi/open/trac/eunuchs/.

[3] Gentoo linux operating system. URL http://www.gentoo.org.

[4] Hierarchical token bucket. URL http://luxik.cdi.cz/~devik/qos/htb/.

[5] Python ipqueue. URL http://woozle.org/~neale/src/ipqueue/.

[6] iproute2 package. URL http://linux-net.osdl.org/index.php/Iproute2.

[7] Jugi's traffic generator. URL http://www.cs.helsinki.fi/u/jmanner/software/jtg/.

[8] Linux advanced routing & traffic control. URL http://lartc.org/.

[9] netfilter/iptables project. URL http://www.netfilter.org/.

[10] The python programming language, . URL http://www.python.org.

[11] Python library reference, . URL http://docs.python.org/lib/lib.html.

[12] Rapi – an RSVP application programming interface. URL http://www.isi.edu/rsvp/DOCUMENTS/rsvpapi.txt.

[13] Tls lite. URL http://trevp.net/tlslite/.

[14] User-mode linux. URL http://user-mode-linux.sourceforge.net/.

[15] Videolan - VLC media player. URL http://www.videolan.org/.

[16] Wikipedia: Integrated services, . URL http://en.wikipedia.org/wiki/Intserv.

[17] Wikipedia: Python (programming language), . URL http://en.wikipedia.org/wiki/Python_(programming_language).

[18] J. Ash, A. Bader, and C. Kappler. QoS NSLP QSPEC template. RFC draft 11, IETF, August 2006. URL http://www.ietf.org/internet-drafts/draft-ietf-nsis-qspec-11.txt.

[19] A. Bader, G. Karagiannis, C. Kappler, and T. Phelan. RMD-QOSM - the resource management in diffserv QOS model. RFC draft 08, IETF, October 2006. URL http://www.ietf.org/internet-drafts/draft-ietf-nsis-rmd-08.txt.

[20] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Service. RFC 2475 (Informational), December 1998. URL `http://www.ietf.org/rfc/rfc2475.txt`. Updated by RFC 3260.

[21] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. RFC 1633 (Informational), June 1994. URL `http://www.ietf.org/rfc/rfc1633.txt`.

[22] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. RFC 2205 (Proposed Standard), September 1997. URL `http://www.ietf.org/rfc/rfc2205.txt`. Updated by RFCs 2750, 3936, 4495.

[23] A. Csaszar, A. Takacs, R. Szabo, and T. Henk. Resilient reduced-state resource reservation. *Journal of Communication and Networks*, 7(4), December 2005.

[24] András Császár, Attila Takács, and Attila Báder. A practical method for the efficient resolution of congestion in an on-path reduced-state signalling environment. *Lecture Notes in Computer Science*, 3552/2005:286–297, June 2005.

[25] C. Dickmann, I. Juchem, S. Willert, and X. Fu. A stateless ping tool for simple tests of GIMPS implementations. RFC draft 01, IETF, May 2005. URL `http://user.informatik.uni-goettingen.de/~nsis/files/internet-drafts/draft-juchem-ns`

[26] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999. URL `http://www.ietf.org/rfc/rfc2246.txt`. Obsoleted by RFC 4346, updated by RFC 3546.

[27] R. Elz and R. Bush. Serial Number Arithmetic. RFC 1982 (Proposed Standard), August 1996. URL `http://www.ietf.org/rfc/rfc1982.txt`.

[28] R. Hancock, G. Karagiannis, J. Loughney, and S. Van den Bosch. Next Steps in Signaling (NSIS): Framework. RFC 4080 (Informational), June 2005. URL `http://www.ietf.org/rfc/rfc4080.txt`.

[29] C. Kappler and X. Fu. A QoS model for signaling IntServ controlled-load service with NSIS. RFC draft 04, IETF, June 2006. URL `http://www.ietf.org/internet-drafts/draft-kappler-nsis-qosmodel-controlledload-04.tx`

[30] D. Katz. IP Router Alert Option. RFC 2113 (Proposed Standard), February 1997. URL `http://www.ietf.org/rfc/rfc2113.txt`.

[31] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), March 2006. URL `http://www.ietf.org/rfc/rfc4340.txt`.

