University of Twente
Enschede - The Netherlands

University of Twente
Master of Science Thesis in Computer Science
Faculty of Electrical Engineering, Mathematics and Computer Science
Design and Analysis of Communication Systems

Implementation of aggregation based Resource Management in DiffServ (RMD) Quality of Service Model (QOSM)

Timothy Sealy
August 26, 2008

Committee:
Dr. ir. Georgios Karagiannis (UT/DACS)
Dr. ir. Geert Heijenk (UT/DACS)
Prof. dr. Hans van den Berg (UT/DACS)

Preface

This thesis is the result of a final thesis project for the Department of Electrical Engineering of the faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS) at the University of Twente. Most of the work was conducted at the laboratory of the Design and Analysis of Communication Systems (DACS) group at the University of Twente.

First and foremost I want to thank my girlfriend, whom I love very much. It has been a long and bumpy road but baby, we made it. Thank you for your support and patience. You always know how to pick me up when I feel down.

Special thanks go out to the members of my examination committee dr. ir. Georgios Karagiannis, dr. ir. Geert Heijenk and prof. dr. Hans van den Berg. Many thanks go to my supervisor dr. ir. Georgios Karagiannis, for his support and insights.

Last but not least I would like to thank Ruud Klaver for his knowledge of the Linux operating system and his guidance during the implementation phase.

Contents

Chapter 1

Introduction

Quality of service is a term used to describe the overall user experience a user or application will receive over a network. In the fields of packet-switched networks and computer networking, the term Quality of Service refers to resource reservation control mechanisms. These resources are reserved, for a user or application, in order to provide certain (QoS) guarantees for the data transfer across a network. Different priorities can be assigned to different users or data flows, and a certain level of performance can be guaranteed to a data flow in accordance with requests from the application program or the internet service provider policy. It is needless to say that Quality of Service guarantees are important if the network capacity is limited. This is typically the case for real-time streaming multimedia applications, for example voice over IP and IP-TV, since these often require a fixed bit rate and may be delay sensitive.

Consider for example a voice application. On a plain old telephony system, voice traffic experiences a low and fixed amount of delay with no loss. Replicating this behaviour on an IP based network is difficult because the IP network introduces a variable and unpredictable amount of delay to the packets and also drops voice packets when the network is congested. The best effort IP network does not provide the necessary means to provide the needed behaviour. QoS techniques are needed to provide the means to control delay and packet loss.

The Internet, largely build on the Internet Protocol, is a best effort network offering no QoS assurances. Due to its popularity and explosive growth it has lead to many services being deployed over the Internet. For example, not so long a go an enterprise would have had an TDM-base voice network for telephony, an ISDN network for video conferencing and a multi-protocol (IPX, AppleTalk) LAN. Today many of these services are deployed over the IP based Internet. Not only enterprises seem benefit from this change, consumers also seem to enjoy the ease and comfort of these services. It is not uncommon for consumers to do their shopping online, watch streaming media over the Internet and make telephone calls using Voice-Over-IP (VOIP) technology. With the growing number of services being offered over the Internet and the increasing (bandwidth) demand of these services, it is necessary to employ QoS techniques into the next generation networks. Only in this way we can develop and deploy advanced network applications and technologies for tomorrow's Internet [26, 27]

Currently there are two major approaches standardized in the Internet Engineering Task Force (IETF): Integrated-Services (IntServ) [5] and Differentiated Services (DiffServ) [4].

1.1 Integrated Services

The idea behind the Integrated Services (IntServ) architecture is that every router in the network implements mechanisms for admission control and resource reservation. Every application that requires some kind of guarantee for a particular flow has to make an individual reservation at every router in the path of the data flow. In order to signal the QoS requirements to the routers the resource reservation protocol (RSVP) is used. The RSVP protocol is described in [6] and specifies how the reservation states are installed, maintained and removed in the routers.

The main problem of the IntServ architecture is that it is not scalable. Routers have to store the

reservation state of every flow that traverse them. For a large scale network like the Internet - where there are millions of flows to be maintained - it places a heavy burden on the routers. Not only is the amount of states that need to be maintained high but also the amount of signaling required to set up, maintain and remove these states is large. In order to address these scalability problems the Differentiated Services architecture was designed.

1.2 Differentiated Services

In the Differentiated Services (DiffServ) architecture routers maintain flows on a coarser granularity. Instead of maintaining per flow reservation states like in the IntServ architecture, flows are maintained per traffic class. In a DiffServ domain a distinction is made between interior routers and border routers. The interior routers are mainly responsible for packet forwarding and routing, whereas all other functionality like admission control, traffic classification and conditioning are pushed to the border routers. All routers in the network maintain a small amount reservation states based on the number of traffic classes. The border routers classify and mark packets passing through the domain. Using Per-Hop Behaviors (PHBs) the packets are associated with a particular class. PHBs define the packet forwarding properties associated with a class of traffic. Thus all the routers in the network provide a certain level of QoS for packets based on the PHB of a flow. The PHB is specified using a 6-bit value called the Differentiated Services Code Point (DSCP) which is stored in the Differentiated Services (DS) field of the IP packet header.

In the DiffServ architecture the network is viewed as a collection of various domains based on administrative boundaries. At the domain boundaries, service level agreements (SLAs) are made regarding the amount of resources allocated to traffic that crosses the domains. A service level agreement also refers to the contract between a service provider and its customers. In this case the SLA specifies for example a peak bit rate, which the customer is responsible for not exceeding. All excess traffic will be dropped. For better link utilization, dynamic SLAs should be supported so customers can request bandwidth on demand.

1.3 NSIS

The Next Steps in Signaling Working Group is responsible for standardizing an IP signaling protocol with QoS signaling as the first use case. The focus of the working group is to re-use, where appropriate, the protocol mechanisms of RSVP, while at the same time simplifying it in order to come to a more general signaling model.

In [16] the NSIS signaling protocol suite is defined. The protocol suite is divided into a generic (lower) layer, with separate upper layers for each specific signaling application. In the upper layer the NSIS Signaling Layer Protocol (NSLP) protocol is defined which establishes and maintains reservation states at routers along the path of a data flow, for the purpose of providing some forwarding resources for that flow. This resource provisioning however can only be accomplished using QoS Models (QoSMs). Such a QoSM is the Resource Management in DiffServ (RMD) QoSM which can be used to provide dynamic resource management within DiffServ, see: http://www.ietf.org/html.charters/nsis-charter.html.

RMD-QoSM comes in different flavors such as measurement based RMD-QoSM, reservation based RMD-QoSM and aggregation based RMD. In measurement based RMD-QoSM admission control and traffic conditioning is done using traffic measurement whereas the reservation and aggregation based RMD use traffic descriptors for admission control and traffic conditioning. For the purpose of this thesis we will design and implement the aggregation based RMD-QoSM. This aggregation based RMD-QoSM is similar to the RMD-QoSM reservation based scheme, with the difference that the boundary nodes of a RMD-QoSM aware domain support the aggregated reservations instead of per flow reservations.

1.4 Problem Definition

Aggregation implies the grouping of several smaller flows into one large flow, called the aggregate flow. The main benefit of aggregation is that the management of flows becomes easier. This is because instead of having to manage many individual flows, also called micro flows, only several large flows need to be managed.

Flows have a rate at which they transfer data. Based on this rate a certain amount of bandwidth is reserved in order to provide the necessary QoS. Note that flows can either send data at a constant bit rate (CBR) or at a variable bit rate (VBR). For CBR flows this implies that the rate of the flows does not change during the flow's lifetime. The amount of bandwidth that needs to be reserved for this type of flows is simply equal to the flow's rate. For VBR flows the rate fluctuates during the flows lifetime. For this type of flows there are different techniques in determining the amount of bandwidth that needs to be reserved. The easiest way to provision bandwidth for these flows is to use the flow's peak rate. By reserving the amount of bandwidth equal to the flow's peak rate the flow always has enough bandwidth reserved for its traffic rate. This way hard QoS guarantees can be provided. Another approach would be to reserve the amount of bandwidth equal to the flow's average bit rate. Whenever the flow's rate exceeds the reserved bandwidth amount, the excessive packets are sent as best effort. Note that in this case hard QoS guarantees are not provided but higher bandwidth utilization is achieved.

For the purpose of this thesis we will only discuss the aggregation of CBR flows. Thus we are not considering the fluctuation of the (data) traffic rate of an individual flow. Whenever an individual flow requests a certain amount of bandwidth and this request is granted, its rate cannot exceed this limit. If flows send more packets than their allowable limit, then these packets will be either shaped, dropped or send as best effort.

The rate of the aggregate flow depends on the rates of the individual flows which make up the aggregate. In the following equation we define the rate of the aggregate flow, which we will refer to as the *aggregate traffic rate* from now on, to be equal to the sum of rates of the individual flows:

$$r_a = \sum_{i=1}^{n} r_i \tag{1.1}$$

where r_a is the aggregate traffic rate, r_i is the rate of the i^{th} flow in the aggregate, and n is the number of flows in the aggregate.

Flows are constantly joining and leaving the aggregate flow and thus the rate of the aggregate flow fluctuates over time. In order to provide sufficient bandwidth for the individual flows the size of the aggregate flow needs to constantly be updated. Figure 1.1 shows how the aggregate traffic rate fluctuates over time. Note that the fluctuations of the aggregate traffic rate occurs on a large timescale of hours. We will not be addressing the fluctuations on a smaller time scale. It is important to know what the relevance of the timescale is.

Fluctuations of the aggregate traffic rate on a small time scale are due to flows joining or leaving the aggregate reservation. Thus in order to capture these fluctuations, and design an update policy to cope with these fluctuations, it would make sense to model the statistical properties of the flow arrivals and departures. For the fluctuations on a larger scale these statistical models can not be used. In this case the effects of the individual flows are negligible small. Building a statistical model for the arrival and departure of end-to-end flows is not very useful because this model does not capture the long term fluctuation we are interested in. The large time scale fluctuations are mainly determined by the combined activities of different groups of people. For example the internet activity on a certain backbone provider is higher during business hours than outside business hours. The first peak would occur between 10am to 12am, while the second peak would occur in the late afternoon between 3pm to 5pm. Usually a prediction of the future course of the aggregate traffic rate is computed based on historic measurements.

In order to provide the necessary QoS for the aggregate flow we need to reserve bandwidth for this flow. The amount of bandwidth to be reserved depends on the aggregate traffic rate and must be equal to or

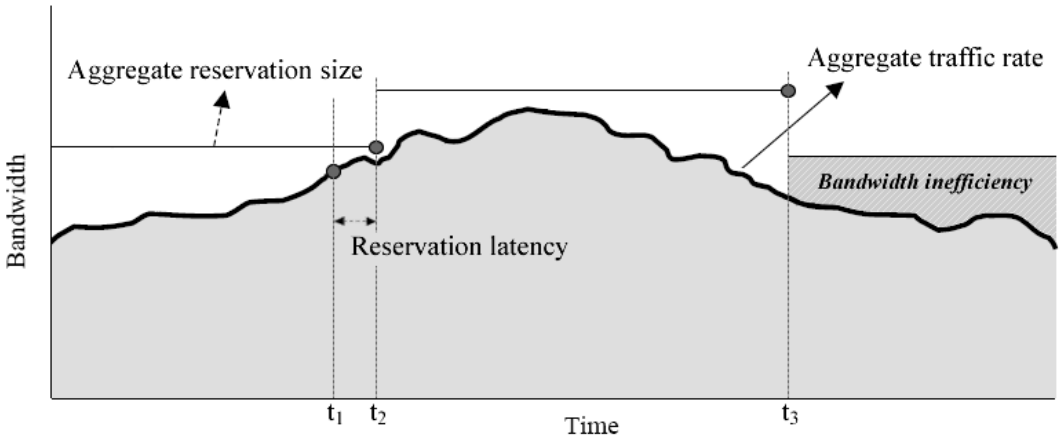


Figure 1.1: The aggregate traffic rate.

larger than the aggregate traffic rate. First a reservation for the aggregate flow needs to be set up so that the individual flows can be added to this aggregate flow. After the aggregate reservation has been set up the amount of reserved bandwidth, referred to as the size of the aggregate reservation, is updated according to the dynamics of the aggregate traffic rate and the aggregate update policy. The *aggregate update policy* states when the size of the aggregate reservation needs to be updated and what the new size of the aggregate reservation should be.

One policy for updating the bandwidth of the aggregate reservation is to increase the amount of bandwidth reserved by the aggregate flow whenever a new flow joins this aggregate. And in a similar manner release bandwidth whenever a flow leaves the aggregate. We will refer to this policy as the *IntServ update policy*. Note that this solution is not very scalable and does not take advantage of the aggregation of the flows. For example, if flows where not aggregated then for every flow bandwidth would have to be reserved in the routers on the path of that flow. Control messages would have to be sent to request bandwidth for this flow and state information would have to be maintained at the routers of this flow's path. If we choose to update the aggregate reservation state (stored in the routers on the path of the aggregate flow) every time a flow joins or leaves the aggregate the amount of control message that needs to be sent is similar to the case of no aggregation. The only gain in this case would be a reduction in the amount of state information that needs to be maintained in the routers, because instead of maintaining the state information for the individual flows the routers now simply maintain the state information for the aggregate flow.

In order to improve this performance we need to reserve additional bandwidth for the aggregate to cope with new flows joining the aggregate. In Figure 1.1 we see a horizontal line which depicts the *aggregate reservation size*. The aggregate reservation size, which we will denote by C_a, is the amount of bandwidth reserved for the aggregate flow. Flows can join the aggregate as long as it has enough spare bandwidth to grant their request. At some point in time the amount of spare bandwidth of the aggregate might not be sufficient for potential new flows. An attempt is then made to increase the aggregate reservation size (denoted in Figure 1.1 by time t_1). A request for extra bandwidth is sent to routers in the path of the aggregate flow. Each router checks locally whether it can grant this request and once all the routers have granted this request the aggregate flow has the additional amount of bandwidth at his disposal. Only then is the aggregate reservation size increased to its new level (this is denoted in Figure 1.1 by time t_2). As one can see, the aggregate reservation size can not be increased instantaneous. Because the additional bandwidth needs to be reserved in all the routers on the path of the aggregate, it might take a while before the aggregate reservation size is increased. This delay between the time the aggregate initiates the request for extra bandwidth and the time this additional bandwidth is granted is referred to as the *reservation latency*. This delay is most likely in the order of milliseconds but can also be in the order of seconds.

The aggregate reservation size should not only be increased but also needs to be decreased from time to time. This should be done in case the aggregate traffic rate is significantly lower than the aggregate reservation size. If the aggregate reservation size is not decreased then the aggregate has reserved (much) more bandwidth then it actually needs. This spare bandwidth, referred to as the *bandwidth inefficiency*, needs to be freed (partially) in order for other flows to use. In Figure 1.1 we see the bandwidth inefficiency marked by the shaded area. Also shown in the figure is how at time t_3 the aggregate reservation size is reduced. Note that the decrease of the aggregate reservation is instantaneous and is not subject to the reservation latency. This is because the release of resources does not have to be confirmed. Once a router receives a signaling message indicating the release of a certain amount of bandwidth, this bandwidth is released immediately. The newly freed up bandwidth is then available for other flows.

For the purpose of this thesis we will design an aggregate update policy that takes the issues discussed above into account. In order to design a proper aggregate update policy a solution to the optimization problem discussed in the following section needs to be found.

1.4.1 Optimization Problem

The problem we are faced with is how to design an aggregate update policy that is scalable and has a reasonable utilization. In addition, our aggregate update policy should reject as few flows as possible. The update policy we need to design is responsible for the following two things:

1. when should the aggregate reservation be updated. The update policy should decide when to increase the aggregate reservation size such that the aggregate reservation can cope with the future aggregate traffic rate. It should also determine when to decrease the aggregate reservation size such that the utilization of the aggregate reservation is as high as possible.
2. what the new size of the aggregate should be. Aside from determining when to update the aggregate reservation the policy should determine the new size of the aggregate reservation.

It should be clear by now that there is a trade-off between the signaling load and the utilization. A high signaling load results in a high utilization for our update policy while a low signaling load yields a low utilization for our update policy. Take for example the *IntServ update policy* described above. This policy has a 100% utilization at the cost of having to update the aggregate reservation for every flow joining or leaving the aggregate. Thus aside from the highest possible utilization, this policy also has the highest possible signaling load.

We want to find a solution that balances the trade-off between the signaling load and the utilization. In order to find such a policy we need to describe the problem above as an optimization problem. In words we can define our optimization problem as follows:

> *Can we design an aggregate update policy so that the signaling load, the (normalized) bandwidth inefficiency and the blocking probability are as low as possible? Or in other words, when do we need to update the aggregate reservation and what should the new size be so that the signaling load, the (normalized) bandwidth inefficiency and the blocking probability are as low as possible?*

We can express the problem defined above using a cost function. Before we can do so we will define the parameters involved in the optimization problem. The relevant parameters are:

- The signaling load $S(t)$.
- The (normalized) bandwidth inefficiency $BI'(t)$.
- The blocking probability P_{bl}.

The details of these parameters are discussed in the following section. For now we will use these parameters to describe our cost function c. This is defined as follows:

$$c = \kappa_1 \times S(t) + \kappa_2 \times BI'(t) + \kappa_3 \times P_{bl} \tag{1.2}$$

where κ_1, κ_2 and κ_3 are constants used to normalize the signaling load $S(t)$, the normalized bandwidth inefficiency $BI'(t)$ and the blocking probability P_{bl}. The constants κ_1, κ_2 and κ_3 are weights that are assigned to the different parameters indicating their importance. For a given set of constants we can now calculate the cost of the update policy.

Solving our optimization problem means finding the update policy with the lowest cost. In other words we need to minimize the cost function c. This problem however is rather theoretical and for the purpose of this assignment we will not attempt to solve it. We will however use this cost function during the testing of our implementation. The cost function can then be used to determine whether a given set of configuration parameters is better than another set of configuration parameters. In the section "*Proposed Solution*" the chosen update policy and the configurable parameters are discussed. The configuration of these parameters is discussed in the chapter "*Implementation and Testing*". In the next section we will discuss the constraints for the aggregate update policy in more detail.

1.4.2 Constraints

The constraints we place on the aggregate update policy are as follows:

- The sum of rates for the individual flows should be lower or equal to the aggregate reservation size at all times. The following equation denotes this constraint:

  $$r_a \leq C_a \tag{1.3}$$

  where r_a is the aggregate traffic rate, as calculated in Equation (1.1), and C_a the maximum capacity of the aggregate reservation.

- The update frequency of the aggregate reservation size should be as small as possible. The fewer updates the more scalable the solution. The number of updates for the aggregate reservation during a time interval of length t is denoted by $n_u(t)$. The signaling load is defined as follows:

  $$S(t) = \frac{n_u(t)}{n_a(t) + n_d(t)} \tag{1.4}$$

  where $n_u(t)$ is the number of update messages sent up to a certain time t, $n_a(t)$ is the number of reservation requests received up to time t, and $n_d(t)$ the number of flows that have left the aggregate up to time t. Note that in case the IntServ update policy is used, the number of update messages sent is equal to the number of flows that have joined plus the number of flows that have left the aggregate. This yields a signaling load of 1, which is the maximum value for $S(t)$.

- The bandwidth inefficiency should be as low as possible. Reserving excessive amounts of bandwidth which is not utilized by the individual flows in the aggregate prevents other (aggregate) flows from reserving these resources. In Figure 1.1 bandwidth inefficiency is defined as the area between the aggregate reservation size and the aggregate traffic rate. It can be computed using the following equation:

  $$BI(t) = \int_{x=t_0}^{x=t} C_a(x) - r_a(x)dx \tag{1.5}$$

  where $BI(t)$ is the bandwidth inefficiency for the aggregate reservation up to time t, $C_a(x)$ is the aggregate reservation size and $r_a(x)$ is the aggregate traffic rate. The bandwidth inefficiency parameter defined above expresses the amount of bandwidth that was not utilized by the aggregate reservation in terms of number of (mega)bits. Because we are more interested in the fraction of spare

bandwidth which is not used by the aggregate flow we define the *normalized bandwidth inefficiency* as follows:

$$BI'(t) = \frac{BI(t)}{\int_{x=t_0}^{x=t} C_a(x)dx} \tag{1.6}$$

The normalized bandwidth inefficiency can also be expressed in terms of *utilization* which is basically the portion of the aggregate reservation size that is used by the aggregate traffic rate. Utilization is denoted by U_r and is calculated as follows:

$$U_r(t) = 1 - BI'(t) = 1 - \frac{BI(t)}{\int_{x=t_0}^{x=t} C_a(x)dx} = 1 - \frac{\int_{x=t_0}^{x=t} C_a(x) - r_a(x)dx}{\int_{x=t_0}^{x=t} C_a(x)dx} = \frac{\int_{x=t_0}^{x=t} r_a(x)dx}{\int_{x=t_0}^{x=t} C_a(x)dx} \tag{1.7}$$

Note that for the IntServ update policy the utilization is near 100%. This is because the aggregate reservation size is always equal to the aggregate rate. In other words $C_a(t) \approx r_a(t)$ and thus the utilization $U_r(t) \approx \frac{\int_{x=t_0}^{x=t} r_a(x)dx}{\int_{x=t_0}^{x=t} r_a(x)dx} = 1$. The normalized bandwidth inefficiency in this case is zero.

- The update policy should reject as few flows as possible. The update policy should somehow predict the future aggregate traffic rate and attempt to reserve this bandwidth in advance. It should not be the case that a flow is rejected because the aggregate has not reserved enough bandwidth to accept this flow. Let's assume that an aggregate reservation has the possibility to reserve an unlimited amount of bandwidth so that the requests for extra bandwidth are always granted. If in this case a (micro) flow is rejected, it's because the aggregate reservation was not updated in time. In order to address this problem we need to update the aggregate reservation whenever the predicted future bandwidth over the reservation latency period is larger than the current aggregate reservation size. In other words, we need to update the aggregate if the extra future bandwidth demand is larger than our current spare bandwidth. The spare bandwidth forms a buffer that enables us to accept flows while we are increasing the aggregate reservation size. This can be expressed in the following probability:

$$P_{bl}\left\{r_a(t') > C_a(t)\right\} \leq \epsilon \tag{1.8}$$

where t' denotes the current time plus the reservation latency delay ($t' = t + \Delta t_{rl}$), $r_a(t')$ is the future aggregate traffic rate at the end of the reservation latency period, $C_a(t)$ is the current aggregate reservation size and ϵ is a small error term. So the probability that our future aggregate traffic rate over the period $[t, t + \Delta t]$ exceeds the current aggregate reservation size may be no more than ϵ. If the probability is larger than ϵ, then the aggregate reservation needs to be increased.

1.5 Outline

This thesis is structured as follows; Chapter 2 will provide the requirements and criteria for aggregation based RMD-QoSM. Different aggregation methods and their corresponding admission control are discussed here. In Chapter 3 an overview of the NSIS protocol suite and the details relevant to this thesis will be described. A description of the RMD-QoSM is also given. The complete specification of the aggregation based RMD-QoSM is given in Chapter 4 and its design and implementation in Chapter 5. If the reader is familiar with the NSIS protocol and the RMD-QoSM, Chapter 3 and 4 may be skipped. Chapter 6 describes the experiments used to evaluate the performance of our implementation, along with the results . Finally, the conclusions and recommendations are given in Chapter 7.

Chapter 2

Related Work and Proposed Solution

2.1 Related Work

In this section we will discuss the different solutions proposed by others regarding the aggregation of flows. Much research has been done regarding the optimization problem described in the previous chapter. In order to reduce the search space for our problem we will narrow our scope to threshold based aggregate update schemes.

2.1.1 Schmitt et al.

Our problem comes closest to the one defined in Schmitt et al. [32]. Schmitt et al. [32] have studied the time scales of different QoS architectures. They state that different QoS architectures have different serving needs. For example an access provider that has a moderate load and directly connects end systems may have a fast time scale system because it has to respond immediately to the end system requests. A backbone provider that connects the access providers is generally faced with a higher load of individual transmission. For the backbone provider it is usually not possible to react on the time scale of individual requests, so a slower time scale system needs to be enforced. It is particularly an issue for access providers that use the IntServ/RSVP architecture to suit their customers' needs and connects these end systems to backbone networks that use the DiffServ architecture (with a Bandwidth Broker). The problem of using different time scales can be illustrated using the following figure.

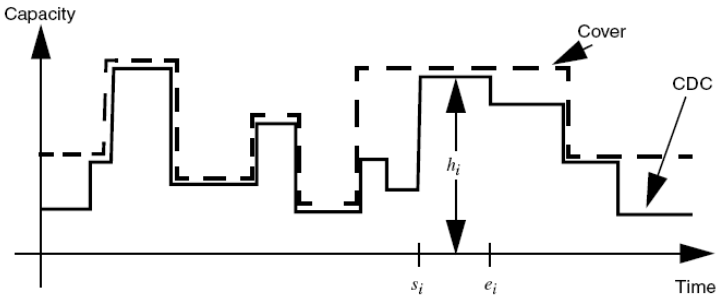


Figure 2.1: Example of different timescales.

In Figure 2.1 we see how the aggregate traffic rate is modeled as a stepwise function. Here, CDC stands for the *capacity demand curve* which is the aggregate traffic rate. The CDC belongs to the underlying QoS system which has a fast time scale, compared to the overlay QoS which has a slower time scale. The capacity demand for the overlay QoS system is depicted by the *cover* in Figure 2.1, which covers the underlying CDC. The capacity demand curve ($R(t)$) of the underlying system is computed as follows:

$$R(t) = \sum_{i=1}^{n^R} f_i^R(t) \tag{2.1}$$

where $f_i^R(t)$ is computed as follows:

$$f_i^R(t) = \begin{cases} h_i^R & , \ t \in [s_i^R, e_i^R] \\ 0 & , \ \text{otherwise} \end{cases}$$

with $e_i^R = s_{i+1}^R$ for $\forall i$.

Here, n^R is the number of steps, h_i^R is the height of step i, s_i^R is the start of step i, e_i^R is the end of step i. In order to provide the same QoS on a different time scale a cover is computed over CDC $R(t)$. Formally a cover is defined as a capacity demand curve $\bar{R}(t)$ for which $\bar{R}(t) \geq R(t) \forall t$.

In Schmitt et al. [32] the main reason why the overlay system needs to enforce a slower time scale, is because the setup costs for requests in the overlay network are relatively high. Based on this assumption a cost model is designed which takes the flow setup and bandwidth usage into account. The cost of a CDC $R(t)$ for the underlying QoS systems in a given time period $[t_0, t_1]$ is defined as:

$$c(R|F,U) = F \times n^R + U \times \int_{t_0}^{t_1} R(t)dt \tag{2.2}$$

where F is the fixed setup cost for changing the requested capacity level and U is the variable cost per capacity unit. These parameters are assumed to be set beforehand and remain unchanged. The problem of decoupling the time scales can now be formulated as follows:

> Find a covering CDC $\bar{R}$ for R such that $c(\bar{R}|F,U)$ is minimal.

According to Schmitt et al. [32] it is rather hard to compute an optimal cover. The complexity lays in the following parameters:

- the step length $e_i^R - s_i^R$, which is a product of the inter-arrival times of the individual requests of the underlying QoS system.
- the step height h_i^R which corresponds to the aggregate capacity required to serve the requests.

It seems that especially the latter parameter is very difficult to model. It depends on the type of applications that issue the reservation requests and how widely the resource requirements for these applications differ from each other. The first parameter can be modeled using Markovian models known from the teletraffic theory as long as the individual requests are user initiated.

In order to address this problem an algorithm was developed to find the optimal cover for a certain known CDC $R(t)$. The optimal cover can be found by using a divide-and-conquer approach. First the peak rate of the CDC ($R(t)$) is found and the height of the cover is set to this value. The cover is then divided into two halves, namely the cover to the left of the peak and the cover on the right of the peak. In a recursive manner all possible covers are calculated for the left and right side of the peak rate. The optimal cover is the cover which has the lowest cost $c(\bar{R}|F,U)$ for a given set of F and U.

Exhaustively searching the search space for the optimal cover is a computational expensive task and is not a feasible solution. An approximation to the calculation of the optimal cover was developed in order to find near-optimal covers. The algorithm for computing near-optimal covers is similar to the algorithm of calculation optimal covers. In the near-optimal cover algorithm the peak rate of the CDC ($R(t)$) is found and the height of the cover is set to this value. The cover is thus again divided into two halves. Let's now denote t_{peak} as the timestamp at which the CDC $R(t)$ is at its peak rate. For the cover on the left of the peak the search for a second peak rate over the interval $[t_0, t_{peak}]$ is conducted. The algorithm now checks whether the cost of decreasing the cover CDC to this second peak is lower than the cost of keeping the height of the covering CDC unchanged up to this peak. If the cost of decreasing the cover is lower, the cover is decreased and the computation of the rest of the cover done in a recursive manner over the left side and the right side of the second peak rate. If the cost of leaving the height of cover unchanged

is lower than decreasing the cover, then the cover remains unchanged over this period. A search for a new peak rate to the left of the second peakrate is conducted and the rest of the cover computed in a recursive manner. Note that this process is also done for the right side of the first peak rate.

The algorithm discussed above can only be used when the CDC is known. Because the course of the CDC is not known before hand the algorithm needs to be modified. The proposed solution to deal with this uncertainty is to use a simple heuristic[1]. The heuristic is called thresholded depot excess (TDE) because it builds a 'depot' of capacity in order to stabilize the fluctuations of the underlying curve. The TDE has one parameter $\alpha_{TDE} \in [0, 1]$ which is used as a relative threshold.