[32] J. Manner and X. Fu. Analysis of Existing Quality-of-Service Signaling Protocols. RFC 4094 (Informational), May 2005. URL `http://www.ietf.org/rfc/rfc4094.txt`.

[33] J. Manner, G. Karagiannis, and A. McDonald. NSLP for quality-of-service signaling. RFC draft 11, IETF, June 2006. URL http://www.ietf.org/internet-drafts/draft-ietf-nsis-qos-nslp-11.txt.

[34] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980. URL http://www.ietf.org/rfc/rfc768.txt.

[35] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. URL http://www.ietf.org/rfc/rfc791.txt. Updated by RFC 1349.

[36] J. Postel. Internet Control Message Protocol. RFC 792 (Standard), September 1981. URL http://www.ietf.org/rfc/rfc792.txt. Updated by RFC 950.

[37] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. URL http://www.ietf.org/rfc/rfc793.txt. Updated by RFC 3168.

[38] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), September 2001. URL http://www.ietf.org/rfc/rfc3168.txt.

[39] H. Schulzrinne and R. Hancock. GIST: General internet signaling transport. RFC draft 09, IETF, February 2006. URL http://www.ietf.org/internet-drafts/draft-ietf-nsis-ntlp-09.txt.

[40] Chuck Semeria. Supporting differentiated service classes: Queue scheduling disciplines. *Juniper Networks Whitepaper*, December 2001.

[41] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960 (Proposed Standard), October 2000. URL http://www.ietf.org/rfc/rfc2960.txt. Updated by RFC 3309.

[42] Martijn Swanink. RMD (resource management in diffserv) within NSIS (next steps in signaling): protocol implementation. Master's thesis, University of Twente, July 2006. URL http://dacs.ewi.utwente.nl/documents/reports/master/Thesis_Martijn_Swanink.pdf.

[43] J. Wroclawski. Specification of the Controlled-Load Network Element Service. RFC 2211 (Proposed Standard), September 1997. URL http://www.ietf.org/rfc/rfc2211.txt.

[44] Mayi Zoumaro-Djayoon. Next step in signaling transport protocol/general internet signaling protocol (NTLP/GIST). Master's thesis, University of Twente, October 2005. URL http://dacs.cs.utwente.nl/assignments/completed/Thesis_Zoumaro.pdf.

A. GIST and QoS-NSLP Logging Example

This is the logging output of the Responding node in the scenario shown in Figure 2.4 and described in section 3.7.1. Additionally, it shows QoS-NSLP debugging information, in which a reservation is received, a notify message sent back and the reservation propagated downstream.

```
==================== Tue Nov 28 16:36:03 2006 ====================
GIST server initializing...
==================== Tue Nov 28 16:36:03 2006 ====================
ICMP Service initialized
==================== Tue Nov 28 16:36:03 2006 ====================
UDP Query Service initialized
==================== Tue Nov 28 16:36:03 2006 ====================
UDP Service initialized
==================== Tue Nov 28 16:36:03 2006 ====================
QoS NSLP Server initialized
==================== Tue Nov 28 16:36:03 2006 ====================
API Service initialized
==================== Tue Nov 28 16:36:03 2006 ====================
TCP Service initialized
==================== Tue Nov 28 16:36:03 2006 ====================
TLS Service initialized
==================== Tue Nov 28 16:36:03 2006 ====================
GIST server started
==================== Tue Nov 28 16:36:09 2006 ====================
RawService: Got packet from ipqueue, processing...
==================== Tue Nov 28 16:36:09 2006 ====================
Incoming message in D-Mode Query from interface eth2 with ttl 64:
GIST-Query Message (complete):
Common Header:
        Version: 1
        GIST hops: 63
        Message length: 27
        NSLPID: 6
        Type: Query
        S: 0
        R: 1
        E: 0
TLV Header
        Extensibility: Mandatory
        Type: MRI
        Length: 5
MRI Object:
        MRM-ID: PathCoupled
        N: 0
        Path Coupled MRM:
                IP-Ver: IPv4
                P: 1
                T: 0
                F: 0
                S: 0
                A: 0
                B: 1
                D: 0 (Downstream)
                Source Address: 192.168.3.2
                Destination Address: 192.168.0.1
                Source Prefix: 32
                Destination Prefix: 32
                Protocol: 17
                Destination Port: 12121
TLV Header
        Extensibility: Mandatory
        Type: SID
        Length: 4
SID Object:
        Session ID:
        0x109f1f05
        0x74ccdf24
        0xf6596703
```

```
        0x53761bbf
TLV Header
        Extensibility: Mandatory
        Type: NLI
        Length: 5
NLI Object:
        PI-Length: 7
        IP-TTL: 64
        IP-Ver: 4
        Routing State Validity Time: 60000
        Peer Identity:
        0x636c6965
        0x6e7431
        Interface Address: 192.168.3.2
TLV Header
        Extensibility: Mandatory
        Type: QueryCookie
        Length: 1
Query Cookie Object:
        Query Cookie:
        0x2ea15f0f
TLV Header
        Extensibility: Mandatory
        Type: StackProposal
        Length: 2
Stack Proposal Object:
        Prof-Count: 1
        Profile stack: 1 layers
                MA-Protocol-ID: TCPForward
TLV Header
        Extensibility: Mandatory
        Type: StackConfigurationData
        Length: 4
Stack Configuration Data Object:
        MA-Hold-Time: 30
        Protocol Option:
                MA-Protocol-ID: TCPForward
                Profile: 1
                Length: 2
                D: 0
                Options Data:
                0x3037
==================== Tue Nov 28 16:36:09 2006 ====================
Sending Service Primitive: RecvMessage
        NSLP-Data:

        NSLP-Data-Size: 0
        NSLPID: 6
        Session-ID:
        0x109f1f05
        0x74ccdf24
        0xf6596703
        0x53761bbf
        MRI:
        0x00004840
        0xc0a80302
        0xc0a80001
        0x20201100
        0x00002f59
        Routing-State-Check: True
        SII-Handle:
        0x00003036
        0x00000007
        0x636c6965
        0x6e7431c0
        0xa80302
        Transfer-Attributes: Reliability: False, Security: False
        IP-TTL: 64
        IP-Distance: 1
        GHC: 63
        InboundInterface: eth2
==================== Tue Nov 28 16:36:09 2006 ====================
Received rx_Query on Responder State Machine:
        State: Awaiting Confirm
        NSLPID: 6
        MRI:
MRI Object:
        MRM-ID: PathCoupled
        N: 0
        Path Coupled MRM:
                IP-Ver: IPv4
                P: 1
                T: 0
                F: 0
```

```
                S: 0
                A: 0
                B: 1
                D: 1 (Upstream)