The TDE algorithm works as follows:

- Let $D(t)$ be the capacity for the cover for the capacity demand curve $R(t)$. The parameter $D(t)$ corresponds to the amount of bandwidth reserved by the covering CDC.
- If the $R(t)$, in time interval t, rises above the current capacity depot $D(t-1)$ then $D(t)$ is increased to $R(t)$ immediately.
- A decrease of the cover $D(t)$ is only triggered when the value of $R(t)$ drops below a fraction of the current capacity ($R(t) < \alpha_{TDE} \times D(t-1)$). The value of $D(t)$ is then set to $R(t)$.

Several things can be noted from this algorithm. The capacity of the cover is only increased when it is absolutely necessary. It is the decreasing of the capacity of the cover that is actually regulated. The decrease is postponed until the capacity of the underlying CDC is well below the capacity of the cover. The parameter α_{TDE} has a crucial role in determining the threshold for the decrease of the cover's capacity. Setting α_{TDE} too high will result in many changes in the cover capacity while setting α_{TDE} too low wastes bandwidth. The key question here is how should the value of α_{TDE} be set.

For a given CDC we can calculate a covering CDC using the TDE algorithm discussed above. Using the approximation for the optimal cover we can also compute a near-optimal cover for this CDC. For both covers we can compute the cost of the cover, namely $c(R^{TDE,\alpha_{TDE}})$ and $c(R^{NEAROPT})$. In order to allow the TDE algorithm to produce a cover which closely matches the near-optimal cover, we need to find the value for α_{TDE} such that the following is minimized:

$$c(R^{TDE,\alpha_{TDE}}) - c(R^{NEAROPT}) \tag{2.3}$$

This minimization can be done using a simple recursive grid search over interval [0,1] for the parameter α_{TDE}.

Note that the above algorithm works with known CDC's. To use this algorithm one must keep track of some historic data about the CDC that the algorithm needs to cover. Based on this historic data a proper value for α_{TDE} is computed. The amount of historic data determines the performance of the algorithm. The more past information is used, the more accurate can the α_{TDE} be computed. However, if too much past information is used, this information might not reflect the current behaviour of the aggregate traffic. The amount of past information also has an impact on the computational cost of the near optimal cover. Calculating the cover for a large amount of past information can be quite expensive.

Using simulation the optimal value for the amount of history to be stored is calculated. However, due to the lack of empirical data these values can not be translated to actual meaning-full results. With this adaptive scheme Schmitt et al. [32] have shown that the optimization problem can be solved using a simple heuristic.

[1] Schmitt et al. [32] note that the proposed solution should be regarded as an illustrative example of how a heuristic can be used in their update scheme.

2.1.2 Pipes Model

A widely used model for the aggregation of flows is the *pipes model* [36, 24]. In the pipes model individual flows are mapped onto an aggregate using 'pipes'. A pipe is defined as *'a logical path between two end points on the network having a predefined capacity'* [36]. In a DiffServ domain a pipe would be a path between an Ingress and an Egress router. End-to-end communication over multiple domains using pipes could be achieved by creating different pipes of similar capacity over the different domains and connecting them together. Figure 2.2 gives a graphical representation of a pipe and how multiple pipes are connected together.

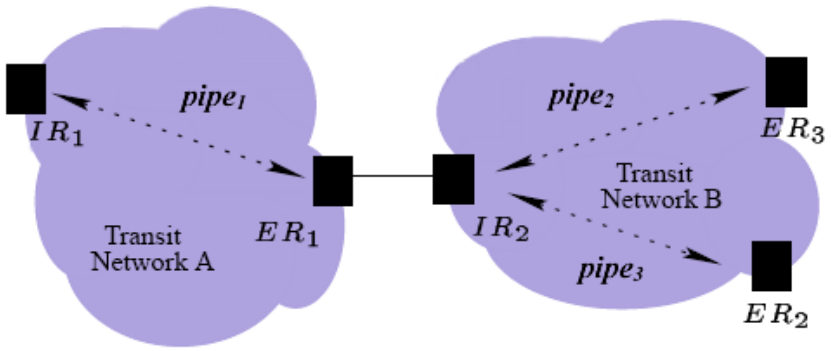


Figure 2.2: The pipes model.

The purpose of the pipe is to reduce the signaling and computation overhead in the core network. Flows between an Ingress-Egress pair belonging to the same traffic class are aggregated onto the pipe and signaling is done for the pipe instead of the individual flows.

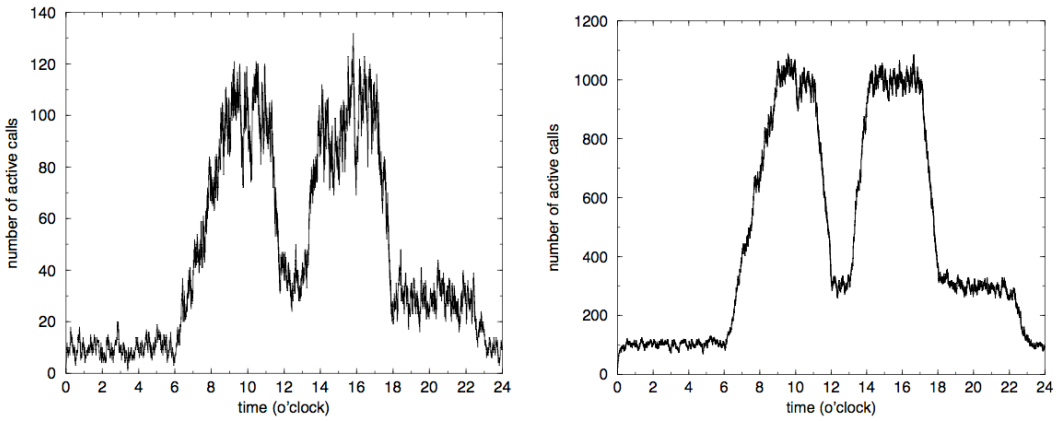
Wang et al. [36] propose a simple threshold based algorithm to update the pipe. Their results show that the relationship between the threshold δ and the utilization is somewhat linear while the relationship between δ and the signaling load is curved. This information can be used to select a proper value for the threshold δ. For example Wang et al. [36] have shown that sacrificing a little utilization can yield significant gains in the signaling load.

Menth [24] has also proposed the use of a threshold based update scheme similar to Wang et al. [36]. In his scheme he defines two thresholds δ_H and δ_L which denote a high threshold and a low threshold respectively. The region between δ_H and δ_L is called the *tolerance window*. The pipe is then updated whenever the aggregate traffic rate either exceeds the high threshold, or drops below the low threshold. Based on his analysis the following rule of thumb is derived for determining the values for the thresholds: if the radius of the tolerance window is set to the square root of the number of calls in the system, the average inter-update interval will be constant if there are a large number of flows in the aggregate. Below the details of these two solutions are discussed in more detail.

Wang et al.

Wang et al. [36] have developed a model that allows individual flows to be mapped onto an aggregate using 'pipes'. How flows are aggregated is not described in this paper but this could easily be solved by aggregating flows using DiffServ classes or other flow traffic characteristics.

Pipes do nothing more than reserve a certain amount of bandwidth between an Ingress and an Egress. As stated before, the traffic of the individual flows between the Ingress and the Egress is passed through the pipe. Because flows come and go it is necessary to update the capacity of the pipe regularly. Updating



(a) 100 calls (b) 1000 calls

Figure 2.3: Traffic model.

the pipe for every arrival or departure of a flow does not reduce the signaling overhead but results in high bandwidth utilization. On the other hand, updating the pipe infrequently results in a significant reduction of the signaling overhead but a lower bandwidth utilization. How frequently the capacity of the pipe is updated depends on the arrival and departure frequency of the individual flows. In Figure 2.3 we see a representation of average intensity of telephone calls a day on the British Telecom network. The figures depict an approximation of the real life behavior by modeling the call arrival process as a Poisson process. The length of each call is exponentially distributed with an average of 5 minutes. In Figure 2.3(a) the peak arrival rate is set to 20 calls per minute so that on average there are 100 calls in the pipe during the peak period (i.e., 9am-11am, 2pm-5pm). In Figure 2.3(b) the peak arrival rate is set to 200 calls per minute so that on average there are 1000 calls in the pipe during the peak period.

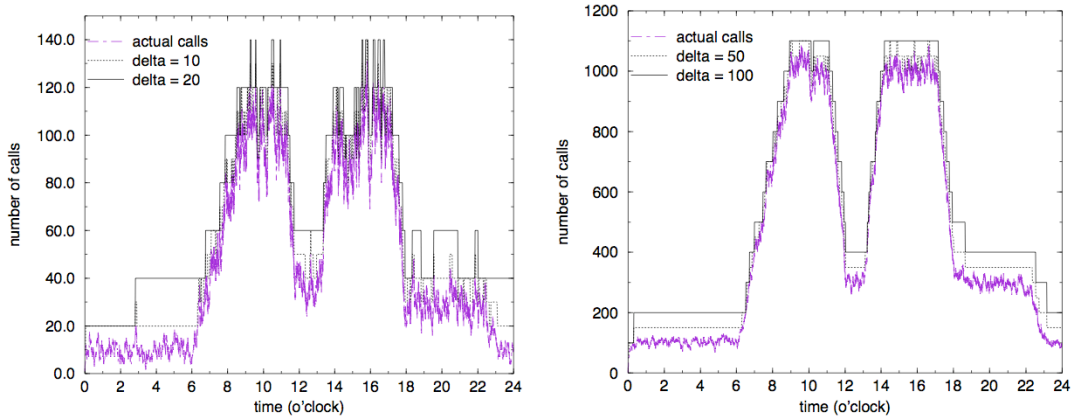
From the graph we observe that the relative burstiness of Figure 2.3(a) is larger than that of Figure 2.3(b), implying that the more calls there are the smoother the curve and the better we can predict the behaviour. This would ultimately result in a less frequent update of the pipe capacity. According to [36] the variance of the number of calls is subject to two main factors: 1) a long term factor which is caused by the different phone usage during different timeslots and 2) a short term factor which is caused by the Poisson call arrival process.

A prediction model is created to update the pipe. The pipe is updated in intervals of 40 minutes. First an ideal prediction model is taken which has knowledge of the peak rate in the next 40 minute interval. Based on this peak rate the capacity of the pipe is adjusted. Of course there is no way of knowing what the peak rate over the next 40 minutes is going to be. The goal however is to find a realistic prediction algorithm that has similar performance as the ideal prediction model described above. A simple threshold based prediction scheme is developed to achieve this (see Figure 2.4).

The value of the threshold δ is chosen based on the tradeoff analysis between the signaling overhead and the bandwidth utilization. The algorithm for updating the pipe capacity can be stated as follows:

1. Upon a call arrival, if the number of calls reaches the *pipe_capacity*, then *pipe_capacity* is increased by δ.
2. Upon a call departure, if the number of calls is under the $pipe\_capacity - 2 \times \delta$, then *pipe_capacity* is decreased by δ.

It is easy to see that if δ is equal to one that the scheme represents the per-flow updating scheme. The larger δ, the less frequent the pipe capacity is updated.



(a) 100 calls (b) 1000 calls

Figure 2.4: Threshold based aggregation.

In this scheme the number of calls is constrained as follows:

$$pipe\_capacity - 2 \times \delta \leq number\_of\_calls < pipe\_capacity \tag{2.4}$$

The utilization for this scheme depends solely on the value of δ. Because the aggregate traffic is always between $pipe\_capacity - 2 \times \delta$ and $pipe\_capacity$ we can compute the lower bound of the utilization using the following equation:

$$utilization \geq \frac{pipe\_capacity - 2 \times \delta}{pipe\_capacity} \tag{2.5}$$

The $normalized\_updates$ metric is used to evaluate how much updating overhead can be saved by using a pipe for the aggregate traffic. It is defined as follows:

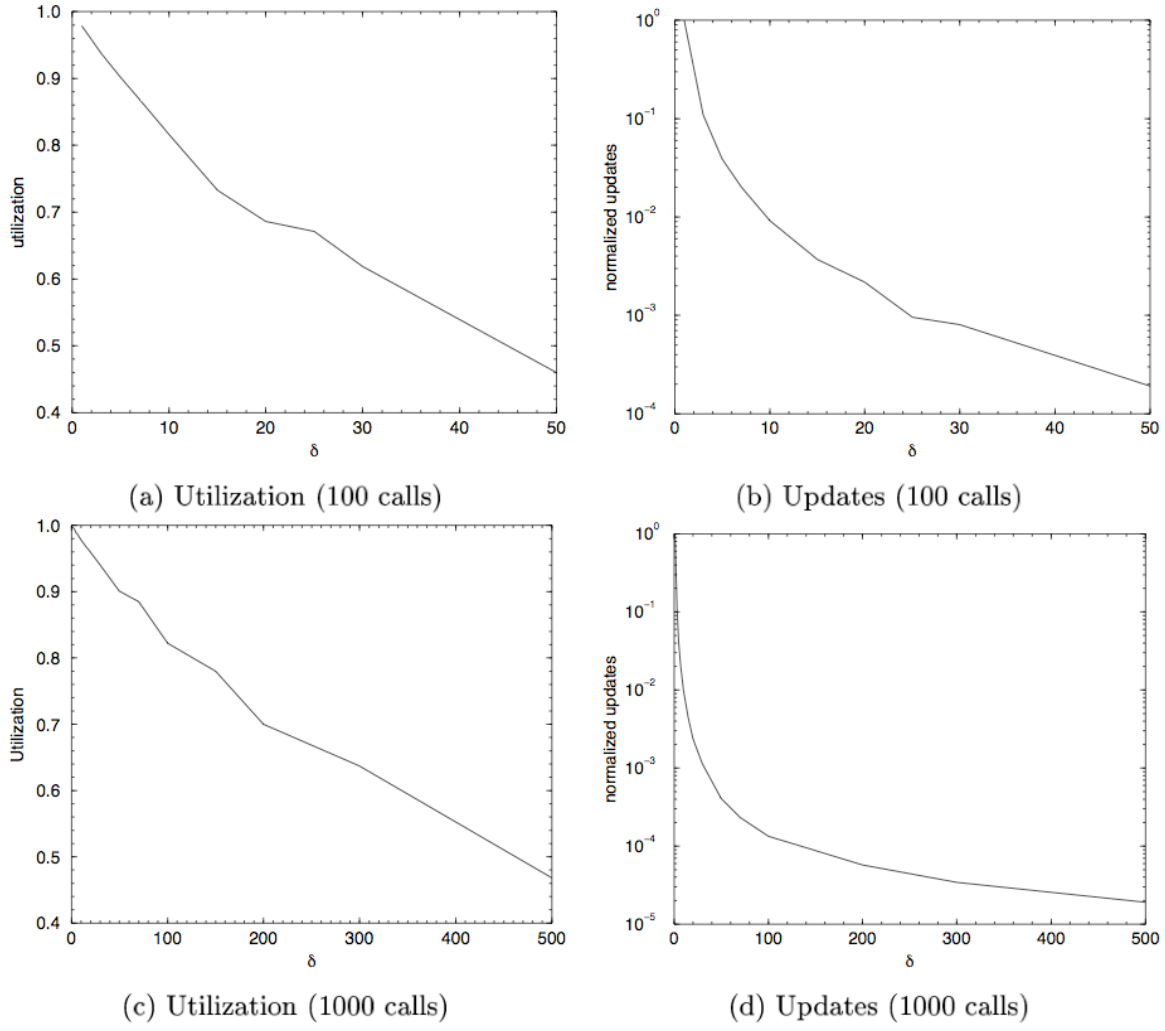
$$normalized\_updates = \frac{number\_of\_updates}{2 \times number\_of\_calls} \tag{2.6}$$

The graphs shown in Figure 2.5 depict the relationship between the threshold δ and the bandwidth, and the threshold δ and the (normalized) amount of updates.

From these figures we can conclude that the utilization has a somewhat linear relationship with the threshold δ. If we set δ to 1 we achieve near 100 % utilization whereas if δ is set to half times the average number of calls we get a utilization of about 50 %. In the both 100 calls and 1000 calls simulation we see that the utilization decreases with an increasing δ. If we select $\delta = 10$ in the case of 100 calls or $\delta = 100$ in the case of 1000 calls we find a utilization of about 80 %. For the number of updates it is a different story. Here we find that initially a small increase of δ leads to large decrease of the number of updates. This eventually flats out for large values of δ. If we look again at the value of $normalized\_updates$ for $\delta = 10$ in the 100 call simulation, we find a result of 10^{-2}. This corresponds to 99 % of updating overhead being removed. In the case of having an average of 1000 calls we find a value of 10^{-4}, which corresponds of 99.99 % of the updating overhead being removed.

Micheal Menth

Micheal Menth [24] also proposed a threshold based scheme similar to the one developed by Wang et al. [36]. In order to reduce the amount of signaling and making the solution more scalable, overreservation is proposed. By reserving more resources than needed, the aggregate can cope with small changes in demand. Overreservation decreases network performance and thus in order to avoid extensive waste of



(a) Utilization (100 calls)

(b) Updates (100 calls)

(c) Utilization (1000 calls)

(d) Updates (1000 calls)

Figure 2.5: Relationship between the utilization, number of updates and δ.

resources a simple control mechanism with a threshold θ is used. In Figure 2.6 a graphical representation of the mechanism is given. In the figure the overall capacity demand for the aggregate is depict. The size

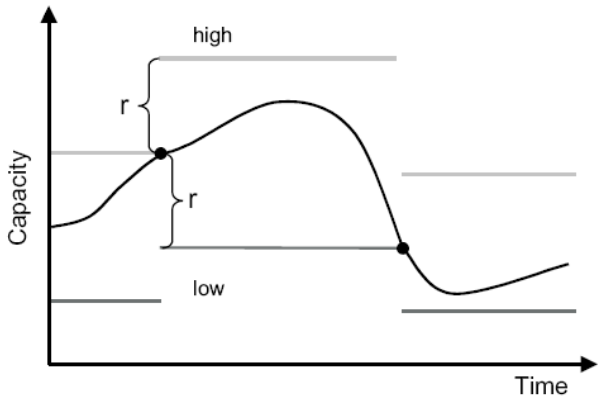


Figure 2.6: Threshold based aggregation.

of the aggregate reservation is only updated in the case that the demand drops under the lower threshold or exceeds the upper threshold.

The signaling load with - respect to the network performance - is analyzed using the mean inter-update time. The model used is based on models used in conventional telephony systems. Here the demand capacity of the aggregate is assumed to be proportional to the number of admitted end-to-end sessions $n(t)$. The lower and upper thresholds are denoted by n_{low} and n_{high} respectively. The aggregate reservation is updated whenever the demand leaves the tolerance window $[n_{low}, n_{high}]$. For two flows the inter-arrival time can be described by an exponentially distributed random variable A (i.e., $A(t) = 1 - e^{-\lambda t}$). The mean inter-arrival time is the inverse of the arrival rate ($E[A] = \frac{1}{\lambda}$). The holding time B is exponentially distributed as well with mean $E[B] = \frac{1}{\mu}$. The process $n(t)$ is modeled as a Markov birth-death process with a state transition from state i to $i+1$ having an arrival rate of λ and the transitions to state $i-1$ having a departure rate of $i \times \mu$. Also the process does not change its state with rate $-\lambda + i \times \mu$ and all other state transitions are not possible.

From this analysis a rule of thumb is derived which states that the tolerance window can be adjusted such that the mean inter-update time remains constant. The radius of the tolerance window r can be computed using: $r = \sqrt{\bar{n}}$ with $\bar{n}$ being the average aggregate window size. Because there is a linear relationship between the inter-update time and the size of the tolerance window a scalar (ω) can be used to regulate the inter-update time. It is easy to see that for large aggregates the degree of overreservation converges towards zero: $lim_{\bar{n}\to\infty} \frac{\lfloor \omega \times \sqrt{\bar{n}} \rfloor}{\bar{n}} = 0$. The rule of thumb proposed in this solution is thus practical, scalable and efficient (for large aggregates).

2.1.3 Bandwidth Broker

Another approach towards scalable QoS signaling is the Bandwidth Broker (BB) architecture. The idea of BB was first proposed by Nichols et al. [25] and introduces a new central logical entity that is responsible for both intra-domain and inter-domain resource management for DiffServ. Its goal is to provide IntServ-like end-to-end QoS guarantees in DiffServ networks. In Figure 2.7 we see two autonomous systems (AS) being controlled by the bandwidth brokers BB_1 and BB_2. Here ER and IR denote the Egress and Ingress routers of the domain respectively.

The tasks of a bandwidth broker are split into two main categories: intra-domain and inter-domain. Intra-domain tasks include resource management and traffic control within the domain of the BB. Inter-domain tasks cover the specification of bilateral service level agreements (SLAs) with neighboring domains and managing the boundary routers to police/shape the incoming/outgoing traffic to adhere to the SLAs.

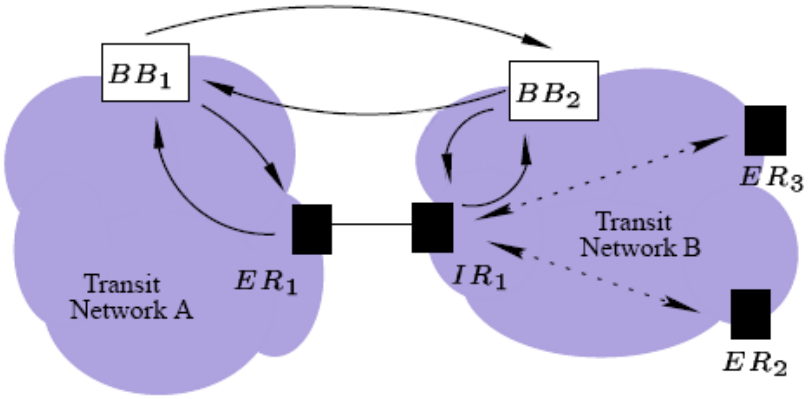


Figure 2.7: Inter domain signaling.

Pan et al. [28] designed a protocol called the Border Gateway Reservation Protocol (BGRP), which handles the inter-domain signaling. The main purpose of the protocol is to support resource reservation for aggregate flows. In Pan et al. [28] the main focus was on the development of the protocol rather then on the update scheme. For updating the aggregate reservation a simple scheme is proposed. However because this aspect has some significance for our problem we will briefly discuss the update scheme.

Mantar et al. [23] researched both the intra-domain as the inter-domain resource management for the BB architecture. For the Internet2 QoS working group [27] they have designed an architecture which defines these functions for the DiffServ architecture. In their architecture they assume the usage of a Bandwidth Broker in the DiffServ domain and the usage of pipes for the aggregation of individual flows. Details of their design is discussed below.

BGRP

The Border Gateway Reservation Protocol (BGRP) was also designed to address the scalability issues of RSVP. The major issue that BGRP was designed to address was how to aggregate flows over multiple domains. This was done by focusing on the inter-domain reservation. The protocol thus specifies how border routers of different domains can setup and maintain aggregate reservations. Here aggregation is done using sink trees. In [28] the BGRP is discussed in detail.

In order to reduce the amount of updates for the aggregate reservation overreservation, quantization and hysteresis is proposed. Border routers always reserve more bandwidth than is actually needed in order to cope with future demand. The size of the aggregate reservation is always a multiple of a certain quantity Q. At a certain point in time the size of the aggregate reservation would be $k \times Q$. If the capacity of the aggregate reservation is not sufficient enough to admit potential new flows the new size for the aggregate reservation is set to $(k+1) \times Q$. The aggregate size is decreased in a similar manner by setting the new size of the aggregate reservation at $(k-1) \times Q$. Hysteresis is used to limit the amount of decreases of the aggregate reservation size. A decrease is triggered only when the bandwidth requirement drops below some threshold $T_l < (k-1) \times Q$. In [28] this threshold is set to $(k-1) \times Q + 1$. Here it is assumed that individual flows require one unit of bandwidth and so the reservable bandwidth wasted due to over-reservation by this technique is less than $2Q$ units.

Mantar et al.

For the Internet2 QoS working group [27] Mantar et al. [22, 23] have conducted much research on how to provide a scalable QoS architecture for large networks. As a base for their architecture they assume the usage of the DiffServ architecture because it is relatively scalable with large networks. This is because '*the number of states in core routers are independent of the network size. Thus, it is considered as the de facto standard for the next generation of the Internet.* According to Mantar et al. [22, 23] the main open issue of the DiffServ architecture is the control plane function. This is because '*unlike the Intserv/RSVP, Diffserv only addresses forwarding/data plane functionality, whereas control plane functions still remain an open issue. Hence, DiffServ alone cannot provide end-to-end QoS guarantees.*' [22]. Furthermore they assume the usage of Bandwidth Brokers (BB) at the DiffServ domains. Here the BB is responsible for providing end-to-end *IntServ-like* QoS guarantees within the DiffServ networks. The main reason why the usage of a BB is preferred is because the '*control functionality such as policy control, admission control and resource reservation are decoupled from the routers into the BB*'. Some other advantages are:

- **Scalability:** A BB increases network core scalability due to the decoupling of control path functions.
- **Easy to deploy:** Network administrators are more likely to use the BB infrastructure because it requires little changes in the network infrastructure.
- **Simplified billing and accounting:** Billing and accounting associated with can be simplified because of the usage of a centralized network entity.
- **Less inconsistency of QoS states:** Another advantage of using a centralized server is the reduced amount of inconsistency of QoS states, faced by distributed approaches in which the edge routers make admission control decisions independent of each other.

All of these advantages makes the BB the strongest for their research on control path mechanisms for DiffServ. In order to address this issue Mantar et al. [23] have developed a signaling protocol for both intra-domain and inter-domain signaling. For intra-domain signaling the BB is used. No real signaling is used because all the control functionality is located in the BB. In other words the routers in the network do not have to be notified of QoS state changes. These routers are only responsible for the forwarding of the traffic data. In the next section the intra-domain signaling is explained.

Intra-domain signaling

In [23] Mantar et al. propose a centralized server called the intra-domain resource manager (IDRM) which is responsible for admission control and provisioning of the network. An architectural view of the IDRM is shown in Figure 2.8. Here all the information regarding the network topology, routing information, available links and the currently reserved resources are stored. The IDRM is located at the BB. In order to achieve this the IDRM maintains four databases:

1. **The Domain Topology Database** which contains the connection map of the routers in the domain and is considered to be static.
2. **The QoS Database** which contains the QoS parameters associated with each PHB. This can be considered static information in the sense that it is updated only at network configuration time.
3. **The Pipe Database** which stores the state of the pipes established between each Ingress and Egress router, for each PHB. This database is dynamically updated.
4. **The Link State Database** which maintains the class-based (PHB-based) QoS state information of all the routers in the form of $<$ *interface, IP, PHB, total capacity, current traffic rate, cost* $>$. This database is also update dynamically.

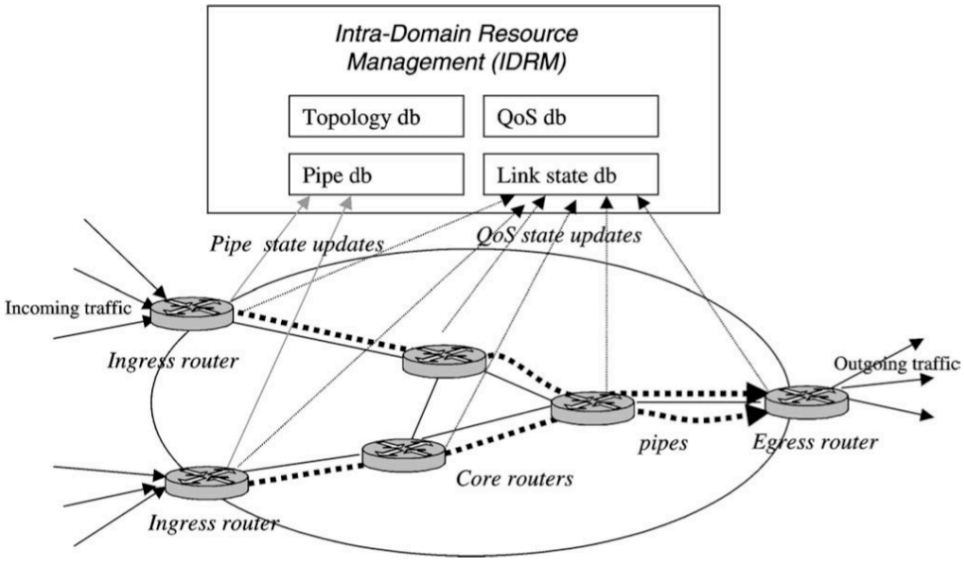


Figure 2.8: The Intra-Domain Resource Manager (IDRM).

Managing network resources within the Bandwidth Brokers domain is now just a matter of keeping these database (mainly the pipes and link databases) up to date. Mantar et al. [23] propose a *pre-established pipe* model in which the pipes between every possible Ingress-Egress router pair, for each PHB, is determined at configuration time. Unlike Wang et al. [36], described earlier, no explicit bandwidth reservation state is stored. Instead the IDRM maintains the reserved and available bandwidth for each pipe in its database. Admission control is done based on the information stored in the database and thus, there is no need to reserve resources along the forwarding path. The IDRM can dynamically resize the pipes without reflecting these changes in the forwarding path.

For the maintenance of the pipe a utilization based update scheme is proposed. The size of the pipe is maintained as follows: whenever the utilization of any pipe exceeds its utilization target (i.e., 90% of its size) then the QoS state information of all links on the path is obtained from the link state database and the appropriate pipe size is determined. Frequent changes of the pipes require frequent access to the link state database which could result in scalability problems. Scalability issues could be resolved by dividing the domain in smaller subdomains, which are all controlled by separate IDRM. This reduces the amount of pipes and thus the frequency of accessing the pipe database. Another point worth noting is that when flows are aggregated on to a single pipe, the fluctuations are smoothed out and the dynamics of the aggregate traffic is much lower.