                Source Address: 192.168.3.2
                Destination Address: 192.168.0.1
                Source Prefix: 32
                Destination Prefix: 32
                Protocol: 17
                Destination Port: 12121
        SID:
        0x109f1f05
        0x74ccdf24
        0xf6596703
        0x53761bbf
        Hop Distance to peer: 0
        Messages in queue: 0
        Reliable MA: Not Connected
        Secure MA: Not Connected
==================== Tue Nov 28 16:36:09 2006 ====================
Added state machine: Messaging Association State Machine
        State: Awaiting Connection
        Reliable: True Secure: False
        Protocol Stack: TCPForward
        Messages in queue: 0
Peer NLI Object:
        PI-Length: 7
        IP-TTL: 64
        IP-Ver: 4
        Routing State Validity Time: 60000
        Peer Identity:
        0x636c6965
        0x6e7431
        Interface Address: 192.168.3.2
        Peer UDP Port: 12342
==================== Tue Nov 28 16:36:09 2006 ====================
Added state machine: Messaging Association State Machine
        State: Awaiting Connection
        Reliable: True Secure: True
        Protocol Stack: TLSForward TCPForward
        Messages in queue: 0
Peer NLI Object:
        PI-Length: 7
        IP-TTL: 64
        IP-Ver: 4
        Routing State Validity Time: 60000
        Peer Identity:
        0x636c6965
        0x6e7431
        Interface Address: 192.168.3.2
        Peer UDP Port: 12342
==================== Tue Nov 28 16:36:09 2006 ====================
Sending Datagram message to 192.168.3.2:12342 with ttl 64:
GIST-Response Message (complete):
Common Header:
        Version: 1
        GIST hops: 64
        Message length: 32
        NSLPID: 6
        Type: Response
        S: 1
        R: 1
        E: 0
TLV Header
        Extensibility: Mandatory
        Type: MRI
        Length: 5
MRI Object:
        MRM-ID: PathCoupled
        N: 0
        Path Coupled MRM:
                IP-Ver: IPv4
                P: 1
                T: 0
                F: 0
                S: 0
                A: 0
                B: 1
                D: 1 (Upstream)
                Source Address: 192.168.3.2
                Destination Address: 192.168.0.1
                Source Prefix: 32
                Destination Prefix: 32
                Protocol: 17
```

```
                Destination Port: 12121
TLV Header
        Extensibility: Mandatory
        Type: SID
        Length: 4
SID Object:
        Session ID:
        0x109f1f05
        0x74ccdf24
        0xf6596703
        0x53761bbf
TLV Header
        Extensibility: Mandatory
        Type: NLI
        Length: 5
NLI Object:
        PI-Length: 5
        IP-TTL: 1
        IP-Ver: 4
        Routing State Validity Time: 20000
        Peer Identity:
        0x65646765
        0x31
        Interface Address: 192.168.3.1
TLV Header
        Extensibility: Mandatory
        Type: QueryCookie
        Length: 1
Query Cookie Object:
        Query Cookie:
        0x2ea15f0f
TLV Header
        Extensibility: Mandatory
        Type: ResponderCookie
        Length: 1
Responder Cookie Object:
        Responder Cookie:
        0x29506300
TLV Header
        Extensibility: Mandatory
        Type: StackProposal
        Length: 3
Stack Proposal Object:
        Prof-Count: 2
        Profile stack: 1 layers
                MA-Protocol-ID: TCPForward
        Profile stack: 2 layers
                MA-Protocol-ID: TLSForward
                MA-Protocol-ID: TCPForward
TLV Header
        Extensibility: Mandatory
        Type: StackConfigurationData
        Length: 6
Stack Configuration Data Object:
        MA-Hold-Time: 30
        Protocol Option:
                MA-Protocol-ID: TCPForward
                Profile: 1
                Length: 2
                D: 0
                Options Data:
                0x3037
        Protocol Option:
                MA-Protocol-ID: TCPForward
                Profile: 2
                Length: 2
                D: 0
                Options Data:
                0x3038
=================== Tue Nov 28 16:36:09 2006 ===================
Added state machine: Responder State Machine:
        State: Awaiting Confirm
        NSLPID: 6
        MRI:
MRI Object:
        MRM-ID: PathCoupled
        N: 0
        Path Coupled MRM:
                IP-Ver: IPv4
                P: 1
                T: 0
                F: 0
                S: 0
                A: 0
```

```
                B: 1
                D: 1 (Upstream)
                Source Address: 192.168.3.2
                Destination Address: 192.168.0.1
                Source Prefix: 32
                Destination Prefix: 32
                Protocol: 17
                Destination Port: 12121
        SID:
        0x109f1f05
        0x74ccdf24
        0xf6596703
        0x53761bbf
        Hop Distance to peer: 1
        Messages in queue: 0
        Reliable MA: Connected
        Secure MA: Connected
=================== Tue Nov 28 16:36:09 2006 ===================
Got new TCP connection from 192.168.3.2
=================== Tue Nov 28 16:36:09 2006 ===================
Incoming message in C-Mode from interface eth2 with ttl 0:
GIST-Confirm Message (complete):
Common Header:
        Version: 1
        GIST hops: 63
        Message length: 44
        NSLPID: 6
        Type: Confirm
        S: 1
        R: 0
        E: 0
TLV Header
        Extensibility: Mandatory
        Type: MRI
        Length: 5
MRI Object:
        MRM-ID: PathCoupled
        N: 0
        Path Coupled MRM:
                IP-Ver: IPv4
                P: 1
                T: 0
                F: 0
                S: 0
                A: 0
                B: 1
                D: 0 (Downstream)
                Source Address: 192.168.3.2
                Destination Address: 192.168.0.1
                Source Prefix: 32
                Destination Prefix: 32
                Protocol: 17
                Destination Port: 12121
TLV Header
        Extensibility: Mandatory
        Type: SID
        Length: 4
SID Object:
        Session ID:
        0x109f1f05
        0x74ccdf24
        0xf6596703
        0x53761bbf
TLV Header
        Extensibility: Mandatory
        Type: NLI
        Length: 5
NLI Object:
        PI-Length: 7
        IP-TTL: 64
        IP-Ver: 4
        Routing State Validity Time: 60000
        Peer Identity:
        0x636c6965
        0x6e7431
        Interface Address: 192.168.3.2
TLV Header
        Extensibility: Mandatory
        Type: ResponderCookie
        Length: 1
Responder Cookie Object:
        Responder Cookie:
        0x29506300
TLV Header
```

```
        Extensibility: Mandatory
        Type: StackProposal
        Length: 3
Stack Proposal Object:
        Prof-Count: 2
        Profile stack: 1 layers
                MA-Protocol-ID: TCPForward
        Profile stack: 2 layers
                MA-Protocol-ID: TLSForward
                MA-Protocol-ID: TCPForward
TLV Header
        Extensibility: Mandatory
        Type: StackConfigurationData
        Length: 2
Stack Configuration Data Object:
        MA-Hold-Time: 30
TLV Header
        Extensibility: Mandatory
        Type: NSLPData
        Length: 17
NSLP Data Object:
        NSLP Data:
        0x01050000
        0x00020002
        0x5fc0c2ec
        0xdc825637
        0x00010001
        0xde6ecaa4
        0x00030001
        0x0000ea60
        0x00070008
        0x00000100
        0x00010006
        0x80020005
        0x48f42400
        0x47000000
        0x07f80000
        0x00000001
        0x000005dc
==================== Tue Nov 28 16:36:09 2006 ====================
Received rx_Confirm on Responder State Machine:
        State: Awaiting Confirm
        NSLPID: 6
        MRI:
MRI Object:
        MRM-ID: PathCoupled
        N: 0
        Path Coupled MRM:
                IP-Ver: IPv4
                P: 1
                T: 0
                F: 0
                S: 0
                A: 0
                B: 1
                D: 1 (Upstream)
                Source Address: 192.168.3.2
                Destination Address: 192.168.0.1
                Source Prefix: 32
                Destination Prefix: 32
                Protocol: 17
                Destination Port: 12121
        SID:
        0x109f1f05
        0x74ccdf24
        0xf6596703
        0x53761bbf
        Hop Distance to peer: 1
        Messages in queue: 0
        Reliable MA: Connected
        Secure MA: Connected
==================== Tue Nov 28 16:36:09 2006 ====================
Received tg_Connected on Messaging Association State Machine
        State: Awaiting Connection
        Reliable: True Secure: False
        Protocol Stack: TCPForward
        Messages in queue: 0
Peer NLI Object:
        PI-Length: 7
        IP-TTL: 64
        IP-Ver: 4
        Routing State Validity Time: 60000
        Peer Identity:
        0x636c6965
```

```
        0x6e7431
        Interface Address: 192.168.3.2
        Peer UDP Port: 12342
==================== Tue Nov 28 16:36:09 2006 ====================