This pre-established model will work well for networks with mutually disjoint pipes but could lead to serious under-utilization for networks that have multiple pipes sharing one link. For example, lets assume that two pipes, P1 and P2, each having a size of 5Mbps, share a link of 10Mbps. Assume now that after some time the traffic rate of P1 drops to 3Mbps and that there is a new reservation request for a flow corresponding to P2. The IDRM would not know that the bandwidth of P1 is available and thus would deny the request. In order to minimize this problem the concept of virtual capacity is introduced. Virtual capacity corresponds to the share of link capacity, for pipes that share the link, assigned to a certain pipe. The virtual capacity vc_{ik} of pipe P_i on link k is computed as follows:

$$vc_{ik} = c_k \frac{P_i^{cl}}{\sum_j^m P_j^{cl}} \tag{2.7}$$

where m pipes share a bottleneck link k with capacity c_k and P_i^{cl} denotes the current traffic load of P_i. For every link in the network the virtual capacity is computed and stored in a virtual rate allocation

matrix (VRAM) where a column represents a link and a field in the column represents the virtual capacity of a particular pipe on the link. Now whenever a pipe reaches it maximum size the pipe resizing process is started. During this process the VRAM is update and the unused link capacities redistributed among the pipes, sharing the same links, that require additional resources. It should be noted that this pipe resizing process is a costly one and should not be done frequently.

Inter-domain signaling

For inter-domain signaling the Simple Inter-domain Bandwidth Broker Signaling (SIBBS) protocol was designed. The SIBBS protocol was developed by the Qbone Signaling Team[2] and is another example of the pipe model. This protocol defines how resources reservation between Bandwidth Brokers, in BB-supported DiffServ networks, can be accomplished.

Aggregation in this architecture is done between Bandwidth Brokers. Here again the pipes model is used. By aggregating the individual reservations between two BBs into an existing pipe, network scalability can be improved in terms of the signaling and state load and admission control time (compared with IntServ/RSVP model). The solution proposed by Mantar et al. [23] works as follows:

- The source domains BB pre-establishes pipes to every other possible destination domain and then multiplexes all the reservation requests (initiated by end hosts) that have the same destination domain and QoS class into the same pipe.
- If a new flow wants to cross the two domains is will be added to the corresponding inter-domain pipe. Before the flow is added, an admission control check is preformed. If the pipe has enough available resources for the flow, it will be admitted to the pipe.

  Note that at this point the flow has already passed the *intra-domain* admission control check. If the requesting flow did not pass this check then it (or more specifically the traffic belonging to the flow) cannot depart the domain.
- From time to time aggregate traffic in the inter-domain pipes are checked whether the aggregate traffic has reached a certain utilization target. If the aggregate traffic has crossed the boundary of the utilization target, the size of the pipe should be updated. The pipe size can either be increased or decreased. Decreasing the pipe occurs instantaneous. This is because the BB issuing the decrease does not have to perform a check for available resources. The resources can thus be release immediately and the BB at the other end of the pipe is directly notified of the change in pipe size. The increasing of the pipe size is a different story. When a pipe needs to be increase, a request is issued for extra bandwidth. This request is sent from one BB to another which performs the admission control check. If the request is granted then a response is sent back to the requesting BB and it is then that the pipe size is increased. Thus the pipe is increased only after a successful response has been received for the extra bandwidth request. The period between the initiation of the request and the completion of the request is called the *reservation latency period*.

In the next section the details of the inter-domain pipe update scheme is explained.

Dynamic Provisioning Algorithm

Mantar et al. [22] have defined a dynamic provisioning algorithm (DPA) which is used to dynamically reserve and release bandwidth for the aggregate reservation. The DPA determines when a BB should modify the aggregate reservation size and how to modify it (i.e., how much to reduce, how much to increase). Two essential issues that have been considered are inter-BB signaling scalability and efficient resource utilization.

The DPA modifies the reservation rate according to a simple threshold-based scheme. The following parameters are used:

[2]The authors: Haci A. Mantar, Junseok Hwang, Ibrahim T. Okumus, and Steve J. Chapin are all members of the Qbone Signalling Team.

- **Outgoing reservation rate** R_{out}: this parameter represents the reserved rate of a Bandwidth Broker's outgoing interface.
- **Instantaneous outgoing traffic rate** R_{curr}: this parameter represents the aggregate traffic rate passing through the Bandwidth Broker's outgoing interface.
- **High threshold** HT: this is the utilization level where the Bandwidth Broker is triggered to increase the outgoing reservation rate.
- **Low threshold** LT : this is the utilization level where the Bandwidth Broker is triggered to reduce the outgoing reservation rate.
- **Operation region** $OR = HT - LT$.

The algorithm dynamically checks if the current traffic rate is within the OR. As long as the traffic rate fluctuates within the OR, no negotiation takes place. Once the traffic rate crosses the boundaries (HT, LT), the algorithm predicts the new width of the OR, and then triggers the BBRP to negotiate resources with the provider BB.

Here, the width of the OR is critical in terms of the tradeoff between resource utilization and the frequency of BBRP invocation. To maintain the balance, the OR width is chosen by taking previous, current, and future traffic demand into account. For simplicity, similar to the mechanism used by Jacobson for estimating TCP round-trip time [19], the DPA uses the first order of autoregressive integrated moving average (ARIMA). The idea here is to make the OR width adaptive to traffic characteristics. Adapting to the traffic characteristics is done by predicting the length of the next inter update period. The calculation for this length is done as follows:

$$T_{next} = \alpha T_{curr} + (1 - \alpha) T_{prev} \tag{2.8}$$

The parameters T_{curr} and T_{prev} are defined as follows:

- Let times $t_n, t_{n-1}, .., t_1$ denote the times at which the pipe size was updated. Assume that $t = t_n$ then T_{curr} and T_{prev} denote the current and previous inter update time respectively and are computed as follows:

$$T_{curr} = t_n - t_{n-1} \tag{2.9}$$
$$T_{prev} = t_{n-1} - t_{n-2} \tag{2.10}$$

Once the prediction of the next interval (T_{next}) has been calculated using Equation (2.8) then we can calculate the next value for our OR:

- Let T denote the *expected pipe size modification period* or *expected inter update period*. The operation region (OR) is computed using the following equation:

$$OR = \frac{T}{T_{next}} OR \tag{2.11}$$

The new size of the pipe is calculated as follows:

$$R_{out} = R_{curr} + \frac{OR}{2} \tag{2.12}$$

where R_{out} is the outgoing reservation rate, R_{curr} the instantaneous outgoing traffic rate and OR the operation region as defined above. Once the new size of the pipe has been calculated the bandwidth needs to be released or reserved at the BB of the neighbouring domain. If bandwidth needs to be released then this is done instantaneous. For the negotiation of the reservation of extra bandwidth the BBs

are constrained by the reservation latency. During the reservation latency period the pipe can not be updated. This means that when a flow wants to join a particular pipe which is being updated and there are not enough available resources in the pipe, the flow is simply rejected. To avoid having to reject flows during these flows Mantar et al. [22] propose the use of a '*cushion*'. The cushion is a spare amount of bandwidth which is reserved specifically for the reservation latency delay. Thus during a pipe update flows are admitted as long as the cushion has enough spare bandwidth. An appropriate size for the cushion has not been given in [22].

2.2 Proposed Solution

The solution proposed in this paper is based on the pipes model and the update algorithm proposed by Mantar et al. [23]. The reason why this algorithm is preferred over the other alternatives is because, aside from the solution proposed by Schmitt et al. [32], all the other alternatives discuss the aggregation of flows with a fixed bandwidth requirement. In these studies [24, 36, 28] the emphasis lies on the arrival process of flows and the prediction or estimation of future flow arrivals or departures. The solution proposed by Schmitt et al. [32] is rather simple and as stated in [32] should be regarded as the illustrative example. In our opinion the algorithm proposed by Mantar et al. [22, 23] is much more sophisticated and takes the dynamics of the underlying aggregate traffic rate into account. This solution also takes the importance of the reservation latency between Bandwidth Brokers into account with their use of a 'cushion'.

2.2.1 Aggregate Update Algorithm

At configuration time pipes for every Ingress-Egress pair in the domain should be created. The initial value of the capacity should be set to a reasonable value such that it is possible for several flows to join the aggregate. The value of the pipe represents the size of the aggregate reservation, C_a. We need to update the capacity of the pipe according to the dynamics of the aggregate traffic rate $r_a(t)$. Updating should occur whenever the pipe is nearly full. Thresholds are used to trigger an increase or a decrease of a pipe. There are two thresholds that need to be set, the high threshold δ_{high} and the low threshold δ_{low}. As long as the aggregate traffic rate lies between the high and low thresholds no update is required. An increase of the aggregate reservation size is triggered whenever the aggregate traffic rate exceeds the high threshold δ_{high}. In this case the new size for the aggregate reservation needs to be computed and the additional bandwidth reserved. A decrease of the aggregate is triggered whenever the aggregate traffic rate drops below the low threshold δ_{low}. In this case the new size for the aggregate reservation is computed and the excess bandwidth released. In Figure 2.9 the threshold based aggregate update algorithm is displayed.

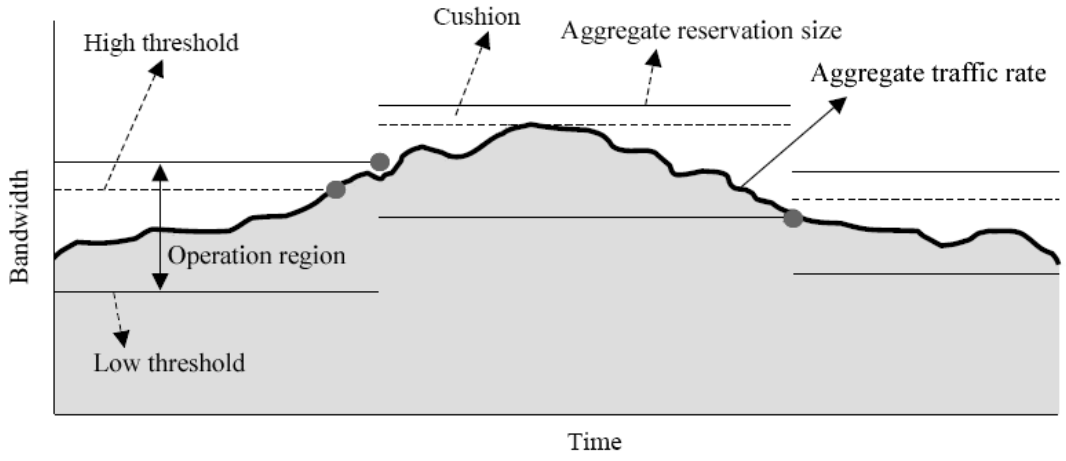


Figure 2.9: The aggregate update algorithm.

Setting the thresholds and the calculation of the new aggregate reservation size is done according to the update algorithm proposed by Mantar et al. [22]. First we will discuss how the new aggregate reservation size is computed and then we will discuss how to set the thresholds δ_{high} and δ_{low}. Note that there is a small difference between our definition of the operation region and definition used by Mantar et al. [22]. Mantar et al. [22] define the region between the high and low threshold as the *operation region* (OR). In our algorithm we define the OR as the region between the maximum capacity and the low threshold. The reason why our definition of the OR is different from Mantar et al. [22] is because our algorithm will use a dynamic cushion instead of the fixed size cushion used in Mantar et al. [22].

Let times $t_n, t_{n-1}, .., t_1$ denote the times at which the pipe size was updated. Assume that $t = t_n$ then T_{curr} and T_{prev} denote the current and previous inter update time respectively and are computed as follows:

$$T_{curr} = t_n - t_{n-1} \tag{2.13}$$
$$T_{prev} = t_{n-1} - t_{n-2} \tag{2.14}$$

Let T denote the expected pipe size modification period or expected inter update period. In our algorithm T is a parameter which should be set by the network administrator. In [15] the operation region (OR) is computed using the following equation:

$$OR = \frac{T}{\alpha T_{prev} + (1-\alpha) T_{curr}} OR \tag{2.15}$$

We note that in [15] exponential averaging is said to be used in order to determine the size of the OR. However, from the notations defined above the exponential averaging is done only over the measured size of the current and previous inter update interval. The recursion of the OR also results in some form of averaging but it is unclear whether these calculations are equivalent to the exponential moving average. We can define the estimation of the inter update time by explicitly using the exponential weighted moving average:

$$T_{av} = \alpha T_{av} + (1-\alpha) T_{curr} \tag{2.16}$$

where T_{av} is the average inter update time computed using an exponentially weighted moving average with smoothing factor α. The calculation of the OR is now done as follows:

$$OR = \frac{T}{T_{av}} OR \tag{2.17}$$

The difference between Equation (2.15) and (2.17) is that the former uses a smaller history and relies more on the recursion of the OR than the latter equation. This makes Equation (2.15) more adaptive to changes in the aggregate traffic rate. The latter equation has a larger history and is thus less vulnerable to fluctuation in aggregate traffic rate. Which computation is better depends on the dynamics of the aggregate traffic rate. For the computation of the OR in our algorithm we will use Equation (2.17). This choice is mainly motivated by the vulnerability to oscillations of Equation (2.15). Because Equation (2.15) uses only the previous and current measured update interval it might be possible that the calculated average $(\alpha T_{prev} + (1-\alpha) T_{curr})$ oscillates.

The new size of the pipe is calculated as follows:

$$C'_a(t_n) = r_a(t_n) + \frac{OR}{2} \tag{2.18}$$

where $C'_a(t)$ is new aggregate reservation size that needs to be reserved. Note that there is a difference between $C'_a(t)$ and $C_a(t)$. $C_a(t)$ denotes the actual aggregate reservation size at time t and $C'_a(t)$ denotes the new aggregate reservation size which still needs to be reserved. Due to the reservation latency we can not set the new reservation size and thresholds immediately. We have to wait for a confirmation that the

extra bandwidth has been reserved by the interior routers. The new values for the aggregate reservation size and thresholds can only be set if this confirmation has been received (at the Ingress router). If the reservation for the new size is granted then $C_a(t)$ is set to $C'_a(t)$.

Once the new aggregate reservation size ($C_a(t)$) is set the lower threshold can be set. The new value for the low threshold is calculated as follows:

$$\delta_{low} = C_a(t) - OR \tag{2.19}$$

The computation of the high threshold (δ_{high}) is a little more complicated because it depends on the value of the 'cushion' (see Figure 2.9). The high threshold is set at:

$$\delta_{high} = C_a(t) - \Delta_{cushion} \tag{2.20}$$

The computation of the cushion is discussed in the next section.

2.2.2 The Cushion

In Mantar et al. [22] the notion of using a 'cushion' is proposed in order to accomodate flows that arrive during the pipe update process. Yet no computation for the appropriate size of the cushion is given. Our proposed solution will attempt to find an appropriate value for the cushion, $\Delta_{cushion}$, by calculating the following probability:

$$P\{r_a(t') > C_a(t)\} \leq \epsilon \tag{2.21}$$

where t' denotes the current time plus the reservation latency delay ($t' = t + \Delta t_{rl}$), $r_a(t')$ is the future aggregate traffic rate at the end of the reservation latency period, $C_a(t)$ is the current aggregate reservation size and ϵ is a small error term. The error term ϵ should be set by the network administrator. It defines the probability of flows that may be rejected. If ϵ is set to a high value, the cushion will be more relaxed in the rejection of flows. Setting ϵ low will make the cushion more sensitive to flow rejections. In this case the value of the cushion will be large in order to cope for a larger number of flows arriving during the update of the aggregate reservation. The maximum value of the cushion however is constrained by the size of the operation region (OR) in the following way:

$$\Delta_{cushion} \leq \frac{OR}{2} \tag{2.22}$$

In other words the cushion can never be larger than half the size of the operating region. If the cushion is larger than $\frac{OR}{2}$, the aggregate traffic rate might come to be larger than the high threshold without an update being triggered.

What we want to compute is the size of the cushion ($\Delta_{cushion}$) such that Equation (2.21) holds. This equation needs to be rewritten so that the cushion can be calculated explicitly. The cushion can be calculated as follows:

$$\Delta_{cushion} = C_a(t) - r_a(t) \tag{2.23}$$

where $C_a(t)$ is the aggregate reservation size at time t and $r_a(t)$ is the aggregate traffic rate at time t. Equation (2.21) states that the future aggregate traffic should not exceed the current aggregate reservation size with high probability. We can rephrase this as follows: the difference between the current aggregate traffic rate ($r_a(t)$) and the future aggregate traffic rate ($r_a(t + \Delta t_{rl})$) should be smaller than the size of the cushion with high probability. This can be expressed with the following probability:

$$P\{r_a(t') - r_a(t) \geq C_a(t) - r_a(t)\} \leq \epsilon \tag{2.24}$$

Substituting (2.23) gives the following:

$$P\{r_a(t') - r_a(t) \geq \Delta_{cushion}\} \leq \epsilon \tag{2.25}$$

As stated in the problem definition only constant bitrate (CBR) flows are considered in our solution. We simplify the problem further by defining flows to be of one particular type. This means that the bandwidth requirement for all flows is constant. As a result we can now easily calculate the amount of bandwidth needed for a particular number of flows. Thus for the equation above we can easily convert the size of the cushion to the number of flows that fit in the cushion. The aggregate traffic rate $r_a(t)$ and the future aggregate traffic rate $r_a(t')$ can also be translated to the number of flows currently in the aggregate and the future number of flows in the aggregate respectively. Because of the one-on-one mapping between aggregate traffic rate and the number of flows we can simplify Equation (2.25) to a counting process:

$$P\{N(t') - N(t) \geq k\} \leq \epsilon \tag{2.26}$$

where $N(t')$ describes the number of flows in the aggregate at time t', $N(t)$ describes the number of flows in the aggregate at time t and k is the number of flows that fit in the cushion. Note that the parameters in the equation above are simply the number of flows equivalents of the rates in Equation (2.25). Now let x_{bw} be the bandwidth requirement for one flow. Because the rate of the individuals flows are all equal we can calculate the number of flows in the aggregate at a certain time t as follows:

$$N(t) = \frac{r_a(t)}{x_{bw}} \tag{2.27}$$

In a similar fashion we can calculate the number of flows that fit in the $\Delta_{cushion}$:

$$k = \frac{\Delta_{cushion}}{x_{bw}} \tag{2.28}$$

In other words if we want to determine an appropriate value for $\Delta_{cushion}$ we must calculate the value of k. Before we can calculate the value of k we will make the following assumption:

- The arrival process for flows can be described as a Poisson process. Poisson models are typically used to describe in telephony model to describe the call-arrival process. In these models calls are assumed to arrive according to a Poisson distribution and their lifetime is assumed to be exponentially distributed. These assumptions will also be used in our proposed solution.
- The length of the reservation latency period is very small and during this period no flows leave the aggregate. This assumption is made in order to simplify the calculation of the cushion. Namely, if no flows leave the aggregate we simply have to calculate the amount of bandwidth needed to cope for the arriving flows during the reservation latency period. Flows leaving the aggregate during this period are not considered. For small values of the reservation latency this assumption might hold but for large value of the reservation latency period this certainly is not true. Due to this assumption our calculation of the cushion size is slightly pessimistic.
- The rate parameter λ of the Poisson process does not change during the reservation latency period. The fluctuations of the aggregate traffic rate on a large time scale are caused by the change in the intensity of the flow arrivals. This intensity is represented by the rate parameter λ. Increasing λ causes the (average) number of flows in the aggregate to grow while decreasing λ causes the (average) number of flows in the aggregate flow to shrink. This process however occurs on a large time scale. For the calculation of the cushion however a small time scale is considered and thus we can safely assume that the rate parameter λ remains constant during the reservation latency period.

What we are interested in is the (extra) number of flows that join the aggregate during the reservation latency period Δt_{rl}. Thus how many *more* flows join the aggregate during a certain period of time τ compared to the flows that leave the aggregate during this period τ. Let $\tilde{N}(\tau)$ be the difference between the number of flows in the aggregate at the beginning of period τ and the number of flows in the aggregate at the end of period τ. In one of the assumptions above we state that during the reservation latency

period Δt_{rl} no flows leave the aggregate. In this case $\tilde{N}(\tau)$ corresponds to the number of flow arrivals during the period τ. Furthermore we have assumed that the flow arrival process can be modeled as a Poisson process. A Poisson process is characterized by a rate parameter λ such that the number of events in time interval $(t, t+\tau]$ follows a Poisson distribution with associated parameter $\lambda\tau$. Given these assumption we can write this relation as follows:

$$P\{\tilde{N}(\tau) = i\} = \frac{e^{-\lambda\tau}(\lambda\tau)^i}{i!} \tag{2.29}$$

Note that the equation above computes the probability that there are *exactly* i arrivals during time interval $(t, t+\tau]$. If we want to compute the probability that there are *more than* i arrivals during time interval $(t, t+\tau]$ we will have to use the following equation:

$$P\{\tilde{N}(\tau) > i\} = \sum_{j=i+1}^{\infty} P\{\tilde{N}(\tau) = j\} \tag{2.30}$$

What we want to compute is the number of arrivals (k) within time interval Δt_{rl} such that the probability that more flows that k arrive in this time interval is very small. In other words, we want to compute the probability:

$$\begin{aligned} P\{\tilde{N}(\tau) > k\} &= \sum_{i=k+1}^{\infty} P\{\tilde{N}(\tau) = i\} \\ &= \sum_{i=k+1}^{\infty} \frac{e^{-\lambda\tau}(\lambda\tau)^i}{i!} \\ &= 1 - \sum_{i=0}^{k} \frac{e^{-\lambda\tau}(\lambda\tau)^i}{i!} \end{aligned} \tag{2.31}$$

With the equation above and a set of values for λ, Δt_{rl} and ϵ we can now calculate k. This can be done by using a counter variable k_i which is initialized to 0. Substituting Equation (2.26) and Equation (2.31) gives:

$$\begin{aligned} P\left\{\tilde{N}(\Delta t_{rl}) > k\right\} &\leq \epsilon \\ 1 - \sum_{i=0}^{k} \frac{e^{-\lambda\Delta t_{rl}}(\lambda\Delta t_{rl})^i}{i!} &\leq \epsilon \\ \sum_{i=0}^{k} \frac{e^{-\lambda\Delta t_{rl}}(\lambda\Delta t_{rl})^i}{i!} &\geq 1 - \epsilon \end{aligned} \tag{2.32}$$

We need to compute the minimum value for k such that the sum $\sum_{i=0}^{k} \frac{e^{-\lambda\Delta t_{rl}}(\lambda\Delta t_{rl})^i}{i!}$ is greater than or equal to $1-\epsilon$. This can be done by computing the summation defined in above and check whether it is smaller than $1-\epsilon$. If this is larger than $1-\epsilon$, k is incremented and the probability computed again. This is repeated until the summation is greater than $1-\epsilon$. In this case we have found the minimum value for k. Note that as an additional constraint we have defined that the size of the cushion can never exceed $\frac{OR}{2}$ (see Equation (2.22)). As a result we can apply the following constraint to the calculation of k:

> The maximum value for k is $\frac{OR}{2 \times x_{bw}}$. This is because the cushion is constrained by the size of the operation region. Because $\Delta_{cushion}$ can never be greater than $\frac{OR}{2}$ and $\Delta_{cushion} = k \times x_{bw}$ this result in the maximum value for k being $\frac{OR}{2 \times x_{bw}}$

Once the correct value for k has been found, the size of the cushion can be computed as follows:

$$\Delta_{cushion} = k \times x_{bw} \tag{2.33}$$

Parameter Estimation

In order to calculate the value of the cushion we need to estimate the parameters λ and the reservation latency Δt_{rl}. The parameter λ is estimated by counting the number of arrivals over small intervals. The length of these intervals is set to 5 minutes. The reason why λ is recalculated every five minutes is because the intensity of arrivals fluctuates over time. For example, the (average) number of arrivals at night is much lower compared to the (average) number of arrivals during business hours.

Estimating the reservation latency Δt_{rl} is done using a moving average. The exponentially weighted moving average is used due to its accuracy and simplicity. Δt_{rl} is estimated as follows:

$$\Delta t_{rl} = \alpha_{rl} \Delta t_{rl} + (1 - \alpha_{rl}) \Delta t_{curr} \tag{2.34}$$

where α_{rl} is the smoothing factor and Δ_{curr} is the current measured latency window.

2.2.3 Scenario's

Maximum capacity reached

Up till now we have assumed that our algorithm always gets the bandwidth it requests. This is not always the case though. Due to congestion or the maximum capacity of a link being reached the request for additional might be denied by an interior router. In our algorithm a new request for additional bandwidth is sent immediately after the previous request was denied. This process is repeated until the additional bandwidth is reserved. The probability that the new request will be denied is rather high and therefor the amount of control messages sent is also rather high which is not desired. In order to address this problem we propose a minimum waiting period before another request can be sent. This can be implemented in two ways:

1. A back-off timer can be used which is triggered whenever a negative acknowledgement is received.
2. A minimum waiting time timer can be used which is triggered whenever a request is sent. The timer denotes the minimum waiting time between to requests.

The difference between the two approaches is that the first is only used when a request is denied while the latter specifies the minimum time between two consecutive reservation requests regardless of a request being granted or rejected. For our algorithm we will use the latter approach because of its simplicity. The parameter T_{min} denotes the minimum waiting time between two consecutive requests.

Congestion

Another scenario which we need to address is the case of congestion. When the network is congested the aggregate reservation needs to be decrease in order to resolve the congestion. In RMD there are mechanisms designed to notify edge routers of the congestion and the degree of congestion. Based on the degree of congestion the aggregate reservation size is decreased. This is done by terminating several end-to-end flows until the desired size is reached. The selection of flows that are to be terminated can be done in a random fashion or by terminating the oldest flows first.

For the purpose of this assignment we will not address the above mentioned, because the current implementation of the NSIS signaling protocol does not support congestion notification. In order to test how our prototype performs in the face of congestion we would first need to implement the congestion notification feature.

Chapter 3

NSIS: Overview

In the previous chapter we have proposed an aggregate update scheme which can be used to balance the number of updates and the utilization. To test the performance of our aggregate update scheme, an implementation of the Next Step In Signaling (NSIS) framework, developed at Twente University, is used. This prototype implementation was developed by Martijn Swanink and Ruud Klaver. It supports the reservation of bandwidth for uni-directional flows but does not support the aggregation of end-to-end flows. So before we can use the current implementation, we must extend its functionality to support the aggregation of flows.

In this chapter we will give an introduction to the NSIS framework. For clarification reasons some parts of the text in this Chapter have been copied from [16, 18, 33]. Here we will describe the components of the NSIS framework. We will also show how these components can be used to setup and maintain reservation states. Furthermore we will take an in depth look at QoS Models (QoSMs) and QoS Specifications (QSpecs).

3.1 Introduction

The NSIS working group is currently working on standardizing an (IP) signaling protocol. This signaling protocol can be used by many different types of applications (e.g., signaling for middleboxes, firewalls or QoS) but for the purpose of this assignment we will narrow our scope to the signaling of Quality of Service (QoS). QoS refers to resource reservation control mechanisms. It can provide different priorities to different users or data flows, or guarantee a certain level of performance to a data flow in accordance with requests from the application program. In the NSIS framework, described in RFC 4080 [16], a data flow is defined as *a stream of packets from sender to receiver that is a distinguishable subset of a packet stream.* We will be using the following five tuple to identify a flow: *<source IP address, source port, destination IP address, destination port, transport protocol>*.

In order to provide QoS guarantees for a data flow we need to reserve resources, like bandwidth, at the routers that the flow traverses. The reservation of resources for this data flow involves the setup and maintenance of reservation states. In this state information regarding the amount and type of resources are stored. This state is related to a data flow and signaling messages can be used to install, modify, refresh or simply read this state from network elements for a particular flow.

3.2 Signaling scenario

In Figure 3.1 a NSIS signaling scenario is displayed. Here is the path of the data flow, which traverses three routers, and its signaling session between the *sender* and *receiver* is shown. The red arrow indicates the direction of the data flow and the yellow arrows indicate the QoS signaling interactions. At the end hosts and the intermediate routers are network entities (NE) installed, which represent '*the function within a node that implements an NSIS protocol*' [16]. In other words, the NE is responsible for the processing of NSIS signaling messages according to the NSIS protocol. Because we are only concerned

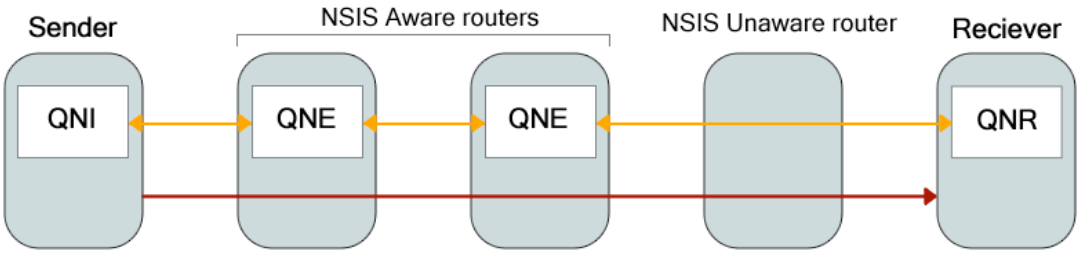


Figure 3.1: A NSIS signaling scenario

with QoS signaling we will refer to these NEs as QNEs. A QNE is '*an NSIS Entity (NE), which supports the QoS NSLP*' [18]. Here NSLP stands for the *NSIS Signaling Layer Protocol* which we will elaborate in section 3.3.2.