Sending Service Primitive: RecvMessage
        NSLP-Data:
        0x01050000
        0x00020002
        0x5fc0c2ec
        0xdc825637
        0x00010001
        0xde6ecaa4
        0x00030001
        0x0000ea60
        0x00070008
        0x00000100
        0x00010006
        0x80020005
        0x48f42400
        0x47000000
        0x07f80000
        0x00000001
        0x000005dc
        NSLP-Data-Size: 68
        NSLPID: 6
        Session-ID:
        0x109f1f05
        0x74ccdf24
        0xf6596703
        0x53761bbf
        MRI:
        0x00004840
        0xc0a80302
        0xc0a80001
        0x20201100
        0x00002f59
        Routing-State-Check: False
        SII-Handle:
        0x01003036
        0x00010100
        0x07636c69
        0x656e7431
        0xc0a80302
        Transfer-Attributes: Reliability: True, Security: False
        IP-TTL: 0
        IP-Distance: 1
        GHC: 63
        InboundInterface: eth2
==================== Tue Nov 28 16:36:09 2006 ====================
QoS NSLP: Received QoS-NSLP Message:
QoS NSLP Reserve Message (complete):
Common Header:
        Message Type: Reserve
        Q: 1
        T: 0
        R: 1
        S: 0
TLV header
        Extensibility: Mandatory
        Type: RSN
        Length: 2
RSN Object:
        RSN: 1606468332
        Epoch Identifier: 3699529271
TLV header
        Extensibility: Mandatory
        Type: RII
        Length: 1
RII Object:
        RII: 3731802788
TLV header
        Extensibility: Mandatory
        Type: RefreshPeriod
        Length: 1
REFRESH_PERIOD Object:
        Refresh Period: 60000 ms
TLV header
        Extensibility: Mandatory
        Type: QSpec
        Length: 8
QSPEC Object
        Version: 0
        QOSM ID: IntServ CL
        Messsage Sequence: Sender Initiated Reservation
```

```
        Object Combination: 1
        QSPEC Object: QoS Desired
                E: 0
                Q: 0
                Length: 6
        QSPEC Parameter: Traffic
                M: 1
                E: 0
                N: 0
                R: 0
                Length: 5
                Parameter:
                Token Bucket Rate: 500000.000000 bytes/s
                Token Bucket Size: 32768.000000 bytes
                Peak Data Rate: inf bytes/s
                Minimum Policed Unit: 1 bytes
                Maximum Pakcet Size: 1500 bytes
==================== Tue Nov 28 16:36:09 2006 ====================
QoS NSLP: Sending QoS-NSLP Message:
QoS NSLP Notify Message (complete):
Common Header:
        Message Type: Notify
TLV header
        Extensibility: Mandatory
        Type: InfoSpec
        Length: 2
INFO_SPEC Object
        Error Code: Reduced refreshes supported
        E-Class: Informational
        ESI Type: IPv4
        ESI_Length: 1
        Error Source Identifier - IPv4
                Address: 192.168.3.1
==================== Tue Nov 28 16:36:09 2006 ====================
Received Service Primitive: SendMessage
        NSLP-Data:
        0x04000000
        0x00060002
        0x00031101
        0xc0a80301
        NSLP-Data-Size: 16
        NSLP-Message-Handle: None
        NSLPID: 6
        Session-ID:
        0x109f1f05
        0x74ccdf24
        0xf6596703
        0x53761bbf
        MRI:
        0x00004860
        0xc0a80302
        0xc0a80001
        0x20201100
        0x00002f59
        Transfer-Attributes: Reliability: True, Security: False
==================== Tue Nov 28 16:36:09 2006 ====================
Received tg_NSLPData on Responder State Machine:
        State: Established
        NSLPID: 6
        MRI:
MRI Object:
        MRM-ID: PathCoupled
        N: 0
        Path Coupled MRM:
                IP-Ver: IPv4
                P: 1
                T: 0
                F: 0
                S: 0
                A: 0
                B: 1
                D: 1 (Upstream)
                Source Address: 192.168.3.2
                Destination Address: 192.168.0.1
                Source Prefix: 32
                Destination Prefix: 32
                Protocol: 17
                Destination Port: 12121
        SID:
        0x109f1f05
        0x74ccdf24
        0xf6596703
        0x53761bbf
        Hop Distance to peer: 1
```

```
        Messages in queue: 0
        Reliable MA: Connected
        Secure MA: Connected
==================== Tue Nov 28 16:36:10 2006 ====================
Received tg_RawData on Messaging Association State Machine
        State: Connected
        Reliable: True Secure: False
        Protocol Stack: TCPForward
        Messages in queue: 0
Peer NLI Object:
        PI-Length: 7
        IP-TTL: 64
        IP-Ver: 4
        Routing State Validity Time: 60000
        Peer Identity:
        0x636c6965
        0x6e7431
        Interface Address: 192.168.3.2
        Peer UDP Port: 12342
==================== Tue Nov 28 16:36:10 2006 ====================
Sending TCP message to 192.168.3.2 with ttl 64:
GIST-Data Message (complete):
Common Header:
        Version: 1
        GIST hops: 64
        Message length: 16
        NSLPID: 6
        Type: Data
        S: 1
        R: 0
        E: 0
TLV Header
        Extensibility: Mandatory
        Type: MRI
        Length: 5
MRI Object:
        MRM-ID: PathCoupled
        N: 0
        Path Coupled MRM:
                IP-Ver: IPv4
                P: 1
                T: 0
                F: 0
                S: 0
                A: 0
                B: 1
                D: 1 (Upstream)
                Source Address: 192.168.3.2
                Destination Address: 192.168.0.1
                Source Prefix: 32
                Destination Prefix: 32
                Protocol: 17
                Destination Port: 12121
TLV Header
        Extensibility: Mandatory
        Type: SID
        Length: 4
SID Object:
        Session ID:
        0x109f1f05
        0x74ccdf24
        0xf6596703
        0x53761bbf
TLV Header
        Extensibility: Mandatory
        Type: NSLPData
        Length: 4
NSLP Data Object:
        NSLP Data:
        0x04000000
        0x00060002
        0x00031101
        0xc0a80301
==================== Tue Nov 28 16:36:10 2006 ====================
QoS NSLP: Reserving:
                Token Bucket Rate: 500000.000000 bytes/s
                Token Bucket Size: 32768.000000 bytes
                Peak Data Rate: inf bytes/s
                Minimum Policed Unit: 1 bytes
                Maximum Pakcet Size: 1500 bytes
==================== Tue Nov 28 16:36:10 2006 ====================
QoS NSLP: Added reservation:
QoS NSLP Persistent State:SID:
0x109f1f05
```

```
        0x74ccdf24
        0xf6596703
        0x53761bbf
Flow ID consists of 1 MRIs
MRI Object:
        MRM-ID: PathCoupled
        Path Coupled MRM:
                IP-Ver: IPv4
                P: 1
                T: 0
                F: 0
                S: 0
                A: 0
                B: 1
                D: 0 (Downstream)
                Source Address: 192.168.3.2
                Destination Address: 192.168.0.1
                Source Prefix: 32
                Destination Prefix: 32
                Protocol: 17
                Destination Port: 12121
Upstream SII:
0x01003036
        0x00010100
        0x07636c69
        0x656e7431
        0xc0a80302
Downstream SII: Unknown
Upstream RSN: 1606468332
Upstream Epoch Identifier: 3699529271
Local RSN: 419703489
Waiting for 0 RIIs:
Waiting for 0 passthrough RIIs:
Lifetime: 60000 ms
0 other sessions bound to this one:
Unknown if downstream peer supports reduced refreshes
Bandwidth:
                Token Bucket Rate: 500000.000000 bytes/s
                Token Bucket Size: 32768.000000 bytes
                Peak Data Rate: inf bytes/s
                Minimum Policed Unit: 1 bytes
                Maximum Pakcet Size: 1500 bytes
=================== Tue Nov 28 16:36:10 2006 ===================
QoS NSLP: Sending QoS-NSLP Message:
QoS NSLP Reserve Message (complete):
Common Header:
        Message Type: Reserve
        Q: 1
        T: 0
        R: 1
        S: 0
TLV header
        Extensibility: Mandatory
        Type: RSN
        Length: 2
RSN Object:
        RSN: 419703489
        Epoch Identifier: 3543578790
TLV header
        Extensibility: Mandatory
        Type: RII
        Length: 1
RII Object:
        RII: 3731802788
TLV header
        Extensibility: Mandatory
        Type: RefreshPeriod
        Length: 1
REFRESH_PERIOD Object:
        Refresh Period: 60000 ms
TLV header
        Extensibility: Mandatory
        Type: QSpec
        Length: 8
QSPEC Object
        Version: 0
        QOSM ID: IntServ CL
        Messsage Sequence: Sender Initiated Reservation
        Object Combination: 1
        QSPEC Object: QoS Desired
                E: 0
                Q: 0
                Length: 6
        QSPEC Parameter: Traffic
```

```
                M: 1
                E: 0
                N: 0
                R: 0
                Length: 5
                Parameter:
                Token Bucket Rate: 500000.000000 bytes/s
                Token Bucket Size: 32768.000000 bytes