Figure 3.1 shows the distinction between NSIS aware and NSIS unaware routers. While the NSIS unaware routers do not have an active role in the resource reservation for a particular flow, they still forward the signaling messages to and from NSIS aware routers. These messages are not processed by the NSIS unaware routers because these routers do not have an QNE.

In a signaling session QNE's can assume different roles. In Figure 3.1 all possible roles of a QNE are shown. We will name the QNE based on the role it is in. For the NSIS aware routers the QNE can be one of the following entities depending on their role:

QoS NSIS Initiator (QNI): the signaling entity that makes the resource request, usually as a result of user application request. It is the first node in the sequence of QNE's that has issued the reservation request.

QoS NSIS Responder (QNR): the signaling entity that acts as the endpoint for the signaling and can optionally interact with applications as well.

QoS NSIS Forwarder (QNF): a signaling entity between a QNI and QNR that propagates NSIS signaling further through the network. In this thesis we will not explicitly refer to the QNE's of intermediate routers as QNF's. Instead we will we simply state that these routers have a QNE because from the context it is clear that these entities are not initiators nor responders and thus are only responsible for the forwarding of signaling messages.

In Figure 3.1 it is easy to see that the signaling entity at the end host *sender* is a QNI and the one at the end host *receiver* is a QNR. The intermediate NSIS aware routers - which have QNE's installed - are QNF's. Note that the role of the QNE is associated to a particular flow. At one point a QNE can be the initiator of a particular reservation while at the same time being a forwarder for other flows.

3.2.1 Internal structure QNE

A representation of the internal structure of a QNE is given in Figure 3.2. Here the different components that are involved in providing QoS for a particular flow are shown. First there is the '*local application*' that interacts with the QNE triggering the setup or tear down of a reservation. This local application can also be an management application for e.g. aggregates. Then there is the QNE which is subdivided in the following components:

- *QoS NSLP Processing*: responsible for the processing of QoS messages received from the lower transport layer. The NSIS framework has its own signaling transport layer, called the '*NSIS Transport Layer Protocol*' (NTLP), which is discussed in section 3.3.1.

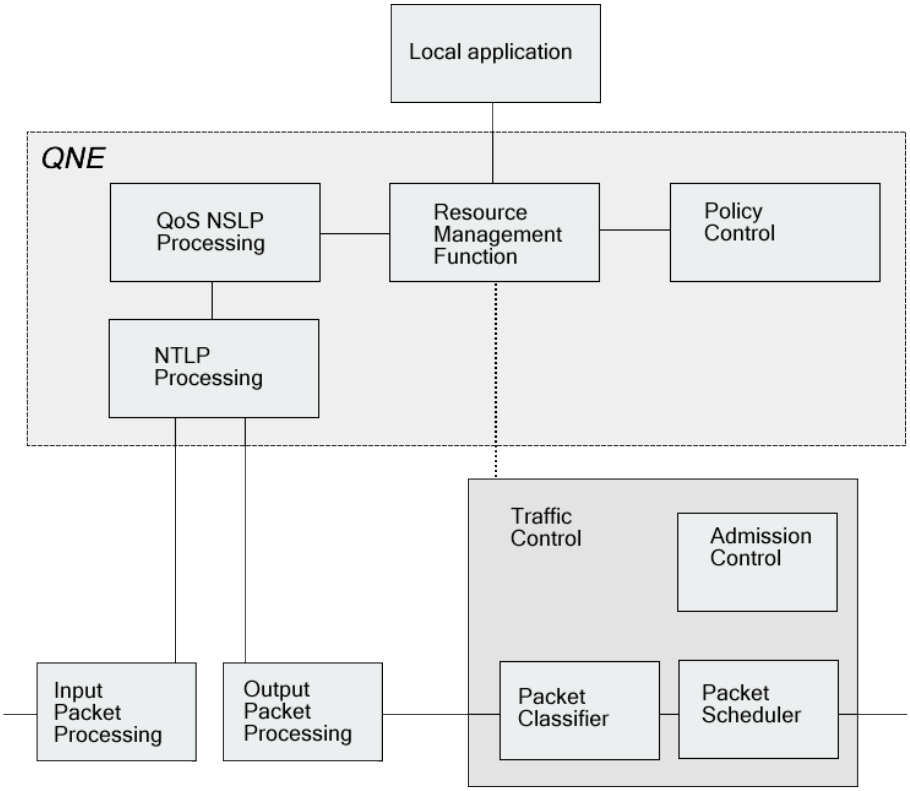


Figure 3.2: An internal representation of a QNE.

- *NTLP Processing*: responsible for node discovery and routing.
- *Resource Management Function (RMF)*: responsible for the actual granting of reservation requests and/or configuration of resources.
- *Policy Control*: the RMF is connected to a policy control component which determines whether the application or its user is authorized to query resources or make an actual reservation. In our prototype implementation we will not implement this component.

The other components shown in Figure 3.2 are lower layer components that are responsible for networking aspects:

- *Input packet processing*: this component handles the node's incoming messages. Only QoS related messages are passed to the NTLP processing component.
- *Output packet processing*: here outgoing messages are processed and actual packet forwarding is done.
- *Traffic Control*: this component controls the flow data packets. It is subdivided in the following components:
  - *Packet Classifier*: as the name states, this component is responsible for the classification of packets. Packets are classified and subsequently prioritized before being sent across a network. Using this mechanism the traffic control can determine the flow for any packet, and thus, the treatment that the packet receives. Once a packet has been classified as belonging

to a particular flow, the *Packet Scheduler* is able to treat it in accordance with that flow's parameters.

- *Packet Scheduler*: The Packet Scheduler enforces QoS parameters for a particular flow. The scheduler retrieves the packets from the queues and transmits them according to the QoS parameters, which generally include a scheduled rate and some indication of priority. The scheduled rate is used to pace the transmission of packets to the network. The priority is used to determine the order in which packets need to be submitted to the network when congestion occurs. This smoothes bursts or peaks of traffic over a period of time, thereby effecting a steadier use of the network and maintaining resource integrity.
- *Admission Control*: this component determines whether a flow can be granted without disrupting any established flows in the network.

3.3 Protocol stack

The NSIS protocol provides functionality to routers which allow them to manage their resources, like bandwidth. In these routers additional layers have been added to the protocol stack, on top of the network layer or the IP layer. In Figure 3.3 the protocol stack of an NSIS aware router is shown.

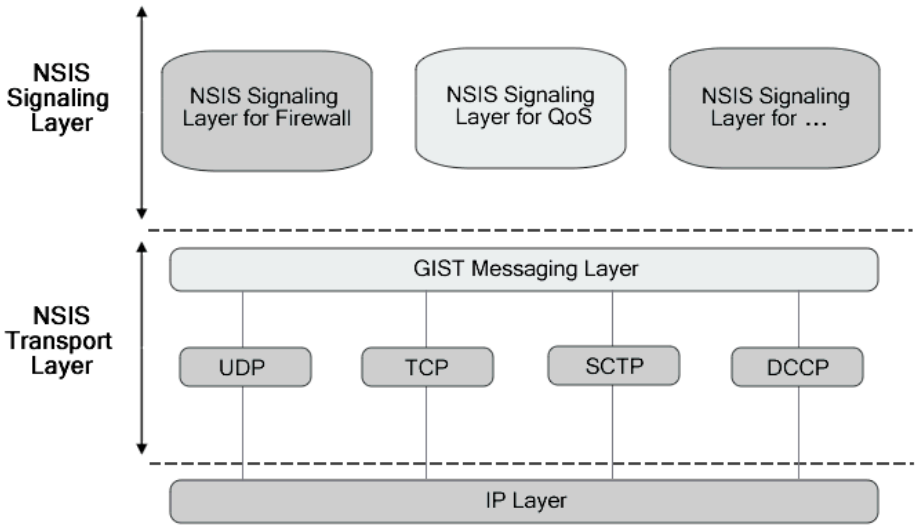


Figure 3.3: The NSIS protocol stack

In Figure 3.3 we see that the NSIS protocol is divided into two layers. This is done to achieve a modular solution. The functionality of these layers are defined as follows:

- the 'signaling transport' layer is responsible for moving signaling messages around, which should be independent of any particular signaling application. From here on forward we will refer to the components of the transport layer as the 'NSIS Transport Layer Protocol', denoted by NTLP.
- the 'signaling application' layer is responsible for the setup, maintenance and tear down of the reservation states. It contains functionality, such as message formats and sequences specific to a particular signaling application. The term 'NSIS Signaling Layer Protocol' (NSLP) refers to any protocol within the signaling application layer.

3.3.1 NSIS Transport Layer Protocol (NTLP)

The NTLP is the lower transport layer of the signaling layer (NSLP). Both layers form the basis of the overall signaling solution and therfore must coexist with each other. The NTLP layer is responsible for the routing and delivery of the signaling messages, while the NSLP is responsible for the 'end-to-end' issues like message retransmissions and failure notification. Note that the NTLP cannot provide end-to-end guarantees because the NTLP interactions occur only between adjacent QNE's, making it a 'hop-by-hop' protocol. As a result, larger scope aspects, like the 'end-to-end' issues, are left up to the NSLP. The NTLP works as follows:

- When a signaling message is ready to be sent by a NSLP, it passes the message along with flow information to the NTLP. Note that the NTLP does not have any knowledge of the purpose of the message being send. It is unaware of the fact that the signaling message is used to setup, update, tear down or just refresh the network control state.
- The NTLP processes the received message and sends this message immediately to the next QNE along the path (this can be either upstream or downstream). It is important to note that the NTLP does not change the sequence of the messages. Because all messages are sent immediately, their timing cannot be jittered at the routers nor are messages stored up to be re-sent.
- When the message is received by the adjacent node the responsibility of the NTLP ends. Because of this, the NTLP has no knowledge of the addresses, capabilities, or status of any QNE other then its direct peers.
- The receiving QNE processes the received message and can either forward the received message or pass it to the signaling layer for further processing.

Essentially the NTLP is just an efficient upstream and downstream peer-to-peer message delivery service, where message delivery includes '*the act of locating and/or selecting which NTLP peer to carry out signaling exchanges with for a specific data flow*' [16]. Thus the NTLP takes care of the transport and routing of the signaling messages. For the actual transport of messages existing protocols are used, like for example Stream Control Transmission Protocol (SCTP)[34], Datagram Congestion Control Protocol (DCCP)[21], Transmission Control Protocol (TCP)[30] or User Datagram Protocol (UDP)[29] protocols (also see Figure 3.3).

> *In our prototype implementation of the NSIS protocol we only support the usage of the TCP and UDP transport protocols. Optionally the TCP can be used in combination with Transport Layer Security (TLS)[9] to provide an secure (encrypted) transport of the signaling messages.*

On top of these transport layers a common messaging layer is located, the General Internet Signaling Transport (GIST) layer.

General Internet Signaling Transport (GIST)

The GIST layer is responsible for most of the NTLP's functionality. The purpose of this layer is defined in [33] as follows:

> *'The purpose of GIST is thus to provide the common functionality of node discovery, message routing and message transport in a way which is simple for multiple signalling applications to re-use.'* from [33]

GIST allows the NSLP layer to send and receive signaling message to and from other QNEs respectively. This is done using the GIST Application Programming Interface (API). The GIST API specifies how information can be passed on from the NSLP to GIST and vice versa using the API's service primitives. In Figure 3.4 the GIST API is shown.

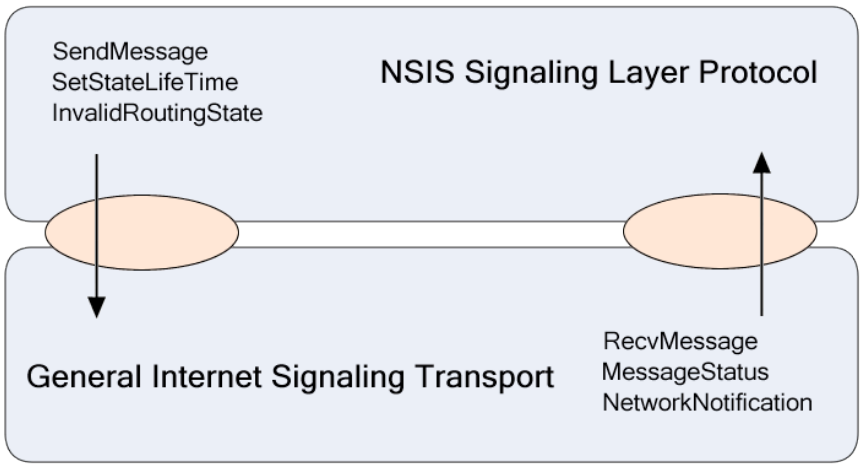


Figure 3.4: The GIST API.

The API has six service primitives, three of which can be used by the NSLP to pass information to GIST and the other three can be used by GIST to pass information to the NSLP. The NSLP layer can interact with GIST using the following primitives:

- *SendMessage*: which allows a NSLP application to send signaling messages to upstream or downstream peers.
- *SetStateLifeTime*: which allow a NSLP application to control how long GIST should retain its routing state.
- *InvalidRoutingState*: which allows a NSLP application to explicitly remove any routing state associated to a particular flow.

GIST can interact with the NSLP player using the following service primitives:

- *RecvMessage*: this service primitive is used to deliver signaling messages received by GIST, including the case of null messages, to a NSLP application.
- *MessageStatus*: this service primitive allows GIST to notify a NSLP application whether a signaling message was send successfully. This service primitive can also be used to inform the NSLP application of the transfer attributes used to send the signaling message. The signaling application can respond to this message with a return code to abort the sending of the message if the attributes are not acceptable.
- *NetworkNotification*: this service primitive can be used to notify a NSLP application of changes in the network status.

We will not discuss the service primitives here, but a more detailed description is given in Appendix A. The routing of the signaling messages is done using the flow's Message Routing Information (MRI). The MRI can be used to describe a flow:

> *'The set of data item values which is used to route a signalling message according to a particular Message Routing Method (MRM); for example, for routing along a flow path, the MRI includes flow source and destination addresses, protocol and port numbers.'* from [33].

Here the MRM is defined as:

'... the different algorithms for discovering the route that signalling messages should take. These are referred to as message routing methods, and GIST supports alternatives within a common protocol framework.' from [33].

The NSIS framework supports 'path-coupled' and 'path-decoupled' MRMs. If path-coupled signaling is used, the signaling messages are routed only through QNEs that are on the data path. In the path-decoupled case, signaling messages are routed to QNEs that are not assumed to be on the data path, but that are (presumably) aware of it. In this case signaling messages will always be addressed directly to the neighbor QNE, and the signaling endpoints may have no relation at all with the ultimate data sender or receiver. In this assignment we will only consider the path-coupled signaling scenario, because this is the paradigm supported by RMD.

In appendix B the bit format of an MRI with a path-coupled MRM is shown. Here we see that a traditional five tuple, which we use to identify a flow, is used to describe a flow: *<source IP address, source port, destination IP address, destination port, transport protocol>*. An important flag in this MRI worth mentioning is the *D-flag*. This flag indicates the direction of the signaling with respect to the flow. If the *D-flag* is set to 0 then the signaling messages are sent in the same direction as the flow. If the flag is set to 1, the signaling messages are sent in the opposite direction. Using this flag a NSLP application can indicate if it wants to send a message to its downstream peer or to its upstream peer.

For the transfer of messages to neighboring peers, GIST has three different modes:

Query Mode (Q-mode): This mode is used to discover downstream peers[1]. In Query Mode a UDP datagram will be sent towards the MRI destination. The IP header of UDP datagram should include a Router Alert Option (RAO), which is an IP option that can be used to notify routers along a path. The RAO is specified in RFC 2113 [20] and is used to:

'... provide a mechanism whereby routers can intercept packets not addressed to them directly, without incurring any significant performance penalty.' from [20].

The bit format of the RAO field is as follows:

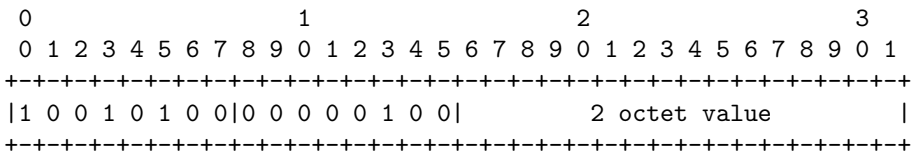


Figure 3.5: The Router Alert Option bit format.

The final 16 bits of the RAO field should contain the NSLP-ID of signaling application on the querying node. This identifier is used by GIST to check whether the message should be associated with the local NSLP. Note that this is the only information visible to GIST about the signaling application being used. If a match is found between the NSLP-ID and a local signaling application, GIST will intercept the message for further processing. At this point GIST will attempt to set up the routing state between the two peers. In the case that no match was found between the NSLP-ID and a local signaling application, the message will be forwarded unchanged.

Datagram Mode (D-mode): This mode is used to send GIST messages between neighboring peers without using any transport layer state or security protection. In Datagram Mode messages are simply sent as UDP datagrams, addressed directly to the GIST node to be reached. The IP destination address of a node can be be derived from nodes previously discovered using the Query, or derived from the flow's MRI.

Connection Mode (C-mode): This mode is used to send messages directly to neighboring peers with the re-use of existing transport and security protocols, where such functionality is required.

[1] GIST can also discover upstream peers but this is not advised. Upstream peers are to be discovered by preceding downstream messages. This way the GIST state is only installed during downstream peer discovery.[31]

In Connection Mode a Messaging Association (MA) 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*' [31]. If no MA is present, it will be setup dynamically. Note that the MA is also stored in soft state, meaning that the connection will automatically be torn down if it is not used for a certain period of time.

We will not discuss the complete routing and peer discovery functionality of GIST, for this the reader is referred to [33] and [31]. Above we have discussed the different transfer modes used by GIST because of their relevance for the RMD QoS model, as we will see in section 4.2.

3.3.2 NSIS Signaling Layer Protocol (NSLP)

The NSLP is responsible for making end-to-end QoS reservation for a particular data flow. This layer uses the services provided by GIST to setup and maintain reservation states at nodes along the path of a data flow. The key features of the NSLP are:

- The NSLP uses **soft state** reservations which need to be refreshed from time to time. If after a pre-defined period of time and no refresh message has been received to maintain the reservation state, the state is removed. Note that NSLP also supports the explicit removal of reservations state.
- The NSLP supports both **sender-initiated reservations**, where the sender of the data flow initiates the reservation, and **receiver-initiated reservations**, where the receiver of the data flow initiates the reservation. **Bi-directional reservations** are also supported by the NSLP, where sender and receiver initiated reservations can be used to signal the initiation, refresh and terminate reservations in both directions, i.e., from sender towards receiver and from receiver towards the sender.
- The NSLP supports **session binding** and **message binding**.

  Session binding is used to indicate a relation between different QoS NSLP sessions which is particularly useful for bi-directional sessions, where the upstream reservation is associated with the downstream reservation, and aggregate sessions.

  Message binding can be used to express the dependencies between different messages. There are some scenarios where two different messages have to be synchronized at a particular QNE. In this case the first arriving message would have to be queued until the other bound message arrives at the node. The messages are processed only after both bound messages have been received.
- The NSLP has mechanisms to protect nodes from **duplicate or re-ordered messages**. The order in which RESERVE messages are received affects the reservation state stored at a QNE. This is because the most recent RESERVE message replaces the current reservation. Therefore, in order to protect against RESERVE message re-ordering or duplication, the NSLP uses a Reservation Sequence Number (RSN).
- Message scoping is supported by the NSLP, where a QNE can decide whether to propagate a message or not. Two scoping flags limit the part of the path over which a message can travel. A SCOPING flag can be used to indicate that the scope is "whole path" or a "single hop". The PROXY scope flag can be set to indicate that the path is terminated at a pre-defined Proxy QNE.
- For limiting the number of individual messages, the NSLP supports a **summary refresh** and **summary tear** messages.
- The NSLP supports **layered reservations**. Layered reservations occur when certain nodes of the network (domains) implement one or more local QoS models. This is particularly true in the case of aggregation where the aggregate reservation uses a different QoSM than the per-flow QoSM.

- The NSLP can **adapt to route changes** in the data path. Whenever rerouting events are detected by the NSLP, a new QoS reservation is created along the new path and optionally the reservations on the old path torn down.
- **Pre-emption** is supported by the NSLP. Note that the NSLP specification does not define how pre-emption should work, but only provides signaling mechanisms that can be used by QoS Models. How this feature is used by QoS Models is out of scope of the NSLP specification.

The NSIS signaling protocol defines how the QNEs (more specifically state information stored at this QNEs) in the network can be influenced. Different NSLP signaling messages can be sent to create, modify, refresh or tear down the reservation state stored at the different routers. Note that aside from the reservation state store at the QNE, the *QoS NSLP operational state* can also be maintained. The QoS NSLP operation state is used by the QoS NSLP processing component to handle messaging aspects. Whereas the *QoS reservation state* is used by RMF to describe reserved resources for a session.

The format and sequence of the signaling messages is defined in the IETF draft "*NSLP for Quality-of-Service Signaling* [18]. In this section we will discuss the different message formats defined in this protocol and using and message sequence diagrams discuss how QoS signaling can be achieved.

Message Formats

The NSIS signaling protocol has defined four different types of messages:

RESERVE: The RESERVE message is the only message that manipulates QoS reservation state. It is used to create, refresh, modify and remove such state for a particular session.

QUERY: A QUERY message is used to request information about the data path without making a reservation. Note that a QUERY does not change an existing reservation state.

RESPONSE: The RESPONSE message is used to provide information about the result of a previous QoS NSLP message. This includes explicit confirmation of the state manipulation signaled in the RESERVE message, the response to a QUERY message or an error code if the QNE or QNR is unable to provide the requested information or if the response is negative. The RESPONSE message does not cause any reservation state to be installed or modified.

NOTIFY: NOTIFY messages are used to convey information to a QNE. They differ from RESPONSE messages in that they are sent asynchronously and need not refer to any particular state or a previously received message. The information conveyed by a NOTIFY message is typically related to error conditions. An example would be the notification to an upstream peer about state being torn down or to indicate when a reservation has been preempted.

The message type, along with several flag bits, is specified in a signaling message's common header. Every signaling message consists of a common header, followed by a body consisting of a number of variable length QoS *objects*. The common header is defined as follows:

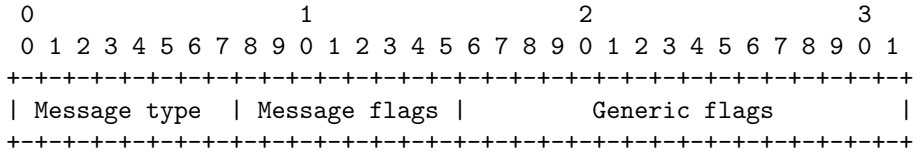


Figure 3.6: The common header bit format.

The first 8 bits of the common header specify the message type which have been discussed above:

1 = **RESERVE**

2 = **QUERY**

3 = **RESPONSE**

4 = **NOTIFY**

The next 8 bits are used for the *message specific flags*. The *message specific flags* are defined as part of the specification of individual messages and are different with each message type. Currently the RESERVE and the QUERY messages are the only message types to have message specific flags. The RESERVE message has the following message specific flags:

TEAR (T) - this flag indicates whether the reservation state and QoS NSLP operation state should be torn down. Depending on the QoS model, the tear message may include a QSpec to further specify state removal.

REPLACE (R) - this flag has two uses. First it indicates that a RESERVE (with a different MRI but same SID) replaces an existing one, so the old one MAY be torn down immediately. Second, this flag is used to indicate whether the reserved resources on the old branch should be torn down or not when a data path change happens. In this case, the MRI remains the same and only the route path changes.

The QUERY message has the following message specific flag:

RESERVE-INIT (R) - this flag indicates whether the QUERY message is a trigger for a receiver initiated reservation. If this flag is set then the QNR is triggered to set up the reservation by sending back a RESERVE message.

The final 16 bits are reserved for the *generic flags*. The *generic flags* are the same for all message types and specify how QNE should process the NSLP messages with respect to routing. The following flags have been defined:

SCOPING (S) - when set, it indicates that the message is scoped and should not travel down the entire path but only as far as the next QNE (scope="next hop"). By default, this flag is not set (default scope="whole path").

PROXY (P) - when set, it indicates that the message is scoped, and should not travel down the entire path but only as far as a pre-defined proxy-QNE. Note that in this thesis proxy mode signaling is not discussed. For more information about proxy mode signaling the reader is referred to [18] (section 4.8). By default, this flag is not set.

ACK-REQ (A) - when set, it indicates that the message should be acknowledged by the receiving peer. The flag is only used between stateful peers, and only used with RESERVE and QUERY messages. Currently, the flag is only used with refresh messages.

This flag is useful when a QNE wants to make sure that the messages received by the downstream QNE have truly been processed by the QoS NSLP and have not just delivered by GIST. This makes faster dead peer diagnostics on the NSLP layer possible.

The A-flag must not be set for RESERVE messages that already include an RII object, since a confirmation has already been requested from the QNR. Also, it should be noted that this flag can provide a more reliable transport of NSLP messages however, the message transmission reliability between two QoS NSLP peer should be handled by GIST, not by the NSLP itself.

BREAK (B) - when set, indicates that there are routers along the path where QoS cannot be provided.

In the message's body several QoS *objects* are specified. A QoS object can be one of the following types:

1. *Control Information*: Control information objects carry general information for the QoS NSLP processing, such as sequence numbers or whether a response is required.
2. *QoS specifications (QSpecs)*: QSpec objects describe the actual resources that are required and depend on the QoS model being used. Besides any resource description they may also contain other control information used by the RMF's processing.
3. *Policy objects*: Policy objects contain data used to authorize the reservation of resources.

A QoS object is specified in a Type-Length-Value (TLV) format. Every object in this format has the following 32-bit object header:

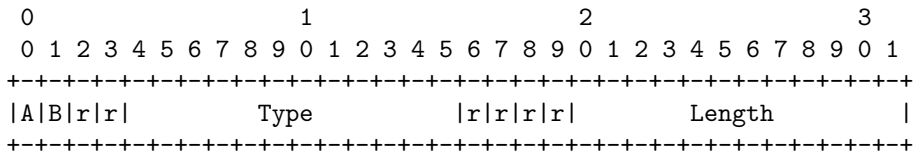


Figure 3.7: The QoS object header bit format.

The 'A' en 'B' flags are used to signal the desired treatment for the object. These flags can have the following values:

AB = 00: indicates that the object is *mandatory*. If this object cannot be processed by a QNE, the entire message containing the object should be rejected and an error message should be sent back to the sender.

AB = 01: indicates that the object should be *ignored*. If this object cannot be processed by a QNE, the object should be ignored and the rest of the message should be processed.

AB = 10: indicates that the object should not be changed if the message is *forwarded*.

AB = 11: indicates that the object should be *refreshed*.

The *type* bits are used to indicate the type of the object. In the NSIS protocol specification the following object types have been defined[2]:

- *Request Identification Information (RII)*. Type: **0x01**.
  The RII object is a 32-bit identifier which must be (probabilistically) unique within the context of a session. For every message that requires a RESPONSE message, a different identifier should be generated which can be used to match the RESPONSE with a previously sent RESERVE or QUERY message.
- *Reservation Sequence Number (RSN)*. Type: **0x02**.
  The RSN is an incrementing sequence which indicates in which order the state modification should be performed. The object consists of two 32-bit words: the RSN and an Epoch identifier, which is used to identify peer restarts.
- *Refresh Period (REFRESH_PERIOD)*. Type: **0x03**.
  This object specifies the refresh timeout period (in milliseconds) to be used for this message.

[2] All the objects defined in the NSIS protocol specification are mandatory objects.

- *Bound Session ID (BOUND_SESSION_ID).* Type: **0x04**.
  The BOUND_SESSION_ID object is used to indicate whether this session is bound to another QoS session. The object is subdivided in a BINDING_CODE and the 128-bit identifier of the associated session. The BINDING_CODE parameter is used to indicate the nature of the binding. The concept of session binding can be used to indicate a dependency relation between the end-to-end session and the aggregate session or in the case of bidirectional reservations, used to express the dependency between the sessions used for forward and reverse reservation.

- *Packet Classifier (PACKET_CLASSIFIER).* Type: **0x05**.
  The PACKET_CLASSIFIER object contains the Message Routing Method (MRM) specific information. The MRM is the algorithm that should be used for the routing of the signaling messages. Currently two MRM have been specified, the patch-coupled and path-decoupled MRM, of which only the former is used. A complete list of all possible error codes is given in [18].

- *Information object (INFO_SPEC).* Type: **0x06**.
  The INFO_SPEC object contains information regarding possible errors that have occurred during a signaling interaction. Note that the errors can also be used for informational purposes or to indicate the successful processing of a signaling message.

- *Session ID List (SESSION_ID_LIST).* Type: **0x07**.
  This object contains a list of 128-bit session identifiers which can be used in summary refresh or summary tear messages.

- *Reservation Sequence Number (RSN) List (RSN_LIST).* Type: **0x08**.
  This object contains a list of RSN values.

- *Message ID (MSG_ID).* Type: **0x09**.
  This object contains an identifier for the signaling message containing the object.

- *Bound Message ID (BOUND_MSG_ID).* Type **0x0A**.
  The BOUND_MSG_ID object can be used to indicate the dependency between two different messages, much like the session binding. This object contains a 1-bit MESSAGE_BINDING_TYPE flag, indicating the nature of the binding, and a 128-bit identifier of the associated message. The two supported binding types are:

  1. a unidirectional binding dependency, and
  2. a bi-directional binding dependency.

- *QoS Specification (QSpec).* Type **0x0B**.
  The QSpec object contains the QoS information needed for the modification of the reservation state. The format of this object is dictated by the QoS model used. The QSpec object and QoS models are discussed in section 3.4.1 and 3.4 respectively.

All the objects listed above, except for the *QSpec* object, are considered to be *Control information* objects. The structure of a sample RESERVE message is given in Appendix C. This sample output was generated by our prototype implementation. In the next section we will discuss how these messages and parameters can be used to setup a reservation for a particular flow.

Message Sequences

The NSIS signaling protocol supports the following two scenarios for the set up of a reservation: the sender-initiated reservation and the receiver-based reservation. Note that we will not discuss the usage of the QUERY and NOTIFY signaling messages described above. This is because the message sequence diagram for the querying of resources or notification of errors or state changes is similar to the message sequence diagram discussed here (also see Figure 3.8). For more information regarding these message sequences the reader is referred to [16].

The sender-initiated reservation The reservation of resources can be explained using the message sequence diagram shown in Figure 3.8. To make a new reservation, the QNI constructs a RESERVE message. This message contains a QSPEC object describing the required QoS parameters.

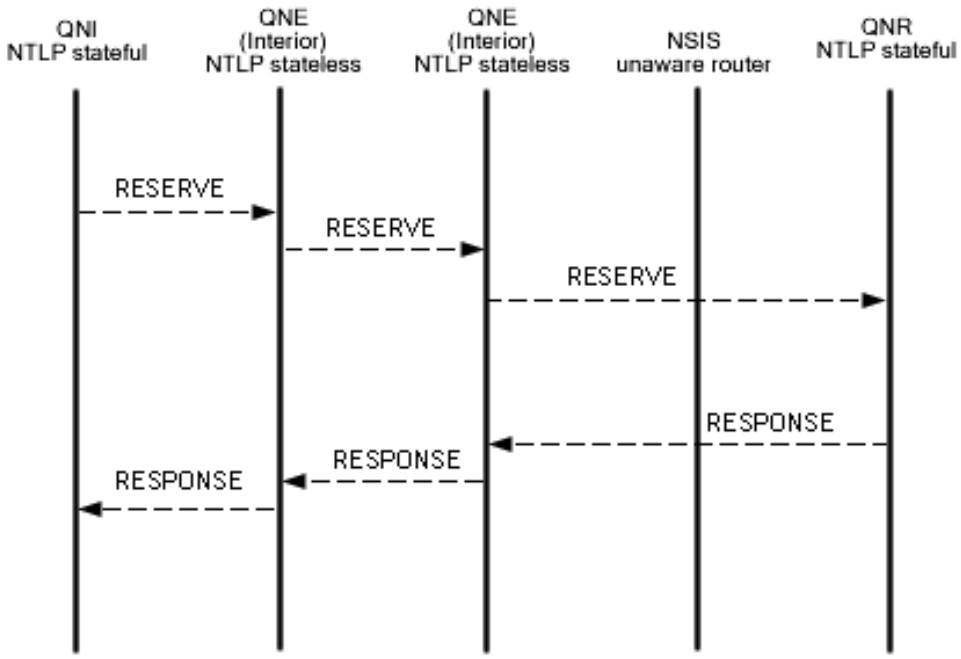


Figure 3.8: A message sequence diagram of a basic reservation.

> *The QSpec object contains all information regarding the resources which need to be reserved. It encapsulates the relevant QoS parameters that are used for the creation of the reservation state. The format of the QSpec object is defined by the QoS model in which these parameters are defined and how they should be interpreted. A more in depth discussion of the QSpec object is given in section 3.4.1.*

The reservation of resources along a particular path is accomplished by establishing peering relationships between neighboring QNE's. Signaling messages are sent peer-to-peer to either downstream or upstream neighboring routers. These routers process the signaling message and based on the type of message and the role of the QNE, the message is forwarded to the next neighboring router or a RESPONSE is sent back to the sender.

In case of a reservation request the QNI passes the newly created RESERVE message to the NTLP layer which will transport it to the next QNE. The GIST located in the NTLP layer is responsible for the transport of signaling messages to neighboring QNE's. GIST is also responsible for the routing of signaling messages and the setup up of peering relationships between routers (QNE's). In section 3.3.1 GIST is discussed in more detail.

Once the QNI's neighboring QNE has received the RESERVE message several error flags are checked. If no error flags have been set, the QSpec object is extracted and processed based on the specified QoS model (QoSM). The node then performs the appropriate actions (e.g., installing reservation) specified by the QSpec object. The QoSM defines how the QSpec should be interpreted and which actions should be executed. QoS models are used to support different QoS architecture[3]. For example a QNE can implement the IntServ architecture as a QoSM while other QNE's use the DiffServ QoSM. Note that

[3] The QSpec object and the QoSM are dependent of each other. The QoSM model defines and implements the QoS parameters used for a particular architecture. The QSPEC object is used to signal the required resource specified in the QoSM. A more detailed description of the QoSMs is given in section 3.4.

although different QoSMs are used the signaling protocol remains the same. The signaling protocol does not depend on the parameters specified in the QSpec object and thus the transport of the QSpec object remains the same for all QoSMs.

If the message is processed correctly the QNE will generate a new RESERVE message (usually based on the one received by the QNE). This message is passed to GIST, which forwards it to the next QNE. The same processing is performed at the other QNEs along the path, up to the QNR. At the QNR the RESERVE message is processed and if no error flags have been set by the intermediate routers, the reservation is considered to be successful. In this case a RESPONSE message confirming the reservation is send back to the QNI. If the reservation was not successful then a RESPONSE message containing an error message is send back to the QNI.

Between the two end hosts a signaling session has been setup. For this session the reserved resources are maintained at the routers along the path of this flow. Formally a session is defined as '*an association between a QNI and QNR related to a data flow*' [16]. This session can identified using the Session Identification (SID) and all QNEs on the path, including the QNI and QNR, use the same identifier to refer to the network control state stored locally for the association. This SID can thus be used by these routers to refresh, update or tear down the reservation. Note that a particular QNI and QNR pair may have more than one session active at a time. In this case every session will have its own unique SID.

The receiver-initiated reservation The receiver-initiated reservation of resources is explained using the message sequence diagram shown in Figure 3.9. Note that in this figure the QNI is located on the right side whereas the QNR is located on the left side. The QNR, which in this case is the sender of the data flow, constructs a QUERY message. This message contains a QSPEC object describing the required QoS parameters. In this Query message the RESERVE-INIT (R) flag is set to indicate the receiver-initiated request as oppose to a QUERY for available resources.

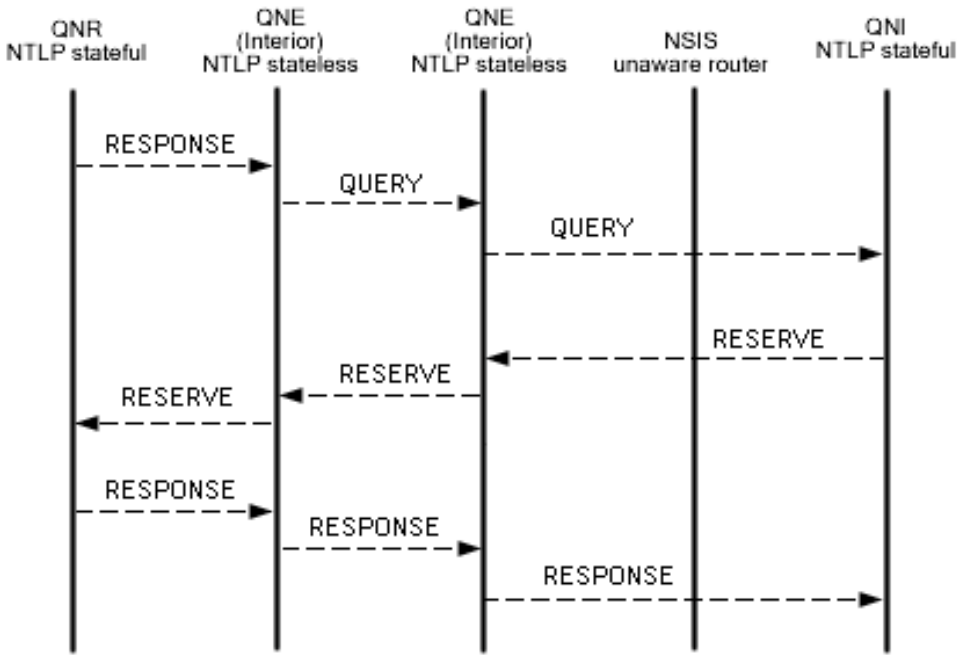


Figure 3.9: A message sequence diagram of a basic reservation.

Once the QNI's neighboring QNE has received the QUERY message it is checked for errors. If no error flags have been set, the QSpec object is extracted and processed based on the specified QoS model (QoSM). The QoSM defines how the QSpec should be interpreted and may choose to inform other

nodes of its available resources. This is done by creating a new QUERY message with an QSpec object containing the information regarding its available resources. Note that the QoSM can specify that an error message is generated if the amount of available resources is less than the amount of the requested resources. These are possible scenarios which have to be defined by the QoSM using this reservation method. This QUERY message is then sent to its downstream peer.

Once the QUERY message is processed accordingly and the QNE performs a check for its available resources. In a similar manner as described above a new QUERY message is constructed, based on the received QUERY message, which is sent downstream. Again, messages received by NSIS unaware routers are not processed by these routers but are simply forwarded.

Eventually the QUERY message is received by the QNI which will check whether the request can be granted, based on the available resources and the requested resources. If the request can be granted the QNI can set the reservation process in motion, which is similar to the sender-initiated reservation process discussed above.

3.4 Quality of Service Models

A QoS model (QoSM) incorporates QoS provisioning methods and an architecture to achieve QoS for a flow. Every QoSM has its own properties which are defined in a specification document. This specification states which NSLP features are to be used, how the QSPEC should be formated and interpreted, and how QoS parameters should be mapped onto the specific properties of the QoSM. By separating this functionality from the signaling protocol, it is now possible for a QNE to support multiple QoS models along a single data path. This is particularly useful for transit domains that may use a different type of signaling than the end node that initiated the signaling session. By using multiple QoSMs, border routers of the transit domain can now process the end-to-end signaling messages according to the associated end-to-end QoSM and translate the parameters to the QoSM used in the transit domain.

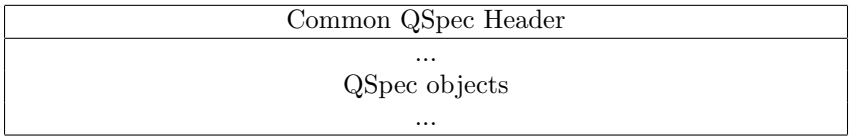
Some examples of QoSMs are, the end-to-end IntServ Controlled Load QoSM and the intra-domain Resource Management for DiffServ (RMD) QoSM [1]. In this section we will discuss the details of the QSpec which is the key component of a QoSM.

3.4.1 QoS Specification (QSpec)

The QSpec, defined in [13], is a template which can be used to hold different types of QoS object and QoS parameters. The QSpec objects are the input or outputs for the RMF and form the main building blocks of the QSpec. The QSpec parameters are part of a QSpec object and can specify one of the following:

- the *traffic* (TMOD) parameter, which must be included in the QSpec,
- a *constraint* (e.g., path latency or path jitter),
- a *traffic handling directive* (e.g., excess treatment),
- a *traffic classifier* (e.g., Per-Hop-Behaviour class).

If we look at the internal structure of a QSpec, shown in Figure 3.4.1, we see that it consists of a *Common QSpec Header* and a set of *QSpec objects*.



The Common QSpec Header consists of a *QSpec version* number (4 bits), which is assigned by Internet Assigned Numbers Authority (IANA), a *QSpec type* (4 bits), which corresponds to a particular QoSM, a *QSpec procedure* (8 bits), an *I-flag* which identifies whether the QSpec is an initiator QSpec (I = 0) or a local Qspec (I = 1), and a *Length* field (12 bits) which specifies the total length of the QSpec excluding the common header. The QSpec procedure field is subdivided into a 4-bit Message Sequence and another 4-bit Object Combination field. The *Message Sequence* field can have the following values:

0: *Sender-Initiated Reservations*

1: *Receiver-Initiated Reservations*

2: *Resource Queries*

The *Object Combination* field, which is dependent of the value of the Message Sequence field, can take the values between 1 and 3:

- Message Sequence: 0
  Object Combination: 1, 2, 3
- Message Sequence: 1
  Object Combination: 1, 2, 3
- Message Sequence: 2
  Object Combination: 1

The Object Combination field defines which QSpec object are to be used for a particular message sequence. For example, if we want to create a QSpec for a RESERVE message which is used in a sender -initiated reservation scenario (*Message Sequence = 0*), we can choose one of the following IDs for our object combination field :

ID	RESERVE	RESPONSE
1	QoS Desired	QoS Reserved
2	QoS Desired, QoS Available	QoS Reserved, QoS Available
3	QoS Desired, QoS Available, Minimum QoS	QoS Reserved, QoS Available

If we set our Object Combination field to '2' for example, we would have to include a QoS Desired and a QoS Available object in our QSpec. Every object combination has its own motivation why the specified QSpec objects should be used. We will not discuss all the possible values for the Object Combination field, or the motivation for their usage. For an exhaustive list the reader is referred to [13, *section 4.3*: **QSPEC Procedures**].

The QSpec also contains a set of QSpec objects which can be of one of the following types:

- **QoS Desired object**: The QoS Desired Object describes the resources the QNI desires to reserve. It is a read-only QSpec object and thus the QSpec parameters carried in the object may not be overwritten. QoS Desired is always included in a RESERVE message.
- **QoS Available object**: The QoS Available Object is included in a RESERVE or QUERY message. Its purpose is to collect information on the resources currently available on the path. This is a read-write object, which implies that the QSPEC parameters contained in this object may be updated. but they cannot be deleted. Every QNE that processes this object must inspect all the parameters carried in this object. If the QNE has less resources available than stated by the parameter, it must update this parameter accordingly. In other words, the QoS Available Object reflects the bottleneck of the resources currently available on a path.

  QoS Available can also be included in a RESPONSE message. In this case the QoS Available Object contains the information collected by a previously sent RESERVE or QUERY message and thus may not be updated by intermediate QNEs.

- **QoS Reserved object**: The QoS Reserved object reflects the resources that were reserved. It is a read-only object.
- **Minimum QoS object**: The Minimum QoS object allows the QNI to define a range of acceptable QoS levels by including both the desired QoS value and the minimum acceptable QoS in the same message. Parameters cannot be overwritten in this QSPEC object.

  Note that the QSpec specification states that the objects *QoS Desired, QoS Available and QoS Reserved* must be supported, whereas the support for the *Minimum QoS* object is optional.

The *QSpec parameters* are encapsulated in QSpec objects which are either desired, available, reserved or minimum. In the "*QoS NSLP QSPEC Template*" [13] several parameters have been defined which can be used freely in any QoSM specification. This is done to achieve interoperability between different QoSMs. The following parameters have been defined in [13]:

- <*TMOD-1*> Parameter:
  This parameter specifies the traffic using a token bucket. The parameter consists of four numbers:

  1. r: which represents the rate of the data flow in bytes/second.
  2. b: which represents the bucket size of the token bucket in bytes.
  3. p: which specifies the peak rate of a flow in bytes/second.
  4. m: which specifies the minimum policed units in bytes.

  This parameter is special because it must be included by the QNI and it must be interpreted by all other QNEs. All other QSpec parameters are populated by a QNI only if they are applicable to the QoSM used.
- <*TMOD-2*> Parameter
  This parameter is identical to the <TMOD-1> parameter. The reason why there is a second traffic parameter to support DiffServ applications. If an application wants to use the full set of Assured Forwarding (AF) [17] then two token buckets are needed.
- <*Path Latency*> Parameter
  This parameter specifies the maximum latency (in milliseconds) a flow wishes to experience.
- <*Path Jitter*> Parameter
  This parameter specifies the maximum jitter (in milliseconds) a flow wishes to experience.
- <*Path PLR*> Parameter
  This parameter specifies the minimum packet loss ratio (PLR) a flow wishes to experience.
- <*Path PER*> Parameter
  This parameter specifies the packet error ratio (PER) a flow wishes to experience.
- <*Slack Term*> Parameter
  The slack term parameter is the difference between desired delay and delay obtained by using bandwidth reservation.
- <*Preemption Priority*> and <*Defending Priority*> Parameters
  The Preemption Priority parameter is the priority of the new flow compared with the Defending Priority of previously admitted flows. Once a flow is admitted, the preemption priority becomes irrelevant.

  The Defending Priority parameter is used to compare the current flow with the preemption priority of new flows. The preemption priority of any flow must always be less than or equal to its defending priority.

- *<Admission Priority> and <RPH Priority>* Parameters
  These parameters provide an essential way to differentiate flows for emergency services like e.g., Emergency Telecommunications Service (ETS) (see [7]) or E911, and assign them a higher admission priority than normal priority flows and best-effort priority flows.

- *<Excess Treatment>* Parameter
  This parameter describes how the QNE will process out-of-profile traffic. Excess traffic can be dropped, shaped and/or remarked

- *<PHB Class>* Parameter
  This parameter specifies the Per-Hop-Behaviour (PHB) class to which this flow belongs. PHBs are used by the DiffServ architecture to differentiate traffic distinct classes. This parameter is used by the RMD-QoSM.

- *<DSTE Class Type>* Parameter
  This parameter specifies a QoS as a DiffServ-aware MPLS traffic engineering (DSTE) class type (see [12, 11]).

- *<Y.1541 QoS Class>* Parameter
  This parameter specifies a QoS as a Y.1541 class type (see [14]).

A QoSM can include any parameter defined above in its specification and define its own processing rules for it. In the next chapter we will discuss the parameters used by the RMD-QoSM, including their processing rules.

Chapter 4

Resource Management in DiffServ

In this chapter we will discuss the Resource Management for DiffServ QoS Model (RMD-QoSM) [1]. For clarification reasons some parts of the text in this Chapter have been copied from [1]. Here we will show how the RMD-QoSM can be used to support the aggregation of flows. In addition, we will show how our Aggregate Update Algorithm can be incorporated in the RMD-QoSM.

4.1 RMD Features Overview

Resource Management in DiffServ (RMD) [1] is an attempt to apply the DiffServ principles to the NSIS signaling framework. It introduces dynamic reservation and admission control into the DiffServ architecture. In RMD per flow classification, conditioning and admission control functions are moved to the edges of a RMD domain. Within the RMD domain per traffic class admission control is done. This way the reservation mechanism for the nodes within the domain is much simpler, which results in a more scalable solution for providing QoS guarantees in large scale networks.

For the dynamic reservation of resources within an RMD domain, RMD describes a method that is able to provide admission control for flows entering the domain. Two admission control modes are supported:

- **Measurement based**: This admission control mechanism uses measured traffic levels to make admission control decisions. Here traffic levels are measured continuously and based on the available amount of resources flows are admitted or not.
- **Reservation based**: Nodes that use the reservation based admission control scheme, base their admission control decisions on a flows traffic descriptor and available resources. This implies that, in the contrast to the measurement based admission control scheme, the reservation of resources needs to be stored.

For the purpose of this assignment we will assume the use of the reservation based admission control scheme. The reason for this is because this is the only admission control scheme currently supported by our prototype. Also in our problem statement we have assumed that only Constant Bit Rate (CBR) flow traverse the RMD domain. Such flow transfer their data at a fixed rate and thus can be described using one parameter, the data rate. Because there are no fluctuation within the flow's data rate the reserved resources are assumed to be fully utilized at all times. For such flows the performance gain when using is measurement base admission control is significantly lower. Last but not least, of the two admission control schemes, the reservation based scheme is easiest to implement.

In Figure 4.1 a simple RMD signaling scenario is shown. Here we see the five routers of which three are located in the RMD domain. The QNI and QNR are not part of the RMD domain but have a flow that traverses the RMD domain, depicted by the red arrow. The other three QNE nodes located in the RMD have one of the following roles:

- **QNE Ingress**: which is responsible for handling the incoming end-to-end signaling messages. At this node per-flow state information is stored for the end-to-end flows traversing the node. At this

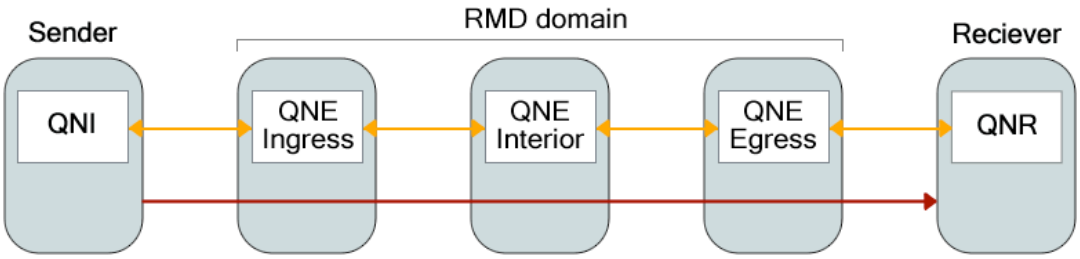


Figure 4.1: A RMD signaling scenario

node two QoSM are supported, the end-to-end QoSM and a local QoSM, the RMD-QoSM. Using the local QoSM intra-domain flows between the QNE Ingress and the QNE-Egress are setup and maintained. Using this local QoSM we will setup an aggregate flow between the QNE-Ingress and QNE-Egress.

- **QNE-Interior**: which is responsible for handling signaling messages according to the local QoSM used by the QNE Ingress (and QNE-Egress). QNE-Interior nodes are very lightweight because they only store reduced NSLP state per traffic class (for reservation-based RMD).
- **QNE-Egress**: which is responsible for handling incoming intra-domain as well as end-to-end signaling messages. At this node the received intra-domain signaling messages are processed according to the local QoSM whereas the end-to-end signaling messages have to be processed according to the end-to-end QoSM. Note that the end-to-end signaling messages are tunneled through the RMD domain and are not processed by the interior nodes.

In an RMD domain edge nodes support the fine-grained end-to-end reservation mechanisms whereas the interior nodes support a simpler (aggregate) reservation mechanism. When using reservation-based RMD per PHB state information is stored at all the nodes located in the communication path from the QNE Ingress node up to the QNE-Egress node. This state is identified by the PHB class value and it maintains the number of currently reserved resource units (or bandwidth). At the QNE-Ingress end-to-end reservations requests are mapped onto a PHB group and associated with the corresponding aggregate reservation. If the end-to-end flow can be admitted to the aggregate reservation, the end-to-end reservation request is forwarded towards the QNE-Egress using some tunneling mechanism in order to bypass the QNE-Interior nodes. At the QNE-Egress an admission control check for the end-to-end request is performed. If the request can be granted, some state information is created and stored at this node. The end-to-end request is then send further downstream towards the QNR.

In a nutshell the RMD specification describes the following procedures:

- RMD support the classification of individual resource reservations or resource query into Per-Hop-Behavior (PHB) groups. This is done at the QNE-Ingress node of the domain.
- A hop-by-hop admission control based on a PHB is used within the RMD domain. It uses either the reservation based admission control scheme or the measurement based admission control scheme.
- RMD supports the tunneling of the original reservation request across the domain. This way end-to-end signaling messages can pass through the RMD domain without being processed by the interior nodes.
- A congestion control algorithm is present in RMD to notify the QNE-Egress nodes about congestion. The algorithm will terminate the appropriate number of flows in case of congestion due to a sudden failure within the domain.

Limitations and Considerations

RMD has the following limitations and considerations:

- RMD can not support the full set of Assured Forwarding (AF) [17] per hop behavior traffic classes. Currently the only supported PHB is the Expedited Forwarding (EF) [8] PHB or one class of the AF PBH sets. The reason why RMD does not support the full set of AF PHBs is because this would require two token buckets. Note that the NSIS QSpec template [13] includes the possibility for specifying two traffic parameters (<TMOD-1> and <TMOD-2>), which can be used to represent two token buckets. Currently the RMD specification does not take advantage of this possibility.
- Only one RMD-QoSM can be used in one RMD domain because NSIS aware routers cannot process two different schemes at the same time. Thus within the RMD domain either the reservation-based or the measurement-based method can be used, not a mix of the two.

4.2 RMD QoS Model

The RMD-QoSM is specified in [1]. Here the signaling, the format of the QSpec and the processing rules are defined. First we will discuss how the signaling messages are to be transported, followed by the format of the QSpec, which we will refer to as the RMD-QSpec. The signaling and processing rules are discussed afterwards.

4.2.1 Transport of signaling messages

If we look at the protocol model of the same signaling scenario (shown in Figure 4.2), we can clearly see the different QoSM used in the signaling scenario described above. The QNI and QNR set up an reservation using a particular end-to-end (e2e) QoS model. The edge nodes of the RMD domain must also support this end-to-end QoSM because they must be able to interpret the end-to-end requests in order to make a proper translation to the local (RMD) QoSM used in the domain. In contrast to the edge nodes, the interior node only supports the local QoSM used in the domain.

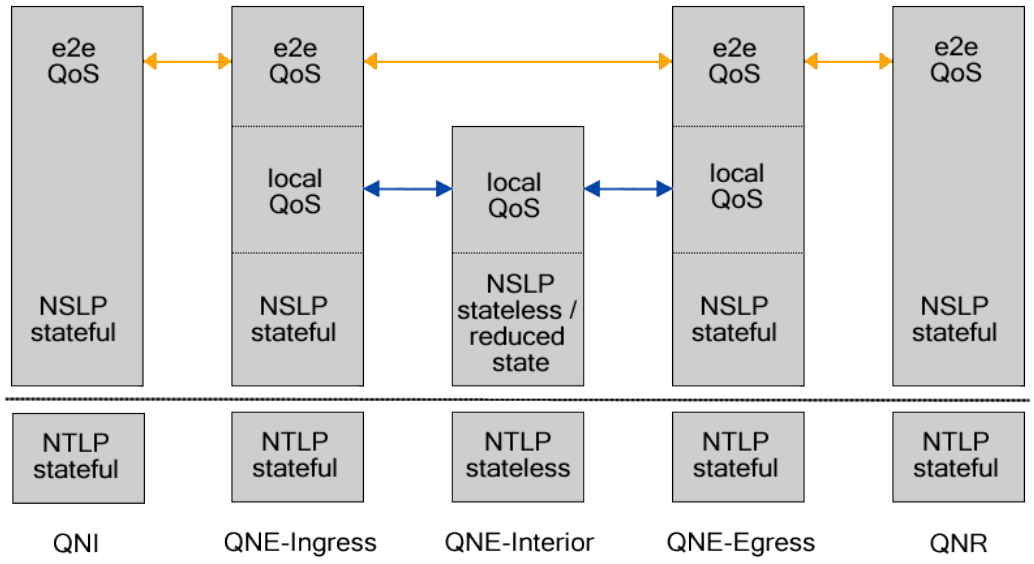


Figure 4.2: The RMD protocol model

In Figure 4.2 the state information stored at the different nodes is shown. The end-to-end QoSM requires the storage of per-flow state information. This translates to the nodes having to maintain a per-flow NSLP operational state. Because the QNI, QNE-Ingress, QNE-Egress and QNR nodes support the end-to-end QoSM, they have to store per-flow QoS NSLP operational state and are considered to be *NSLP stateful*. In contrast to the other nodes, the QNE-Interior node can either store 'reduced' NSLP state or be completely stateless. When using the measurement based admission control scheme, the QNE-Interior nodes are stateless. If the reservation based scheme is used then reduced NSLP state, storing per PHB aggregated QoS NSLP states, is maintained at the QNE-Interior node.

For the NTLP layer state information is stored at all nodes except the QNE-Interior nodes. This implies that no routing association is set up for nodes within the RMD domain. In order to achieve this (intra-domain) signaling messages have to be sent in GIST Datagram mode (D-mode). Intra-domain signaling is denoted in Figure 4.2 by the blue arrow. This is done as follows:

- When the QNE-Ingress is ready to send an (intra-domain) NSLP signaling message it must instruct GIST to send the message in unreliable mode with no security. This can be done by simply setting the transfer-attributes of the GIST API to operate in unreliable mode with no security. In doing so GIST will send the NSLP signaling message by piggybacking it on a GIST QUERY message.
- At the QNE-Interior node the GIST QUERY message is received and using the *RecvMessage* service primitive of the GIST API passed on to the NSLP layer. This service primitive has a parameter called 'Routing-State-Check' (also see appendix A.2). This boolean is used by GIST to check with the signaling application whether routing state should be created. At the QNE-Interior node this boolean should be set to false, in order to notify GIST that no routing state should be created. In this case the (modified) NSLP payload is forwarded downstream using a GIST QUERY message. This process is repeated for every QNE-Interior node until the NSLP signaling message is delivered to the QNE-Egress.
- At the QNE-Egress routing state should be created and thus when the NSLP message is received, using the *RecvMessage* service primitive, the Routing-State-Check boolean is set to true. At this point GIST creates the routing state between the QNE-Ingress and QNE-Egress and traffic can now be routed directly from one node to another. The routing state at the QNE-Egress can immediately be used to send NSLP signaling message back tot the QNE-Ingress.
- Note that in the procedure described above we have discussed how (intra-domain) signaling messages can be sent when no routing state has been set up. When an (intra-domain) message needs to be send after the routing state has been set up, the NSLP should notify GIST to send the message in Query mode (Q-mode). In this case the message should be send with the *SendMessage* service primitive with the transfer attributes set to *unreliable* and *no security*. In addition, the *local processing* parameter of the transfer attributes should be set such that GIST sends the (intra-domain) signaling message in a Q-mode even if there is a routing state at the QNE Ingress. The intra-domain signaling message is piggybacked on the GIST DATA message that is forwarded in Q-mode and processed by the QNE-Interior nodes up to the QNE-Egress.