                Peak Data Rate: inf bytes/s
                Minimum Policed Unit: 1 bytes
                Maximum Pakcet Size: 1500 bytes
==================== Tue Nov 28 16:36:10 2006 ====================
Received Service Primitive: SendMessage
        NSLP-Data:
        0x01050000
        0x00020002
        0x19042ac1
        0xd336b8a6
        0x00010001
        0xde6ecaa4
        0x00030001
        0x0000ea60
        0x00070008
        0x00000100
        0x00010006
        0x80020005
        0x48f42400
        0x47000000
        0x07f80000
        0x00000001
        0x000005dc
        NSLP-Data-Size: 68
        NSLP-Message-Handle: None
        NSLPID: 6
        Session-ID:
        0x109f1f05
        0x74ccdf24
        0xf6596703
        0x53761bbf
        MRI:
        0x00004840
        0xc0a80302
        0xc0a80001
        0x20201100
        0x00002f59
        Transfer-Attributes: Reliability: True, Security: False
==================== Tue Nov 28 16:36:10 2006 ====================
Sending Downstream Query encapsulated message with ttl 64:
GIST-Query Message (complete):
Common Header:
        Version: 1
        GIST hops: 64
        Message length: 27
        NSLPID: 6
        Type: Query
        S: 0
        R: 1
        E: 0
TLV Header
        Extensibility: Mandatory
        Type: MRI
        Length: 5
MRI Object:
        MRM-ID: PathCoupled
        N: 0
        Path Coupled MRM:
                IP-Ver: IPv4
                P: 1
                T: 0
                F: 0
                S: 0
                A: 0
                B: 1
                D: 0 (Downstream)
                Source Address: 192.168.3.2
                Destination Address: 192.168.0.1
                Source Prefix: 32
                Destination Prefix: 32
                Protocol: 17
                Destination Port: 12121
TLV Header
        Extensibility: Mandatory
        Type: SID
        Length: 4
```

```
SID Object:
        Session ID:
        0x109f1f05
        0x74ccdf24
        0xf6596703
        0x53761bbf
TLV Header
        Extensibility: Mandatory
        Type: NLI
        Length: 5
NLI Object:
        PI-Length: 5
        IP-TTL: 64
        IP-Ver: 4
        Routing State Validity Time: 60000
        Peer Identity:
        0x65646765
        0x31
        Interface Address: 192.168.2.2
TLV Header
        Extensibility: Mandatory
        Type: QueryCookie
        Length: 1
Query Cookie Object:
        Query Cookie:
        0xa766039b
TLV Header
        Extensibility: Mandatory
        Type: StackProposal
        Length: 2
Stack Proposal Object:
        Prof-Count: 1
        Profile stack: 1 layers
                MA-Protocol-ID: TCPForward
TLV Header
        Extensibility: Mandatory
        Type: StackConfigurationData
        Length: 4
Stack Configuration Data Object:
        MA-Hold-Time: 30
        Protocol Option:
                MA-Protocol-ID: TCPForward
                Profile: 1
                Length: 2
                D: 0
                Options Data:
                0x3037
=================== Tue Nov 28 16:36:10 2006 ===================
Added state machine: Query State Machine:
        State: Awaiting Response
        NSLPID: 6
        MRI:
MRI Object:
        MRM-ID: PathCoupled
        N: 0
        Path Coupled MRM:
                IP-Ver: IPv4
                P: 1
                T: 0
                F: 0
                S: 0
                A: 0
                B: 1
                D: 0 (Downstream)
                Source Address: 192.168.3.2
                Destination Address: 192.168.0.1
                Source Prefix: 32
                Destination Prefix: 32
                Protocol: 17
                Destination Port: 12121
        SID:
        0x109f1f05
        0x74ccdf24
        0xf6596703
        0x53761bbf
        Hop Distance to peer: 0
        Messages in queue: 1
        Reliable MA: Not Connected
        Secure MA: Not Connected
=================== Tue Nov 28 16:36:10 2006 ===================
Dying: Messaging Association State Machine
        State: Awaiting Connection
        Reliable: True Secure: True
        Protocol Stack: TLSForward TCPForward
```

```
        Messages in queue: 0
Peer NLI Object:
        PI-Length: 7
        IP-TTL: 64
        IP-Ver: 4
        Routing State Validity Time: 60000
        Peer Identity:
        0x636c6965
        0x6e7431
        Interface Address: 192.168.3.2
        Peer UDP Port: 12342
==================== Tue Nov 28 16:36:21 2006 ====================
GIST server terminated
```