The transport of the signaling message above applies only to the transport of intra-domain NSLP signaling messages. For the transport of end-to-end signaling messages (denoted in Figure 4.2 by the yellow arrow) the following procedure should be used:

- At the QNE-Ingress the end-to-end signaling message should be forwarded to the QNE-Egress and further but ignored by the QNE-Interior nodes. The QNE-Interior nodes are bypassed using multiple levels of the router alert option (ROA). In this case, interior routers are configured to handle only certain levels of (RAO) values. The ROA is set by GIST which derives its value from the supplied NSLPID by the signaling application. Using the *SendMessage* service primitive, the signaling can pass the NSLPID as parameter to GIST. At the QNE-Ingress the default NSLPID value of the end-to-end signaling message is replaced with another predefined NSLPID and send towards the QNE-Egress.

- At the QNE-Interior nodes GIST checks whether the RAO of the signaling message is supported by the node. Because these nodes are only configured to handle certain RAO values, and not the one included in the signaling message, the message is not processed and simply forwarded downstream.
- Once the signaling message reaches the QNE-Egress the marking performed by the QNE-Ingress is reversed. The value of the NSLPID of the signaling message is replaced with the default NSLPID and the message is forwarded towards the QNR.

4.2.2 RMD-QSpec

The format of the RMD-QSpec, defined in [1], is derived from the QSpec template defined in [13]. The **common QSpec header** fields in the QSpec object carried by the RESERVE message are set as follows:

- *the <QSpec Version> is set to the default version. Currently this is '0' in our prototype.*
- *the <QSpec Type> is the ID of RMD-QoSM to be assigned by IANA.*
- *the <I> flag is set to 'Local' (I =1).*
- *the <QSpec Procedure> is set as follows:*
  - *Message Sequence = 0: Sender initiated*
  - *Object combination = 1: <QoS Desired> for RESERVE and <QoS Reserved> for RESPONSE, see table below (taken from [13]):*

ID	*RESERVE*	*RESPONSE*
1	***QoS Desired***	***QoS Reserved***
2	*QoS Desired, QoS Avail.*	*QoS Reserved, QoS Avail.*
3	*QoS Desired, QoS Avail., Min. QoS*	*QoS Reserved, QoS Avail.*

The RESERVE and RESPONSE messages used in the RMD-QSpec carry a QoS Desired object and a QoS reserved object respectively. The QoS Desired object contains the following parameters:

> *<QoS Desired> = <Bandwidth> <PHB Class> <Admission Priority>*

The QoS Reserved object in the RESPONSE message has the following parameters:

> *<QoS Reserved> = <Bandwidth> <PHB Class> <Admission Priority>*

The *<Bandwidth>* parameter is used to provide information about the amount of bandwidth that needs to be reserved or released. The parameter consist of a *Bandwidth_IDID*, which is not yet assigned by IANA, and a *Peak Data Rate* field.

The *<PHB Class>* parameter specifies the Per-Hop-Behavior class for a particular flow. There are two ways to specify the PHB class. One is using the DSCP parameter while the other uses a PHB ID parameter.

The *<Admission Priority>* provides an essential way to differentiate flows for emergency services, priority flows and best-effort flows. High priority flows, normal priority flows, and best-effort priority flows can have access to resources depending on their admission priority value, described as follows:

- 0 - best-effort priority flow

- 1 - normal priority flow
- 2 - high priority flow

Note that in the RMD-QOSM a reservation established with an <*Admission Priority*> parameter with value 1, is equivalent to a reservation established without an <*Admission Priority*>.

In the RMD-QoSM specification [1] the <*Traffic Handling Directives*> is specified with the following fields:

> || <*Traffic Handling Directives*> = <*PHR container*> <*PDR container*>

It seems that the <*Traffic Handling Directives*> is modeled as a QoS object. According to the QSpec template [13] there is no such <*Traffic Handling Directives*> object or parameter. The QSpec template does define some *Traffic Handling Directives* parameters such as the <*Preemption Priority*>, <*Defending Priority*>, <*Admission Priority*>, <*RPH Priority*> and the <*Excess Treatment*> parameters. Note that a QoSM can define its own traffic handling directives parameter, like the <PHR container> and <PDR container> parameter. These parameters should be included in a QoS Desired object, according to the QSpec template:

> *'Generally, a traffic-handling-directives parameter is expected to be set by the QNI in <QoS Desired>, and to not be included in <QoS Available>. If such a parameter is included in <QoS Available>, QNEs may change their value.'* from [13].

In our prototype implementation we will include the <PHR container> and <PDR container> parameters in the <QoS Desired> object. The format of the <QoS Desired> is then specified as follows:

> || <*QoS Desired*> = <*PHR container*> <*PDR container*> <*Bandwidth*> <*PHB Class*>
> <*Admission Priority*>

In the RMD-QoSM specification [1] the <PHR container> and <PDR container> parameters are also used to notify the QNE-Ingress of the successful or unsuccessful setup of reservations. We need to modify our QoS Reserved object in the RESPONSE message as follows:

> || <*QoS Reserved*> = <*PHR container*> <*PDR container*> <*Bandwidth*> <*PHB Class*>
> <*Admission Priority*>

The <PHR container> parameter contains control information for intra-domain communication and reservation. The <PDR container> contains additional control information that is needed for edge-to-edge communication. The details of these parameters are discussed below.

PHR container

The PHR container consists of the following parameters:

> <*PHR container*> = <*S*>, <*M*>, <*Admitted Hops*>, <*B*>, <*Hop_U*>, <*Time Lag*>,
> <*Overload %*>, <*I*>

The bit format of the PHR container is shown in Figure 4.3. Note that the <Hop_U> parameter is given by <U>.

Parameter/Container ID:
8 bits. The container ID indicates the PHR type, which can be one of the following:

- **PHR_Resource_Request**: used to initiate or update the traffic class reservation state on all nodes located on the communication path between the QNE-Ingress and QNE-Egress nodes.
- **PHR_Refresh_Update**: used to refresh the traffic class reservation soft state on all nodes located on the communication path between the QNE-Ingress and QNE-Egress nodes according to a resource reservation request that was successfully processed during a previous refresh period.

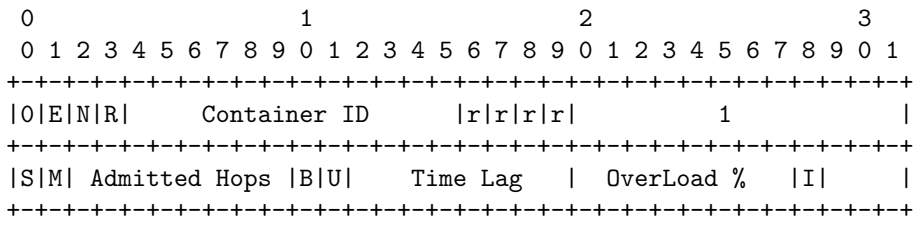


Figure 4.3: The PHR container bit format.

- **PHR_Release_Request**: used to explicitly release a certain amount of reserved resources, by subtraction, for a particular flow from a traffic class reservation state.

<S> (Severe Congestion):
1 bit. In case of a route change refreshing RESERVE messages follow the new data path, and hence resources are requested there. If the resources are not sufficient to accommodate the new traffic, severe congestion occurs. Severe congested Interior nodes should notify Edge QNEs about the congestion by setting the 'S' bit.

<M>:
1 bit. In case of unsuccessful resource reservation or resource query in an Interior QNE, this QNE sets the M bit in order to notify the Egress QNE.

<Admitted Hops>:
8 bits. The <Admitted Hops> parameter counts the number of hops in the RMD domain where the reservation was successful. It is set to '0' when a RESERVE message enters a domain and is incremented by each Interior QNE, provided that the 'Hop_U' bit is not set. However, when a QNE does not have sufficient resources to admit the reservation, the 'M' bit is set, and the <Admitted Hops> value is frozen, by setting the 'Hop_U' bit to '1'.

<B>:
1 bit. When set to '1' it indicates bi-directional reservation.

<Hop_U> (NSLP_Hops unset):
1-bit. The QNE-Ingress node sets the <Hop_U> parameter to '0'. This parameter should be set to '1' by a Interior node when it does not increase the <Admitted Hops> value. This is the case when a reservation request is not granted. When <Hop_U> is set '1' the <Admitted Hops> are not to be changed. Note that this flag in combination with the <Admitted Hops> flag are used to locate the last node that successfully processed a reservation request.

<Time Lag>:
8 bits. The time lag used in a sliding window over the refresh period.

<Overload %>:
8 bits. In case of severe congestion the level of overload is indicated by the <Overload %> parameter. This parameter should be higher than 0 if 'S' bit is set. If overload in a node is greater than the overload in a previous node then the <Overload %> parameter should be updated.

<I>:
1 bit. When set to '1' it indicates that the resources/bandwidth carried by a tearing RESERVE must not be released.

PDR container

The PDR container consists of the following parameters:

<PDR container> = *<S>*, *<M>*, *<Max Admitted Hops>*, *<B>*, *<Overload %>*,
[*<PDR Bandwidth >*]

The bit format of the PHR container is shown in Figure 4.4. Note that the <Max Admitted Hops> parameter is given by <Max Adm Hops>

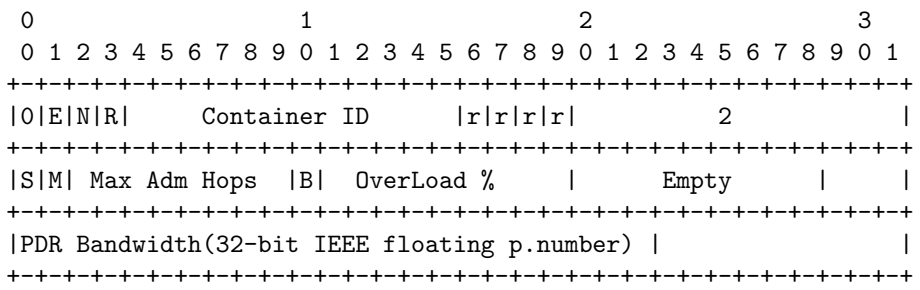


Figure 4.4: The PDR container bit format.

Parameter/Container ID:
8 bits. The container ID indicates the PDR type, which can be one of the following:

- **PDR_Reservation_Request**: generated by the QNE-Ingress node in order to initiate or update the QoS-NSLP per domain reservation state in the QNE-Egress node.
- **PDR_Refresh_Request**: generated by the QNE-Ingress node and sent to the QNE-Egress node to refresh, in case needed, the QoS-NSLP per domain reservation states located in the QNE-Egress node.
- **PDR_Release_Request**: generated and sent by the QNE-Ingress node to the QNE-Egress node to explicitly release the per domain reservation states.
- **PDR_Reservation_Report**: generated and sent by the QNE-Egress node to the QNE-Ingress node to report that a "PHR_Resource_Request" and a "PDR_Reservation_Request" traffic handling directive fields have been received and that the request has been admitted or rejected.
- **PDR_Refresh_Report**: generated and sent by the QNE-Egress node in case needed, to the QNE-Ingress node to report that a "PHR_Refresh_Update" traffic handling directive field has been received and has been processed.
- **PDR_Release_Report**: generated and sent by the QNE-Egress node in case needed, to the QNE-Ingress node to report that a "PHR_Release_Request" and a "PDR_Release_Request" traffic handling directive fields have been received and have been processed.
- **PDR_Congestion_Report**: generated and sent by the QNE-Egress node to the QNE-Ingress node and used for congestion notification.

<S> (Severe Congestion):
1 bit. Specifies if a severe congestion situation occurred. It can also carry the <S> parameter of the "PHR_Resource_Request" or "PHR_Refresh_Update" fields.

<M>:
1 bit. Carries the <M> value of the "PHR_Resource_Request" or "PHR_Refresh_Update" traffic handling directive fields.

<Max Admitted Hops>:
8-bit. The <Admitted Hops> value that has been carried by the PHR container field used to identify the RMD reservation based nodes that admitted or processed a "PHR_Resource_Request".

<B>:
1 bit. When set to '1' it indicates bi-directional reservation.

<Overload %>:
8 bits. This parameter includes the <Overload %> parameter specified by the "PHR_Resource_Request" or "PHR_Refresh_Update" control information fields, indicating the level of overload to the QNE-Ingress node.

<PDR Bandwidth>:
32 bits. This field specifies the bandwidth that either applies when the 'B' flag is set to '1' and when this parameter is carried by a RESPONSE message, or when a severe congestion occurs and the QNE edges maintain an aggregated intra-domain QoS-NSLP operational state and it is carried by a NOTIFY message. In the situation that the 'B' flag is set to '1' this parameter specifies the requested bandwidth that has to be reserved by a node in the reverse direction and when the intra-domain signaling procedures require a bi-directional reservation procedure. In the severe congestion situation this parameter specifies the bandwidth that has to be released.

In the next section we will discuss the processing rules for the parameters discussed above. The processing rules are explained using relevant signaling scenarios. The scenarios discussed in the next section are:

- The successful setup of an aggregate reservation.
- The unsuccessful setup of an aggregate reservation.
- The successful increase of an aggregate reservation.
- The unsuccessful increase of an aggregate reservation.
- The successful decrease of an aggregate reservation.
- The successful refresh of an aggregate reservation.

4.3 Flow aggregation in RMD-QoSM

In this section we will discuss how the RMD-QoSM can be used for the aggregation of flows. In order to add support for flow aggregation we need to focus on the following areas:

- **Setting up the aggregate reservation**. Before flows can be aggregated there needs to be an aggregate flow onto which the end-to-end flows can be mapped. So the first thing to do would be to set up an aggregate reservation.
- **Admission control**. In a per flow reservation scheme the resource management function only checks whether there are enough resources available to grant the end-to-end reservation request. With the aggregation scheme this is a different story. Here the RMF should query the aggregate reservation state to check whether it has enough spare bandwidth to grant the end-to-end reservation request.
- **The increasing and decreasing of the aggregate reservation**. Increasing the aggregate reservation means reserving extra bandwidth for the aggregate flow. In this case the RMF should check locally whether there is enough bandwidth to grant this request. Only then is the aggregate reservation increased. Decreasing the aggregate reservation is done in a similar manner as the release of an end-to-end reservation. When an aggregate reservation wishes to decrease its reservation size, it should notify the RMF to release the requested amount of bandwidth. The RMF on its turn then does the actual release of the bandwidth.

- **The refreshing of the aggregate reservation**. Just as end-to-end reservations, the aggregate reservation should also be refreshed from time to time. The main advantage here is that instead of having to refresh all the end-to-end reservation mapped onto the aggregate reservation, only the aggregate reservation needs to be refreshed. The refreshing of the aggregate reservation is done in a similar manner as the refreshing of the end-to-end reservation.

The modifications to the protocol are explained using message sequence diagrams. For these message sequence diagrams we will assume the following network topology.

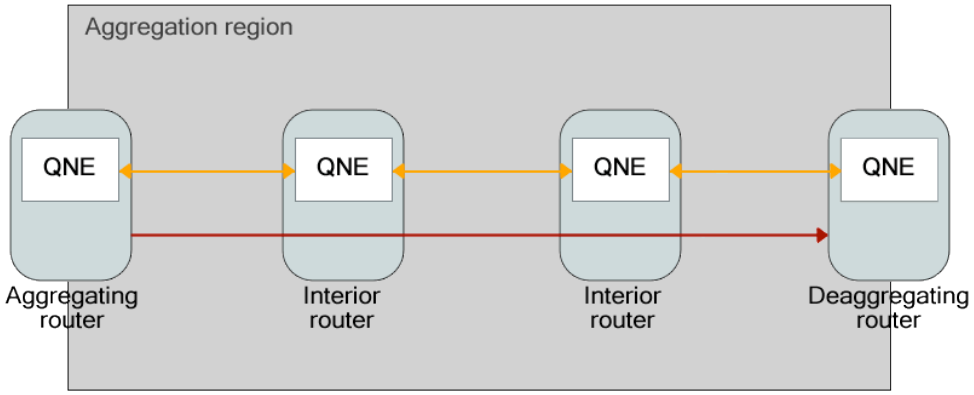


Figure 4.5: The scenario for the following reservation actions.

In Figure 4.5 only the routers that are part of the RMD domain are shown. The routers that fall outside the RMD domain are not shown but their existence is assumed in the message sequence diagrams. For example, the diagrams assume that the QNE Ingress routers can receive reservation requests from QNE routers that want to set up end-to-end data flows that traverse the RMD domain.

The '*aggregation region*' shown in Figure 4.5 is defined in RFC 3175 [2], which discusses the problem of aggregating RSVP flows. In RFC 3175 an extension to the RSVP protocol is discussed which makes it possible to aggregate multiple end-to-end RSVP reservations. Here the 'aggregation region' is defined as follows:

> *"... a set of RSVP-capable routers for which end-to-end (E2E for short) RSVP messages arriving on an exterior interface of one router in the set would traverse one or more interior interfaces (of this and possibly of other routers in the set) before finally traversing an exterior interface"*, from RFC 3175 [2].

This definition simply states that (RSVP) signaling messages arriving at the aggregation region traverse one or more routers before finally exiting the aggregation region. The signaling messages belonging to an end-to-end (E2E) reservation however are not processed by the routers in the aggregation region. Thus aggregation implies the ability to hide E2E RSVP messages from RSVP-capable routers inside the aggregation region. This is useful because this way reservations within the aggregation region can be created, maintained and removed independent of the E2E reservations that cross the aggregation region.

The router at which the signaling message entering the aggregation region first arrive is defined as the 'aggregating' router or 'aggregator' while the router at which the signaling messages departing the aggregation region arrive is referred to as the 'de-aggregating' router or the 'de-aggregator'. In Figure 4.5 the aggregation region with an aggregator and a de-aggregator is shown. The aggregating router has an incoming interface that does not fall within the aggregation region, whereas the outgoing interface of this router does fall within the aggregation region.

In our signaling scenarios we will also use this concept of aggregation region. Here we define the RMD domain to be the aggregation region. The QNE-Ingress node of the RMD domain will be the aggregator and the QNE-Egress node will be the de-aggregator.

4.3.1 Aggregate reservation setup

Setting up the aggregate reservation can be done in several ways. An aggregate reservation can be setup either by the *sender* (QNE Ingress) or by the *receiver* (QNE-Egress). Also, initialization of the aggregate reservation can be done dynamically or manually.

The NSIS framework supports both sender initiated reservation as well as receiver initiated reservations. In a sender initiated reservation, the sender of the data flow is responsible for the maintenance of the signaling session. Whereas, in the receiver initiated case the receiver of the data flow requests needs to maintain the signaling session. Also see Section 3.3.2 in which the message sequence diagram of the basic sender initiated and basic receiver initiated signaling is discussed.

The sender and receiver initiated approaches have several differences, of which the main ones are:

- In a receiver-initiated approach, the signaling messages traveling from the receiver to the sender must be backward routed such that they follow exactly the same path as they came from. In a sender-initiated approach backward routing is not necessary, therefore the nodes on the path do not have to maintain backward routing state.
- Mobile nodes using the sender-initiated approach can initiate a reservation for its outgoing flows as soon as it has moved to another roaming subnetwork. In a receiver-initiated approach, the mobile node has to inform the receiver about its handover, thus allowing the receiver to initiate a reservation for these flows. For incoming flows, the reverse argument applies.
- In general, setup and modification will be fastest if the node responsible for authorizing these actions can initiate them directly within the NSLP. A mismatch between authorizing and initiating QNEs will cause additional message exchanges, either in the NSLP or in the protocol executed prior to NSIS invocation. Depending on how the authorization for a particular signaling application is done, this may favor either sender- or receiver-initiated signaling.

For the setup of our aggregate reservation we will implement the sender initiated approach. There are several advantages when using the sender initiated approach:

1. As stated above in the receiver-initiated signaling scenario, backwards routing state has to be maintained. This is not necessary when sender-initiated signaling is used.
2. When using the sender initiated approach the QNE Ingress is the one responsible for the maintenance of the aggregate reservation. Because the new end-to-end reservation requests first arrive at the QNE Ingress, this would be a logical place to control the aggregate reservation. If the receiver based approach is used then the QNE-Egress would have to send a QUERY message back to the QNE Ingress which is then triggered to set the actual reservation request in motion. Thus in the receiver based approach the setup of the aggregate reservation would take 1.5 * RTT[1] whereas the sender initiated approach would take 1 RTT.

The initialization of the aggregate reservation can be done in a dynamic or manual fashion. Possible options are:

- If the aggregate reservation is setup in a dynamic fashion, the component responsible for the aggregate reservation will determine when the aggregate is setup. For example a border router can be triggered to setup an aggregate reservation when receiving an end-to-end reservation. Upon receiving this reservation request, the border router can lookup and map the aggregate reservation for the end-to-end reservation. If no aggregate reservation exists, the node can choose to set one

[1] RTT stands for Round Trip Time. This is the time it takes for a message to be send from the sender to the receiver (or vice versa) and a response being received by the sender.

up. In this case a request is sent to the relevant Egress router. Once the aggregate reservation has been set up, the end-to-end request is mapped onto the aggregate reservation.

Note that the setup of the aggregate reservation does not necessary have to be triggered by the first received end-to-end reservation request. The threshold for the setup of the aggregate reservation can be set to a certain amount of bandwidth for example.

- The aggregate can also be setup at configuration time by the network administrator. In this case the network administrator chooses the Ingress and Egress routers for which an aggregate reservation should be set up. Once these reservations have been set they will exist until the reservation can not be refreshed anymore and is torn down. This is particularly the case during link or router failure. Note that in this case the aggregate reservation for that particular link or router have to be set up again.

Of the options described above the latter is chosen for our prototype implementation. The main reason for this is because it is the easiest to implement. On a side note, our selection for a particular setup mechanism has no major impact on the performance for our flow aggregation algorithm, because of the simple network topology. Our test environment consists of five nodes: a sender and a receiver router, two border routers and one interior router.

A successful reservation setup. In Figure 4.6 the message sequence diagram for the successful setup of the aggregate reservation is shown. The setup for the aggregate reservation is done as follows:

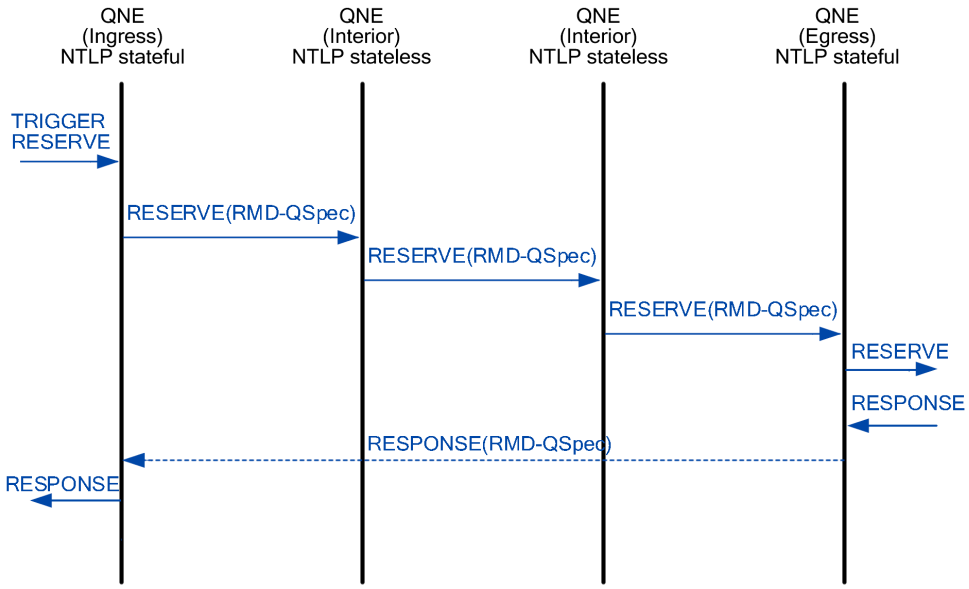


Figure 4.6: A successful setup of the aggregate reservation.

- The QNE Ingress node receives a trigger to set up the aggregate reservation. Upon receiving the trigger, the RMF checks whether the request can be granted locally and if so, create a (RMD) QSpec object with the relevant parameters. This QSpec is wrapped in a RESERVE message and sent downstream towards the QNE-Egress.

  The flags and objects of the RESERVE message must be set to the following values:

  The **generic flags** of the common header of the RESERVE message are set as follows:

- *The SCOPING (S) flag is not set.*
- *The PROXY (P) flag is not set.*
- *The ACK-REQ (A) flag is not set.*
- *The BREAK (B) flag is not set.*

The **message specific flags** of the common header of the RESERVE message are set as follows:

- *The TEAR (T) flag is not set.*
- *The REPLACE (R) flag is not set.*

In the RESERVE message the following **QoS objects** are set accordingly:

- *A new <RII> object is generated and included in the message.*
- *A newly generated <RSN> object is included in the message.*
- *The value of the <REFRESH_PERIOD> object is calculated and set by the QNE Ingress.*
- *The message includes a <PACKET_CLASSIFIER> object which is associated with the path-coupled MRM. Currently this is the only MRM supported by GIST and thus the <PACKET_CLASSIFIER> object does not have to be set to this MRM explicitly. Of all the flags of the <PACKET_CLASSIFIER> object only the T-flag needs to be set. This flag indicates that the DiffServ Code Point (DSCP) field included in the packet's IP header should be used for packet classification.*

  Note that the DSCP value in the MRI can be derived from the <PHB class> object which can be set by the QNE Ingress by passing this value to GIST, using the GIST API.

The **common QSpec header** fields in the QSpec object carried by the RESERVE message are set according to the RMD-QSpec definition (see section 4.2.2). The RMD-QSpec of the RESERVE message contains the following **QSpec object**:

- *a <QoS Desired> QSpec object containing the <Bandwidth> parameter, a <PHB Class> parameter and an <Admission Priority> parameter.*
  - *The value for the <Bandwidth> parameter is set to the initial size of the aggregate reservation. Only the peak rate field [p] of this parameter should be set.*
  - *The value for the <PHB Class> parameter is set to the EF traffic class. This is achieved by setting the DSCP field of this parameter to 0x2E.*
  - *The value for the <Admission Priority> parameter is set to 1 which is equivalent to a reservation without <Admission Priority> parameter.*
- *the <QoS Desired> QSpec object also carries the RMD traffic handling directives with a <PHR container> and a <PDR container> parameter.*
  - *The container id for the <PHR container> parameter is set to 'PHR_Resource_Request'. The value of the <Admitted Hops> field is set to 1. All other flags of the <PHR container> parameter are unset.*
  - *Note that the <PDR Container> parameter is not included in this message. In a single RMD domain the <PDR Container> parameter in the RESERVE message may be omitted.*

- Once the QNE Ingress has sent the RESERVE message downstream it is received by one of the QNE-Interior nodes. The QNE-Interior node processes the QSpec and checks whether the resources can be granted. This is done as follows[2]:

  - *Through the <PACKET_CLASSIFIER> object the QNE-Interior node is informed that packet classification should be done based on the DSCP value. The value of the DSCP can be obtained from the MRI which the NSLP receives from GIST. Once the value for the DSCP is obtained it must be associated with the value carried by the <PHB Class> carried in by the RMD-QSpec.*

    This is required, because there are situations that the <PHB class> parameter is not carrying a DSCP value, but a "PHB ID code".
  - *The QNE-Interior node decodes the PHR container parameter for processing. Here several flags are checked. First the container ID is checked to see which actions should be performed by the RMF. In this case it would be to reserve resources because the QNE-Ingress has set the container ID to 'PHR_Resource_Request'. Before actually performing this action the QNE-Interior node checks whether the M-flag of the PHR container is set. This flag indicates whether the resource reservation has failed at some upstream node. In this scenario the flag is not set and thus the RMF has to process the QSpec in order to perform the resource reservation.*
  - *The value of <Bandwidth> parameter of the RMD-QSpec object is obtained and used by the QNE-Interior node for admission control.*
  - *If the resource request is granted then they are added to the currently reserved resources (stored in the reservation state). Furthermore, the value of the <Admitted Hops> parameter in the PHR container has to be increased by one.*

  In this case the resources are reserved and a new RESERVE message, based on the received RESERVE message, is created. This RESERVE message is then sent further downstream. This RESERVE message is identical to the received RESERVE message with the only exception that the <Admitted Hops> parameter has been incremented.

[2]The steps defined here are executed by the RMF component. Thus this is after the RESERVE message has been processed by the QoS NSLP processing component according to the general processing rules.

- Eventually the RESERVE message is received by the QNE-Egress node. The QNE-Egress processes the message and checks whether the resources can be reserved locally. This is done in a similar fashion as QNE-Interior:

  - *The <PACKET_CLASSIFIER> and <PHB Class> are processed in the same way as done by the QNE-Interior nodes (described above).*
  - *The QNE-Egress processes the <PHR container> parameter in order to find the proper RMF action (resource reservation in this case). Here, the M-flag of this parameter is checked in order to verify whether the reservation was successful at all the QNE-Interior nodes in the path. If this is the case then the QNE-Egress has to install the QoS NSLP operational and QoS reservation state.*

  If this is the case then the QNE-Egress generates a RESPONSE message with a successful INFO_Spec object and sends it directly to the QNE Ingress. The INFO_Spec object is used to notify the QNE-Ingress that the reservation was successful. The values for the INFO_Spec object are set as follows:

  Error Severity Class: Success
  Error Code value: Reservation successful

  Below the complete details of the RESPONSE message are discussed:

  - *The <RII> object carried by the intra-domain RESERVE message is copied and added to the RESPONSE message.*
  - *Included in the RESPONSE message is an INFO_Spec object with the values set as described above.*
  - *In addition to the INFO_Spec a RMD-QSpec object is also added to the RESPONSE message. The RMD-QSpec contains a <QoS Reserved> object with the following values:*
    - *A <PDR container> parameter is included in the QSpec with the value of the container ID field set to 'PDR_Reservation_Report'. The value of the <M> field of the PDR container is set to be equal to the value of the <M> parameter of the PHR container that was carried by the RESERVE message.*
    - *The <PHR container>, <Bandwidth>, <PHB Class> and <Admission Priority> parameters included in the <QoS Reserved> object are omitted.*

- Upon receiving a RESPONSE message the QNE-Ingress uses the QoS NSLP functionality to match it to the RESERVE message sent earlier. After that, the RMD-QSpec carried by the RESPONSE message is identified and processed. The container ID of the <PDR container> parameter included in the RMD-QSpec is checked for the required action. With the container ID set to 'PDR_Reservation_Report' the QNE-Ingress needs to check whether the reservation was successful. This is achieved by checking the <M> flag of the <PDR container> parameter and the INFO_Spec carried by the RESPONSE. If the QNE Ingress has received a RESPONSE message with a successful INFO_Spec and a <PDR container> parameter with an <M> flag set to '0' then the reservation has been completed successfully.

An unsuccessful reservation setup It is also possible that the resources requested by the aggregate reservation could not be granted by a node in the path. In Figure 4.7 the message sequence diagram for the unsuccessful setup of the aggregate reservation is shown. The initiation of the reservation request is the same as described in the previous section.

- The QNE-Ingress creates a RESERVE message with a proper RMD-QSpec and sends this towards the QNE-Egress. The details of the RESERVE message are described in the previous section.

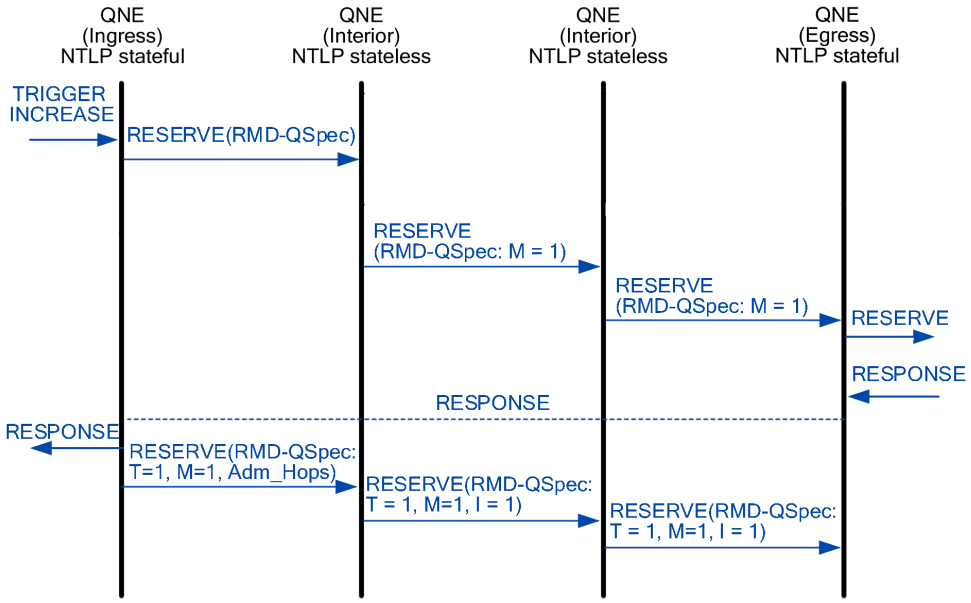


Figure 4.7: An unsuccessful setup of the aggregate reservation.

- At the QNE-Interior node the RESERVE message is processed and the relevant parameters decoded. In this case the first QNE-Interior node can not reserve the requested amount of bandwidth. At this point the QNE-Interior will update the RMD-QSpec as follows:

  - *The <M> flag of the <PHR container> parameter to '1', indicating that the reservation was unsuccessful.*
  - *In addition to the <M> flag, the <Hop_U> field of the <PHR container> parameter is set to '1', indicating that the node has not increased the <Admitted Hops> field.*

  The updated RMD-QSpec is then forwarded in a RESERVE message towards the QNE-Egress node.

- At the next QNE-Interior node the <PHR container> parameter is checked to see which RMF-related action should be executed and to check whether the reservation was successful at the upstream nodes. This QNE-Interior node will inspect the <M> flag and notice that the reservation was unsuccessful at some upstream node. At the same time the <Hop_U> field is inspected in order to check whether the <Admitted Hops> field should be updated. The value of the <Hop_U> field has been set to '1' and thus the <Admitted Hops> field is not to be updated. At this point no RMF-related action is performed and the message is forwarded towards the QNE-Egress.

- At the QNE-Egress router the RESERVE message is processed as described in the previous section. The QNE-Egress will notice that the <M> flag in the <PHR container> parameter is set and has to notify the QNE-Ingress of the reservation failure. This is done by creating a RESPONSE message with an INFO_Spec parameter set to the following error:

  Error Severity Class: Transient failure
  Error Code value: Reservation failure

  The RESPONSE message also includes the following objects:

- *The parameters of the RMD-QSpec for the RESPONSE message are derived from the RMS-QSpec that was carried by the RESERVE message. All the parameters for the <QoS Reserved> object are copied from the <QoS Desired> object.*
- *A <PDR container> parameter is also included in the RESPONSE message.*
  - *The container ID field is set to 'PDR_Reservation_Report'.*
  - *The value of the <Max Admitted Hops> parameter is derived from the <Admitted Hops> parameter of the PHR container.*
  - *The value of the (PDR) <M> flag is set to '1'.*

- At the QNE-Ingress the RESPONSE message is received and processed. The <PDR container> parameter of the RMD-QSpec is processed and the <M> flag inspected. This flag is set, indicating that the reservation has failed. At this point the QNE-Ingress needs to tear down the reservation that has been created at this node. In addition, a new tearing RESERVE message is constructed in order to release the bandwidth that has been reserved at some of the QNE-Interior nodes. This release procedure is called the '*RMD partial release procedure*'. In this RESERVE message none of the common header's **generic flags** are set. The **message specific flags** of the common header are set as follows:

  - *The TEAR (T) flag is set.*
  - *The REPLACE (R) flag is not set.*

  The RESERVE message has a RMD-QSpec with the following values:

  - *The RMD-QSpec for the RESERVE message should derive its parameters from the QSpec that was carried by the RESPONSE message. The <Bandwidth>, <PHB class> and <Admission Priority> parameters for the <QoS Desired> object are copied from the <QoS Reserved> object.*
  - *A <PHR container> parameter included in the RESERVE message are set as follows:*
    - *The container ID field is set to 'PHR_Release_Request'.*
    - *The value of the <Admitted Hops> parameter is set to '0'.*
    - *The value of the <M> parameter is set to '1'.*
  - *The <PDR container> parameter is copied from the RESPONSE message.*

- The RESERVE message is send downstream to the QNE-Interior nodes. The first QNE-Interior processes the message and passes the RMD-QSpec to the RMF. Here the <PHR container> is inspected for the proper RMF action. The container ID indicates that the RMF should release the previously installed reservation state. Before actually releasing these resources the <I> flag is checked. If the flag is set to '0' then the resources can be released. If the flag is set to '1', the resources are not released and the message is ignored. In this case the flag is set to '0' and the resources released. This is achieved by subtracting the value of the <Bandwidth> parameter from the reserved resources. After the resources have been removed, the <Admitted Hops> is incremented. After the <Admitted Hops> has been updated it is compared to the <Max Admitted Hops> from the <PDR container>. If the two values are equal then the partial release procedure should be terminated. This is achieved by setting the <I> flag to '1'. This updated RESERVE message is sent towards the QNE-Egress.

- At the next QNE-Interior the RESERVE message is processed and the RMD-QSpec extracted. The RMF inspects the container ID and the PHR container flags. The 'PHR_Release_Request' indicates

that the resources should be removed but the <I> flag is set to '1' and thus no resources will be released. At this point the QNE-Interior has processed the RESERVE message and a newly constructed RESERVE message (identical to the received message) is sent further downstream.

- The RESERVE message is finally received by the QNE-Egress node which will terminate the release procedure.

4.3.2 Admission Control

Using the protocol described in the previous section we can set up an aggregate reservation between a QNE-Ingress and a QNE-Egress node. The QNE-Ingress node can, after having set up such a reservation, map end-to-end flows on to the aggregate reservation. For every end-to-end reservation request arriving at the QNE-Ingress a check is performed whether the bandwidth requirements for that flow can be satisfied locally. This is done by the RMF, which checks whether the router's outgoing links have sufficient available bandwidth to grant the flow's bandwidth requirement. When aggregation is used, the RMF should check whether the aggregate reservation, for which the end-to-end flow is mapped on, has enough available resources for the end-to-end session. This is done as follows:

- First the RMF needs to establish on which aggregate session the end-to-end flow should be mapped. This is done using the per hop behavior (PHB) defined by the DiffServ architecture. Using the *<PHB Class>* parameter carried by the QSpec of the end-to-end flow and its destination QNE-Egress the appropriate aggregate session is chosen. If the initial QSpec does not contain the *<PHB Class>* parameter, the selection of the proper PHB is defined by a local policy similar to the procedures discussed in RFC 2998 [3] and RFC 3175 [2]. For example, in the situation that the initial QSpec is used by the IntServ Controlled Load QoSM, the appropriate PHB class to be used by the intra-domain RMD-QSpec would be the Expedited Forwarding (EF) PHB, see RFC 3175 [2]. Again we note that the RMD QoSM does not support the full set of Assured Forwarding (AF) PHBs but is limited to the use of one PHB class. For the purpose of this assignment we will not implement a fully functioning mapping function. Instead all end-to-end flows will be mapped to the Expedited Forwarding (EF) PHB.
- Let $flow\_bw\_req$ be the amount of bandwidth a particular end-to-end flow wishes to reserve. The flow is only admitted to the aggregate reservation if the condition $r_a + flow\_bw\_req \leq C_a$ holds, where r_a is the aggregate traffic rate[3] and C_a is the aggregate reservation size which is the amount of bandwidth reserved for the aggregate flow.
- If $r_a + flow\_bw\_req > C_a$ then the flow is not admitted to the aggregate reservation and the reservation request for the end-to-end session denied. If required a RESPONSE message is send to the requesting node.

If the flow is admitted then r_a is increased:

$$r_a = r_a + flow\_bw\_req \tag{4.1}$$

After the r_a parameter is updated, the end-to-end RESERVE message is forwarded downstream towards the QNR. Note that the QNE-Ingress marks the RESERVE message in such a way that it is not processed by the interior routers. The details of this marking process is discussed in section 4.2.1. The QNE-Egress removes the marking in order to recreate the original RESERVE message. This message is then forwarded towards the QNR. The QNR in turn can be instructed to send a response message back to the requesting QNI. In this case the RESPONSE message is processed by the QNE-Ingress and the QNE-Egress nodes through which the flow traverses. The QNE-Interior nodes are again bypassed using the same procedure defined for the RESERVE message.

[3]The aggregate traffic rate corresponds with the sum of rates of the end-to-end flows that make up the aggregate flow (also see section 1.4).

Note that in contrast to the reservation protocol discussed in [1], the end-to-end RESERVE message do not have to be synchronized with an intra-domain RESERVE message. An end-to-end reservation request has to pass the admission control check defined in Equation (4.1). This check does not require the QNE-Ingress to reserve bandwidth within the RMD domain. As long as the new value for r_a is less than the upper threshold Δ_{high} no increase of the aggregate is triggered and the end-to-end RESERVE messages are forwarded immediately after the reservation has been granted.

4.3.3 Increasing the aggregate reservation

After every admission control check the QNE-Ingress checks whether the aggregate reservation needs to be updated. As stated in section 2.2, a threshold based update scheme is used to trigger the aggregate reservation to increase or decrease its amount of reserved bandwidth. In this section we will discuss how the protocol can be used to increase the aggregate reservation size. This is done as follows:

- After an end-to-end flow has been admitted to the aggregate reservation, a check is performed to see whether the new r_a has exceeded an upper threshold. If this is the case, the aggregate needs to be updated. In other words if $r_a > \Delta_{high}$ then an increase of the aggregate reservation is triggered. This is achieved by raising the $TRIGGER\_INCREASE$ trigger.
- If the new r_a value does not exceed the upper threshold then the aggregate reservation is not updated and the end-to-end flow is simply admitted to the aggregate reservation.

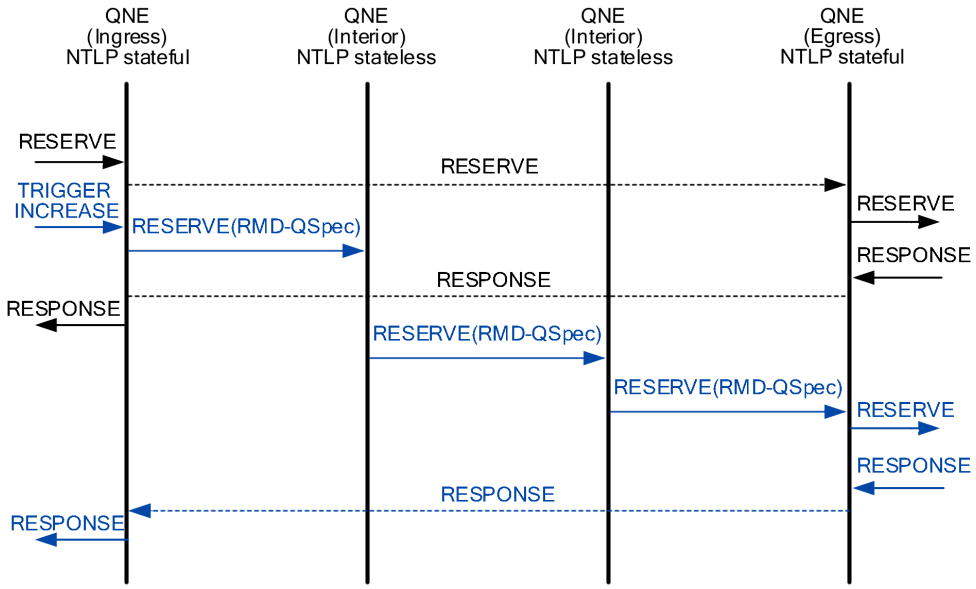


Figure 4.8: A successful increase of the aggregate reservation.

The increase procedure of the aggregate reservation is shown in Figure 4.8. The blue arrows depict the signaling messages associated with the increase of the aggregate reservation, while the black arrows depict the signaling messages associated with the end-to-end flow. In case an increase of the aggregate reservation is required, the new size of the aggregate reservation needs to be calculated (also see section 2.2):

$$C'\_a = r_a + \frac{OR}{2} \tag{4.2}$$

where C'_a is the new size for the aggregate reservation, r_a is the aggregate traffic rate and OR is the operation region. The calculation of the operation region OR is done as follows:

$$OR = \frac{T}{T_{av}} OR \tag{4.3}$$

where T is the target inter-update period and T_{av} is the average inter update interval based on previous measurements. The average inter update interval in turn is computed as follows:

$$T_{av} = \alpha T_{av} + (1 - \alpha) T_{curr} \tag{4.4}$$

where α is used as a smoothing factor for the exponentially weighted moving average and T_{curr} is the size of the current inter update interval.

If we want to calculate the amount of bandwidth needed to increase the aggregate reservation size to C'_a we use the following equation:

$$bw\_update = C'\_a - C_a \tag{4.5}$$

The *bw_update* is used to signal the interior nodes of the additional amount of bandwidth to be reserved for the aggregate reservation. Based on this parameter a new RMD-QSpec is created and sent towards the QNE-Egress. A RESERVE message including this QSpec is then sent towards the QNE-Egress which is processed by all the (NSIS aware) nodes in its path. This RESERVE message is formatted as follows:

- *The TEAR (T) flag is not set.*
- *The REPLACE (R) flag is not set.*
- *A newly generated <RII> object is included.*
- *A newly generated <RSN> object is included.*
- *A value of the <REFRESH_PERIOD> object is calculated and set by the QNE Ingress.*
- *A <PACKET_CLASSIFIER> object associated with the path-coupled MRM and with the <T> flag set is included.*
- *The RMD-QSpec for the RESERVE message has a <QoS Desired> object with the following parameters:*
  - *The 'Peak Data Rate' [p] field of the <Bandwidth> is set to the value of bw_update.*
  - *The <PHB class> is set to represent the EF PHB.*
  - *The <Admission Priority> parameter is set to '1' to represent the default priority.*
  - *A <PHR container> parameter included in the RESERVE message are set as follows:*
    - *The container ID field is set to 'PHR_Resource_Request'.*
    - *The value of the other fields is set to '0'.*
  - *A <PDR container> parameter is not included in this message.*

This RESERVE message is sent to the QNE-Interior nodes which will check whether the requested bandwidth can be reserved locally. Note that there is a small difference here that the QoS NSLP operational state and the QoS reservation have already been set up. When this RESERVE message is processed by the QoS NSLP processing component it will be regarded as a refreshing RESERVE. IN this case the QoS NSLP operational state is simply refreshed. The RMD-QSpec is processed by the RMF which will have to check whether there is a reservation state present. In this case the reservation state should be updated by increasing the amount of reserved resources (in the case that the request was granted).

If the additional bandwidth can not be reserved the M-Flag in the RMD-QSpec is set. Otherwise the bandwidth is reserved and the RESERVE message is forwarded. This RESERVE message is identical to the one received by the QNE-Interior with the exception that the <Admitted Hops> parameter is incremented.

The QNE-Egress processes the received RESERVE message and checks the relevant flags of the RMD-QSpec. If necessary a notification of the reservation setup or failure is send to the QNE-Ingress. In the case of a successful reservation the QNE-Egress sends back a RESPONSE message to the QNE-Ingress router indicating that the reservation of the additional bandwidth for the aggregate reservation was successful. The INFO_Spec object is used to notify the QNE-Ingress if the reservation was successful or not. In case of a successful reservation an INFO_Spec parameter with the following values is included in the RESPONSE message:

Error Severity Class: Success
Error Code value: Reservation successful

In the case that the reservation was unsuccessful the INFO_Spec parameter is set as follows:

Error Severity Class: Transient failure
Error Code value: Reservation failure

The message sequence diagram for an unsuccessful increase of the aggregate reservation is shown in Figure 4.9. Also see section 4.3.1 for a more detailed discussion of the successful and unsuccessful RMD reservation procedures.

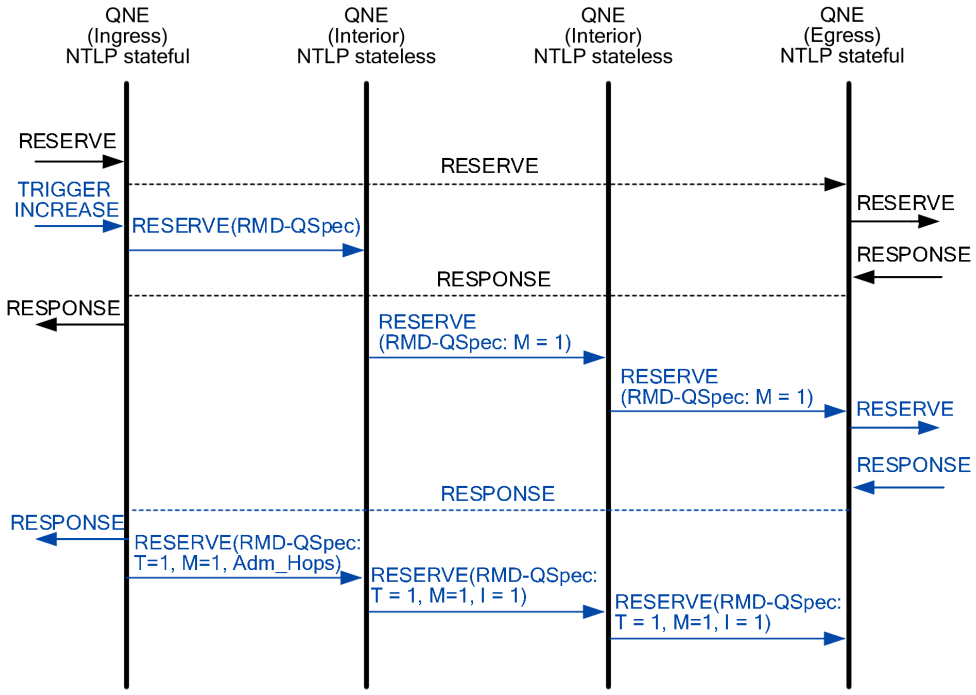


Figure 4.9: An unsuccessful increase of the aggregate reservation.

Once the QNE-Ingress receives the RESPONSE message it checks whether the reservation of the additional bandwidth for the aggregate reservation was successful. This check is performed by inspecting the error value of the INFO_Spec object in the RESPONSE message. If the reservation was successful then

the C_a parameter is increased by the reserved amount *bw_update*. The new value of the aggregate size is thus computed by $C_a = C_a + bw\_update$.

If the request for additional bandwidth was rejected then the *bw_update* parameter is set back to 0. In addition a RESERVE tear message is created to release the (partially) reserved bandwidth at certain interior routers.

For end-to-end reservation requests arriving at the QNE-Ingress during the update of the aggregate reservation, the admission control is performed based on the C_a parameter as stated above. Note that this admission control check is performed based on the 'old' size if the aggregate reservation rather then the new, yet unconfirmed, aggregate reservation size value, C'_a. Thus an end-to-end reservation request is granted only if $r_a + flow\_bw\_req \leq C_a$. In the case that this condition does not hold the reservation request is rejected. It should be noted in this case that the reservation is granted, the r_a parameter could possible exceed the Δ_{high} again. This would trigger another update of the aggregate reservation which is not desired. Thus if the increase of the aggregate reservation is triggered we need to perform another check to see whether the aggregate reservation is already being updated.

If an increase of the aggregate reservation is triggered is only triggered if T_{min} is equal to zero. T_{min} is a timer which is set whenever the first request for additional bandwidth is triggered. The timer counts down to zero and the next request can only be send when the timer has reached zero.

In addition to the update of the C_a and C'_a parameters, the threshold values Δ_{high} and Δ_{low} are recomputed. Their new values are calculated as follows:

$$\Delta_{high} = C_a - cushion \tag{4.6}$$

$$\Delta_{low} = C_a - OR \tag{4.7}$$

where *cushion* is the size of the cushion and OR the size of the operation region. Note that the new values for the thresholds are computed after the increase of the aggregate reservation increase has been confirmed and the C_a parameter has been updated. The calculation of the value for the cushion is done as discussed in section 2.2.1.

$$cushion = k \times x_{bw} \tag{4.8}$$

where k is the minimum value for k such that the sum $\sum_{i=0}^{k-1} \frac{e^{-\lambda \Delta t_{rl}} (\lambda \Delta t_{rl})^i}{i!}$ is greater than or equal to $1 - \epsilon$ and x_{bw} is the bandwidth requirement for one flow (also see Equation (2.32)).

4.3.4 Decreasing the aggregate reservation

A decrease of the aggregate reservation is triggered by an end-to-end flow leaving the aggregate session. This could be due to an explicit release or due to a failure of the reservation request. In the case of an explicit release the QNE-Ingress receives a RESERVE message with the tear flag (T-Flag) set to one from the QNI. In the case of a reservation failure the QNE-Ingress receives a RESPONSE message form the QNR containing an INFO_Spec with an error code indicating that the request was rejected somewhere along the path. Whenever an end-to-end flow leaves the aggregate session the r_a parameter is updated:

$$r_a = r_a - flow\_bw\_req \tag{4.9}$$

where $flow\_bw\_req$ is the reserved bandwidth for the flow leaving the aggregate session. After the r_a parameter is updated a check is performed to see whether the new value for r_a has dropped below the Δ_{low}. So if $r_a < \Delta_{low}$ then an decrease of the aggregate reservation is triggered. The message sequence diagram for the decrease of the aggregate reservation is shown in Figure 4.10. The new value of the aggregate reservation size is computed as follows:

- First the new value for the operation region OR is calculated:

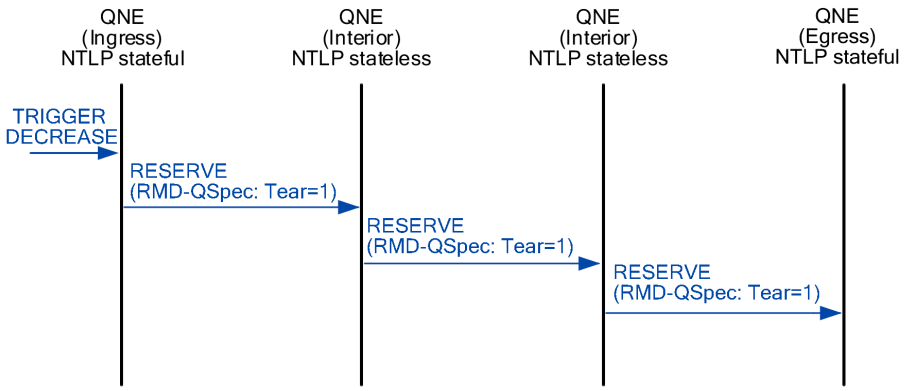


Figure 4.10: An decrease of the aggregate reservation.

$$OR = \frac{T}{T_{av}} \times OR \tag{4.10}$$

where OR is the size of the operation region calculated using Equation (4.3) and T_{av} is the average inter update period calculated using Equation (4.4).

- After the new value of the OR has been calculated the new aggregate size is calculated as follows:

$$C'\_a = C_a - \frac{OR}{2} \tag{4.11}$$

where C_a is the current aggregate reservation size and C'_a is the new size for the aggregate reservation. The amount of bandwidth to be released is calculated as follows:

$$bw\_update = C_a - C'\_a \tag{4.12}$$

Based on the parameter *bw_update* an RMD-QSpec is created. The parameters in the RMD-QSpec are set as follows:

- *The RMD-QSpec for the RESERVE message has a <QoS Desired> object with the following parameters:*
  - *The 'Peak Data Rate' [p] field of the <Bandwidth> is set to the value of bw_update.*
  - *The <PHB class> is set to represent the EF PHB.*
  - *The <Admission Priority> parameter is set to '1' to represent the default priority.*
  - *A <PHR container> parameter included in the RESERVE message are set as follows:*
    - *The container ID field is set to 'PHR_Release_Request'.*
    - *The value of the <M> field is set to '1'.*
    - *The value of the other fields is set to '0'.*
  - *A <PDR container> parameter is not included in this message.*

A RESERVE message with the tear flag (T-flag) **not** set and the RMD-QSpec included, is sent towards the QNE-Egress. This way the QNE-Interior nodes know that they have to release the requested bandwidth. Note that the T-flag of the RESERVE message is not set because the QNE-Interior and QNE-Egress nodes do not have to tear down the QoS NSLP operational state. The aggregate session is supported to be kept intact. It is only the QoS reservation state that needs to be updated. Again here the reservation state should not be deleted but the value of the <Bandwidth> parameter subtracted form the reserved resources.

In addition to sending the RESERVE the C_a, C'_a parameters and the thresholds need to be updated. After a request for the release of bandwidth is sent towards the QNE-Egress the size of the aggregate reservation is set to the new aggregate reservation size ($C_a = C'\_a$). The parameter C'_a is reset by setting its value to -1. The new threshold values are calculated as follows:

$$\Delta_{high} = C_a - cushion \tag{4.13}$$

$$\Delta_{low} = C_a - OR \tag{4.14}$$

The value of the OR using Equation (4.3) and the value of the cushion is calculated using Equation (4.8). Notice that the C_a parameter is immediately updated to its new value, unlike in the event of an increase of the aggregate reservation. This is because the release of bandwidth does not need to be confirmed.

4.3.5 Refreshing the aggregate reservation

In RMD soft-states are used which need to be refresh periodically. This process is similar to the setup or increase of the aggregate reservation. The message sequence diagram for the refresh of the aggregate reservation is shown in Figure 4.11.

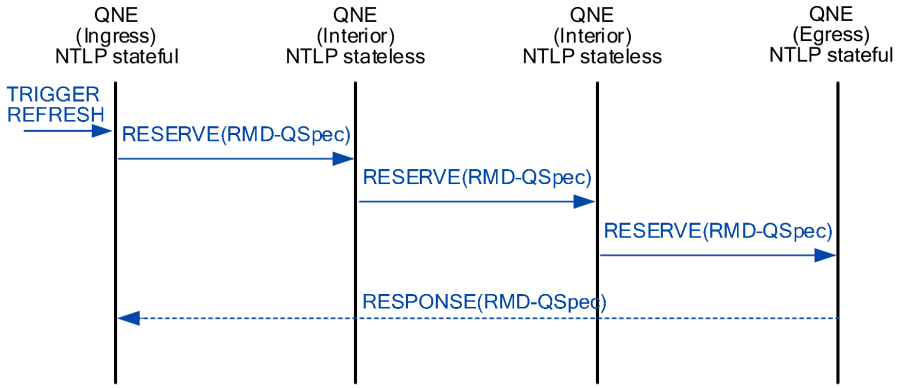


Figure 4.11: A refresh of the aggregate reservation.

At the QNE-Ingress a trigger (TRIGGER_REFRESH) is released whenever the RMD states need to be refreshed. Note that the refresh periods must be equal at all QNE nodes in the RMD domain. This refresh period should also be smaller (by default more than two times smaller) than the refresh period used the end-to-end QoSM at the edge QNEs. The structure of the RESERVE message is defined as follows:

- *The TEAR (T) flag is not set.*
- *The REPLACE (R) flag is not set.*
- *A newly generated <RII> object is included.*
- *A newly generated <RSN> object is included.*
- *A value of the <REFRESH_PERIOD> object is calculated and set by the QNE Ingress.*
- *A <PACKET_CLASSIFIER> object associated with the path-coupled MRM and with the <T> flag set is included.*
- *The RMD-QSpec for the RESERVE message has a <QoS Desired> object with the following parameters:*
  - *The 'Peak Data Rate' [p] field of the <Bandwidth> is set to the the amount of resources reserved for the aggregate reservation.*
  - *The <PHB class> is set to represent the EF PHB.*
  - *The <Admission Priority> parameter is set to '1' to represent the default priority.*
  - *A <PHR container> parameter included in the RESERVE message are set as follows:*
    - *The container ID field is set to 'PHR_Refresh_Update'.*
    - *The value of the other fields is set to '0'.*
  - *A <PDR container> parameter is not included in this message.*

At the QNE-Interior nodes the intra-domain RESERVE message carrying the RMD-QSpec is received and processed. The 'PHR_Refresh_Update' indicates that the QNE-Interior node should refresh the QoS reservation state is has stored. In doing so the QNE-Interior must identify the traffic class state (PHB) (using the <PHB Class> parameter). For every traffic class a reservation state is stored at the QNE-Interior nodes.

Note that the QNE-Interior nodes do not store reservation states for the aggregate flows but store the reservation state for the traffic classes of the flows traversing the nodes. This is different to RSVP where trunk reservations are stored for the aggregate reservations. In other words, the amount of reservation states stored at an interior node is equal to the amount of aggregate flows traversing the interior node (for every traffic class). In RMD one reservation state per traffic class is stored at the QNE-Interior nodes. Because there can be multiple aggregate flows manipulating the same reservation state a refresh timer is used to refresh the reservation. It works as follows:

- At the beginning of the refresh period the QNE-Interior node sets the $current\_refreshed$ counter to zero. The $current\_refreshed$ counter represents the amount of bandwidth refreshed within the current refresh period.
- Upon receiving a refreshing RESERVE message the counter is increased with the amount of bandwidth specified in the <Bandwidth> parameter in the RMD-QSpec. The '*Peak Data Rate [p]*' value of <Bandwidth> parameter is used for refreshing the reservation state. The value of the <Bandwidth> parameter is added to the $current\_refreshed$ counter. After the $current\_refreshed$ counter has been updated it is compared to the currently reserved resources C_a. If the $current\_refreshed$ counter is larger than C_a then extra bandwidth needs to be reserved. In this case the QNE-Interior node attempts to reserve the extra bandwidth. If the bandwidth is granted then the C_a counter at the node is updated to be equal to the $current\_refreshed$ counter.

  If the bandwidth can not be reserved then the refreshing of the reservation state is considered to have failed. In this case the the <M> flag of the <PHR container> parameter is set to '1'

indicating that the refresh has failed. In addition the <Admitted Hops> and <Hop_U> parameter is set accordingly. The $current\_refreshed$ counter is set back to its previous value.

- At the end of the refresh period the QNE-Interior sets its r_a parameter to the value of the $current\_refreshed$ counter. After the r_a parameter has been updated, the $current\_refreshed$ is reset to zero and the refresh process starts from the beginning.

If the reservation state has been refreshed successfully refreshed then the QNE-Interior node forwards the RESERVE message downstream. The refresh procedure is repeated at the neighboring QNE-Interior node until it eventually is delivered to the QNE-Egress node. At the QNE-Egress the RESERVE message is processed and the M-flag inspected. If the M-flag is set then the refresh of the aggregate reservation has failed and the 'RMD partial release procedure' should be triggered. This is done by including a INFO_Spec parameter set as follows:

Error Severity Class: Transient failure
Error Code value: Reservation failure

In case of a successful refresh an INFO_Spec parameter with the following values is included in the RESPONSE message:

Error Severity Class: Success
Error Code value: Reservation successful

The complete RESPONSE message is constructed as follows:

- *The <RII> object carried by the RESERVE message is copied and added to the RESPONSE message.*
- *Included in the RESPONSE message is an INFO_Spec object with the values set as described above.*
- *The RMD-QSpec for the RESPONSE message has a <QoS Reserved> object with the following parameters:*
  - *The <Bandwidth>, <PHB class> and <Admission Priority> parameters are copied from the received RESERVE message.*
  - *A <PDR container> parameter included in the RESERVE message are set as follows:*
    - *The container ID field is set to 'PDR_Refresh_Report'.*
    - *The value of the other fields is set to '0'.*

This RESPONSE message is sent to the QNE-Ingress node. Here it is processed and if the INFO_Spec is set to the following value, the partial release procedure is initiated:

Error Severity Class: Transient failure
Error Code value: Reservation failure

If the INFO_Spec is set to '*Reservation successful*' then the refresh is considered to have been refreshed successful.

In the next chapter we will discuss how the aggregation based RMD-QoSM model specified here is implemented in our prototype implementation.

Chapter 5

Design and Implementation

In this chapter will discuss the design and implementation of the aggregation based RMD-QoSM which was specified in the previous chapter. First we will discuss the design of the current prototype implementation developed by Ruud Klaver [31] and Martijn Swanink [35]. Next we will show how the prototype can be extended to support aggregation based RMD-QoSM.

5.1 Previous Work

The current prototype was developed by Ruud Klaver [31] and Martijn Swanink [35] in the DACS laboratory at Twente University. It consists of two separate implementations: one implementing the GIST functionality, the other implementing the reservation based RMD-QoSM. Ruud Klaver is responsible for the implementation of the GIST functionality. His implementation runs on a Linux operating system and was created using the Python (2.5) programming language. Martijn Swanink has worked on the implementation of the (reservation based) RMD-QoSM and has also implemented some of the NSLP layer functionality. His implementation was programmed in the C programming language and runs on the Linux operating system. Although the two implementations have been coded in different languages they interact with each other flawlessly.

The choice for the Linux operating system is obvious because it is *open source* and there is a large number of *open source* applications available for the operating system. *Open source* means that the source code of the application is distributed with the application and can be modified freely. Thus when developing an application for the Linux operating system one can easily look into the source code of other applications that have implemented similar functionality. This makes Linux the ideal platform for our prototype implementation. An additional advantage of using Linux is the fact that it is free. This is particular advantageous for our test environment in which five independently running Linux operating systems are needed. The Linux distribution used for our testing is Gentoo Linux 3.4.5 running the Linux kernel version 2.6.14-gentoo-r3.

For the purpose of this assignment we must implement the aggregation based RMD-QoSM discussed in Chapter 4. Here we are presented with two options:

1. *We can extend the implementation developed by Martijn Swanink.* This would imply that our the extension would have to be programmed in the C programming language which has several disadvantages. First and foremost, the C programming language is a procedural language lacking functionality for object oriented programming. Although an object oriented programming is not a must, it certainly has its benefits when using Rapid Application Development (RAD). In addition, C is a low-level programming language and does not support features like automatic garbage collection, exception handling and advanced multithreading. These features are all necessary for the implementation of the aggregation based RMD-QoSM. Multithreading is needed for the simulation of the many end-to-end flows that are to be aggregated. Automatic garbage collection significantly relieves the programmer of having to explicitly allocate and free blocks of memory, which is a primitive and error-prone in C. In our implementation we will set up, maintain and tear down many

NSLP operational and reservation states, making automatic garbage collection a nice feature to have. Exception handling is particularly useful when debugging an application because when an exception is thrown a complete stack track is included, from which the exact executing method causing the error can be retrieved. This allows more efficient debugging compared to C's infamous core dumps.

On the plus side, C has the advantage that it generates highly optimized code because of its low level nature. Because we are implementing a QoSM that needs to aggregate many flows this would be a nice feature to have.

2. *We can continue the development of the prototype implementation created by Ruud Klaver.* The main purpose of this implementation was to create a proof-of-concept for performing dynamic QoS reservations for multimedia application using the NSIS signaling framework. Although the overall focus of the assignment was on designing and implementing the GIST functionality, some basic NSLP functionality had to be implemented in order to fully test the GIST implementation. The implementation was tested using the RMD-QoSM implementation of Martijn Swanink but a basic QoSM was implemented and used for additional testing. The QoSM used was the *IntServ Controlled Load (ISCL)* QoSM. This QoSM allows applications to reserve per-flow resources at the nodes in the network it traverses.

   Because the main focus of this implementation was not to implement a fully featured NSLP layer, the implemented functionality is very basic. Also, the NSLP functionality was not implemented in a modular fashion. Thus extending this implementation will require some changes in the design. The ISCL QoSM implementation can certainly be re-used but the other NSLP functionality will require some serious refactoring.

Of the two options presented above we have chosen the latter. Our aggregation based RMD-QoSM will be implemented as an extension on the prototype implementation of Ruud Klaver. The main reason why this implementation was chosen is because of the programming language used. Python is an object oriented programming language which offers many advantages which have been described best in [31]:

> *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. (...) 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 (...).*

In essence Python allows us to do RAD in return for sacrificing some performance. For our prototype implementation this disadvantage does not weight against the benefits of using Python. Another advantage of using Python over C is the fact that it has better multithreading support. Multithreading is needed for the simulation of the arrival and departure of many concurrent end-to-end flows. Another advantage offered by the prototype implementation of Ruud Klaver is the implementation of a simple end-to-end QoSM. This QoSM can be used as a reference for our QoSM implementation and can be used for the simulation of the end-to-end flows. As it happens, for the end-to-end flows in our simulation an end-to-end QoSM is needed and the ISCL QoSM satisfies this requirement.

5.2 Implementation overview

In this section an overview of the prototype implementation is given. First the design of the implementation provided by Ruud Klaver is presented, followed by a discussion on how it can be extended to support our aggregation based RMD-QoSM.

5.2.1 Current design

In this section the current design of GIST is explained. Afterwards the NSLP functionality implemented in the prototype will be discussed. In Figure 5.1 the components that make up the current GIST implementation are shown.

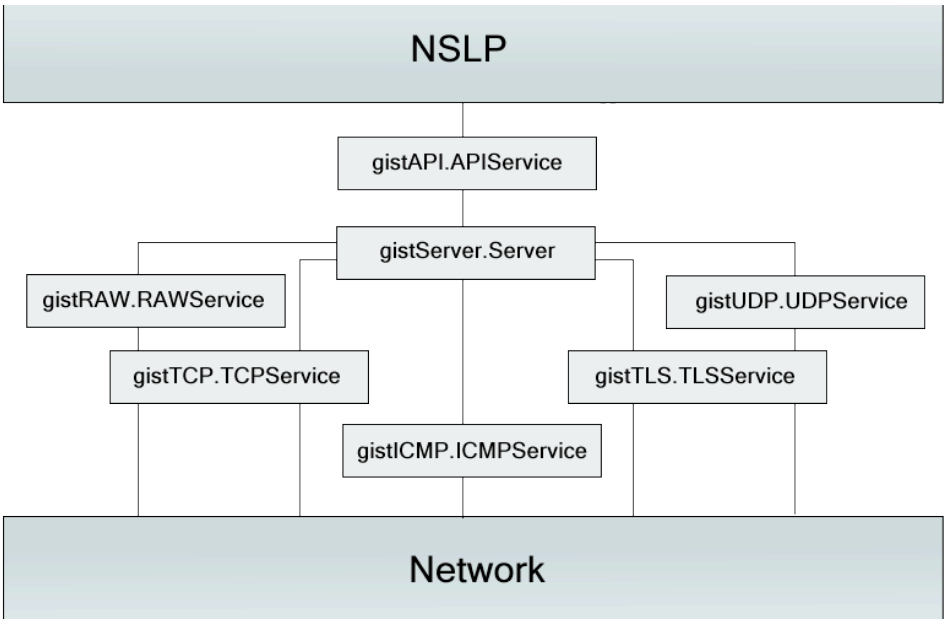


Figure 5.1: The GIST implementation overview.

The main class here is the *gistServer.Server* class. This server class contains instances to several other classes, which work as threads and provide different types of services to the server. The threads of these service classes monitor their specific domain and notify the GIST server of these incoming events. In this design each established connection has its own thread, which is used to handle the network traffic for that connection. A single thread per connection is needed because reading data from a connection on which nothing is received results in a so-called blocking call, halting program execution at that point. By using threading and proper exception handling the server thread can remain active and be notified of errors occurred during the reading and writing of data from or to a connection.

The service classes all have their own domain for which they are responsible. In the current GIST implementation the following service classes are supported:

gistRaw.RawService: This class is responsible for sending and receiving of messages in *GIST-Query Mode*. The GIST-Query Mode is a transmission mode used for the discovery of downstream peers. The name of this service class is derived from the method by which the messages are sent, using raw sockets.

gistUDP.UDPService This class is used for sending and receiving messages in *GIST-Datagram Mode*, using normal UDP encapsulation. The GIST-Datagram Mode is primarily used to transmit data unreliably and insecurely between nodes.

gistTCP.TCPService This class handles incoming and outgoing TCP connections. This service class is primarily used as the GIST-Connection Mode protocol. The GIST-Connection Mode allows nodes to reliably send data with specific network or transport layer security. 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 GIST server.

gistTLS.TLSService This class handles incoming and outgoing TLS over TCP connections. This class is specifically used for the flows sending messages with the GIST Security *Transfer Attribute* set to 'true'. TLS connections are also managed by their own class, *gistTLS.TLSTCPConnection*.

gistICMP.ICMPService This class monitors *Internet Control Message Protocol* (ICMP) traffic sent to the node. ICMP messages are mostly used to send error messages, indicating for instance that a requested service is not available or that a host or router could not be reached. In our prototype implementation ICMP messages are mainly used to deduce the last NSIS capable node on the path.

In Figure 5.1 a *gistAPI.APIService* class is also displayed. This class provides the communication between GIST and the local NSLP applications, which is achieved using the service primitives defined in GIST API (see Section 3.3.1). In the *gistAPI.APIService* class an instance of the *gistServer.Server* and an instance of the *qosServer.QOSServer* are maintained. Interaction between the two instances is done using this service class. In this class the interfaces for the *SendMessage*, *SetStateLifetime* and *InvalidRoutingState* service primitives are defined. Using these service primitives messages and notifications are send from the NSLP layer to the GIST. GIST on its turn can communicate with the NSLP layer using the *RecvMessage*, *MessageStatus* and *NetworkNotification* service primitives of the NSLP layer. In Figure 5.2 an overview of the relevant classes in their corresponding layers is given.

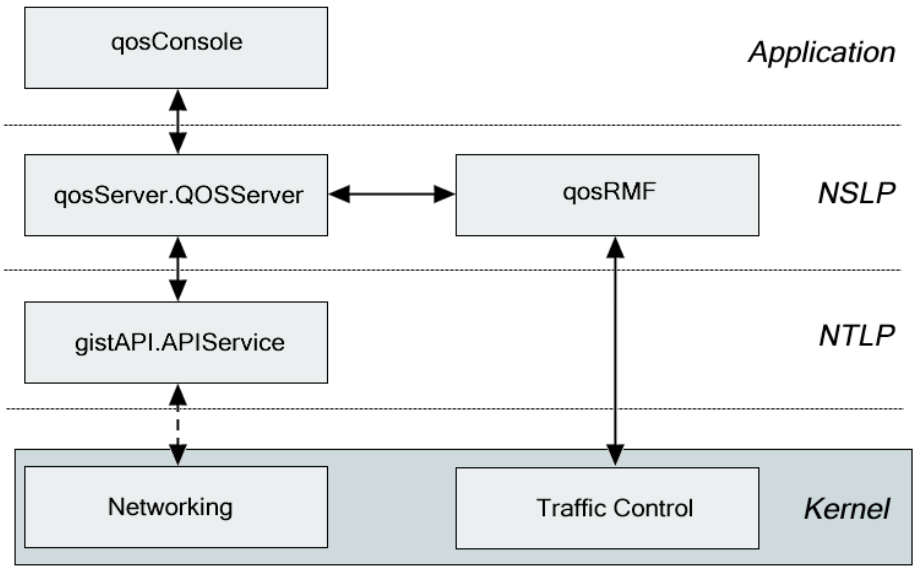


Figure 5.2: NSLP implementation overview.

In Figure 5.2 we see how the prototype is built up. A layered approach is used in this design separating the NTLP, NSLP and Application functionality. The kernel is explicitly shown because here the network and traffic control functionality is located. The traffic control in the kernel corresponds to the *Traffic Control* component shown in Figure 3.2 (see Section 3.2.1) where the flow of data packets is controlled. In the networking component located in the kernel the *Input Packet Processing* and *Output Packet Processing* components can be found. These components handle incoming and outgoing messages respectively. The relation between the *gistAPI.APIService* class and the *Networking* component is depict by a dotted line because they are not connect directly to each other. As shown in Figure 5.1 there is a *gistServer.Server* and a service class located between the *gistAPI.APIService* class and the *Networking* component. These classes have been omitted here for clarity.

In the NSLP layer the following two classes are shown:

qosServer.QOSServer: The *qosServer.QOSServer* class represents the message handling component of the NSLP layer. It is equivalent to the *QoS NSLP Processing* component shown in Figure 3.2. Here messages received from the NTLP layer using the *RecvMessage* service primitive are processed. Note that the *MessageStatus* and *NetworkNotification* have also been implemented in this class but the bulk of the processing is done in the *RecvMessage* method. There is a small difference between the implementation and the reference model, shown in Figure 3.2, is the communication between the NSLP and the local application. In the reference model a local application interacts directly with the *Resource Management Function*, while in this implementation all interactions between the NSLP layer and the local application is handled by the *qosServer.QOSServer* class.

qosRMF: The *qosRMF* module represents the Resource Management component. Here the actual reservation and release of resources is done. In this class the different QoS models can be implemented. In the current implementation only the *IntServ Controlled Load* QoSM has been implemented:

- **qosRMF.ISCL**: This class represent the *IntServ Controlled Load* QoSM. This QoSM uses the IntServ architecture where per-flow QoS guarantees are provided. "Controlled load" here means that a flow receives the equivalent treatment of a best effort flow in a lightly loaded network. This QoSM is a simple one having only two QSpec parameters: a token bucket and an excess treatment parameter (which is optional). The latter parameter specifies what actions should be taken when a flow exceeds the given parameters, e.g. dropping or reclassifying its packets.

Aside from the two classes discussed above the following helper classes can be found in the NSLP layer of our prototype:

qosAPI: Here an interface for communication between the *Application* layer and *NSLP* layer is defined. Communication between an application and the NSLP layer is done using a socket connection. In this class some parameters have been defined which correspond to the actions that can be executed. The following actions have been defined:

- **Kill**: this message instructs the *QOSServer* to shutdown and discontinue operation.
- **List**: this message asks the *QOSServer* to return a list of all the reservations that are currently in place at this node.
- **Add**: this message instructs the *QOSServer* to install a new reservation and initiate signaling for it.
- **Del**: this message instructs the *QOSServer* to remove state for a previously installed session and start an explicit tear down procedure for it.

qosException: In this class most error handling is done. Every error is represented by a different class. In each error class the *Error Class* and *Error Code* is defined. Additionally an INFO_Spec object is defined in the class which can be used to transfer the error message.

qosIface: This class is used for low level interaction with the kernel. Specifically, it is used to get information about the network interfaces.

qosMsg: This class defines the structure of an NSLP signaling message. All the QoS objects and parameters defined in the QSpec template [13] have been implemented here.

qosRSN: This class is a special helper class which is used to generate a reservation sequence number. This (RSN) object is used to set the state modification actions in the correct order. This class is implemented according to the RFC 1982: "*Serial Number Arithmetic*" [10].

qosState: The *qosState* module contains the QoS NSLP operational state class:

- **qosState.PersistentState**: In the *qosServer.QOSServer* class QoS NSLP persistent session state is stored. It is indexed per Session ID (SID) and is represented by instances of the *qosState.PersistentState* class. Here information about sequence numbers (RSN), Request Identification Information (RII) objects, Message Routing Information (MRI), refresh and expiry timers are stored.

In the current implementation a small console base application has been implemented which can be used for testing purposes. The application is represented by the *qosConsole* class and can perform basic operations like set up and tear down end-to-end reservations. This class is mainly used for testing purposes.

5.3 Extension on the design

In this section we will discuss the extensions made on the current design in order to support our aggregation based RMD-QoSM. In short, the following changes on the current prototype are required:

- *Implementation of the aggregation based RMD-QoSM.* In Chapter 4 we have specified how our aggregate update policy can be incorporated in the RMD-QoSM. This QoSM will be added to the QoSM implemented in the **qosRMF** module, as the *qosRMF.RMD* class.
- *Implementation of a Aggregate Management Unit (AMU).* Up till now we have talked about the aggregate update policy as if it was part of the aggregation based RMD-QoSM. In our design however, we will model it as a separate object, the Aggregate Management Unit (AMU). This class is responsible for the updating of the aggregate reservation. Here the future aggregate traffic is predicted and based on this prediction the RMF is instructed to increase or decrease the aggregate reservation. In the Section 5.3.3 we will discuss this component in more detail. There are several reasons for modeling the AMU as a separate component:
  1. This way a clear overview of the responsibility of the different classes can be shown. From a design perspective the RMF is responsible for the setup and tear down of resources at a node. We feel that this does not include updating the reservation according to some update algorithm. This would be the responsibility of an external component which will instruct the RMF to setup, tear down or modify the resources reserved at the node. In other words the RMF's only role is to reserve resources, release resources and update the amount of resources reserved at the node, when it is instructed to do so. The RMF thus may not decide when resources should be reserved or how much resources should be reserved, it simply needs to execute the actions triggered by other components (mainly the *qosServer.QOSServer* class).
  2. Strictly speaking the mechanism used to determine how the aggregate reservation should be updated is not part of a QoSM. A QoSM specifies how message are constructed and processed and can use an aggregate update policy to determine when and how to update the aggregate reservation. By modeling a separate component which implements an aggregate update policy we can now easily add support for flow aggregation to other QoSMs.
  3. In addition we can design the aggregate management unit in such a way that different update policies can be used to update the aggregate reservation. Note however we will not implement aggregate update policies other than the one specified in our proposed solution. In our design we will take into account that different update policy can be used and supply the means to allow the easy implementation of other methods.
- *Explicitly implement a ReservationState class.* In the current implementation no explicit reservation state class has been implemented. Instead, the amount of bandwidth reserved at a particular node is stored in the *qosState.PersistentState* class. This class should only be used to store the parameters which are valid for a particular signaling session. The reason why the amount of reserved bandwidth is stored here is because the parameter is too simple to model as a class. While this is certainly

possible in the current implementation, for the implementation of our aggregation based QoSM the amount of reserved bandwidth needs to be maintained separately. The reason why we need to explicitly model a reservation state class is because the QNE-Interior nodes do not store any NSLP state but they do store per traffic class reservation state. Also the refresh procedure of the amount of reserved resources at the QNE-Interior nodes required us to implement a more complex reservation state object.

- *Implement the QoS Server as a daemon.* The current QoS NSLP implementation was coded to work with one simple QoSM, the *IntServ Controlled Load (ISCL) QoSM*. In other words, it was not designed to be used with multiple QoSMs. Because of that several changes need to be made to the *qosServer.QOSServer*. First of all a clear distinction has to be made between the signaling functionality and the functionality of the QoSM. After this has been achieved the message processing and QoSM specifications can be implemented in different classes. In our design a statemachine is used to handle the NSLP signaling message processing. By implementing the signaling as a statemachine, the *qosServer.QOSServer* will now serve as a daemon which is only responsible for the management of the statemachines. For the design of this statemachine, the statemachine designed by Fu et al. [37] will be used as a base.

- *Implement a console application which can be used for testing.* Finally the **qosConsole** application will be extended, so that different end-to-end flows can be simulated. In order to achieve this will will implement the following option in the application:

  Sim: When this message is passed to the console application, a trace file will be open containing the details of the flows in the aggregate traffic. The file consists of lines which each define the start time, holding time and bandwidth requirement for one end-to-end flow. A flow is simulated using threads.

The changes defined above have led to a new design of the NSLP layer. An overview of this design is shown in Figure 5.3.

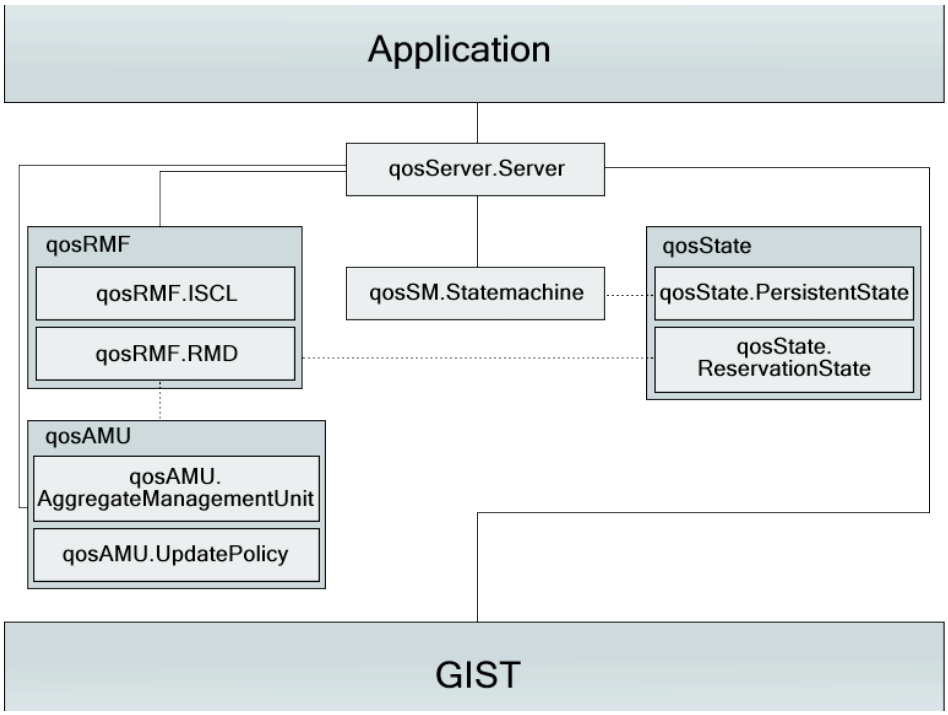


Figure 5.3: The QoS NSLP design overview.

The old design of the NSLP layer consisted mainly of the *qosServer.Server* class and the *qosRMF.ISCL* class. In this design the following components have been added: the **qosAMU** module, the *qosRMF.RMD* class, the *qosState.ReservationState* class and the *qosSM.Statemachine* class. The *qosServer.QOSServer* class has also been modified extensively in order to work correctly with the other components. The *qosState.ReservationState* class represents the reservation state and the *qosSM.Statemachine* class represents the NSLP statemachine, which we will discuss in Section 5.3.1. The *qosRMF.RMD* class represents the aggregation based RMD QoSM which has been defined in the previous chapter and the **qosAMU** module contains the functionality for our Aggregate Management Unit, which we will discuss in Section 5.3.3.

The central component in Figure 5.3 is the *qosServer.QOSServer*. This class is responsible for the handling of incoming signaling messages. These messages can be received from either GIST or applications desiring QoS. In the previous design of the prototype implementation the *qosServer.QOSServer* contained all the logic for the processing of the NSLP signaling messages, according to the *'General Processing Rules'* defined in [18]. In the current design this logic is moved from the *qosServer.QOSServer* class to a separate class. This way the *qosServer.QOSServer* class will simply act as a daemon that forwards incoming messages to other components for processing. The actual processing of the incoming messages is done using a statemachine, implemented in the *qosSM.Statemachine* class. This statemachine will be discussed in the next section.

The reason why a statemachine is used to model message processing is because it gives a nice overview of the processing rules. This makes debugging easier because one can easily trace back the state of the signaling session and at which transition the error occurred. Also by modeling the processing of the signaling message as separate component one can easily see where the responsibility of the NSLP signaling stops and that of the QoSM starts. A clear separation between the roles of the two components is vital for a robust implementation.

The *qosServer.QOSServer* class is also connected to the **qosRMF** module. In this module the QoSM implementations are located. In this design there are two QoSMs: (i) the *ISCL-QoSM* used for end-to-end QoS and (ii) the *RMD-QoSM* used for QoS within a RMD domain. The *qosRMF.RMD* class implements the RMD QoSM discussed in Chapter 4; the reader is also referred to [1] for a complete specification of the RMD QoSM. The details of the *qosRMF.RMD* class implementation are discussed in Section 5.3.2.

As stated in the previous section the RMF is only responsible for updating the reservation state. This is depicted in Figure 5.3 by a relationship between the *qosState.ReservationState* class and the *qosRMF* module. Managing resources in this design means managing the *qosState.ReservationState* instances. In a single end-to-end signaling session, one *qosState.PersistentState* instance and one - *qosState.ReservationState* instance are maintained by the **qosRMF**. In an aggregate signaling scenario the end-to-end flows will all have a single *qosState.PersistentState* instance while the aggregate session has a *qosState.PersistentState* instance and a *qosState.ReservationState* instance storing the reserved bandwidth for the aggregate reservation. The *qosState.ReservationState* class represents the resources reserved at a particular node for a particular session. Currently this class only stores reserved bandwidth information. It also contains a timer used in the reservation refresh procedure at the QNE-Core nodes (see Section 4.3.5).

The *qosRMF.RMD* class uses the **qosAMU** module for the management of the aggregate reservation. It instructs the *qosAMU.AggregateManagementUnit* class to admit a flow to the aggregate or to remove a flow from the aggregate. The admission control for the end-to-end flows is performed at this class. The *qosAMU* module also has a relationship with the *qosServer.QOSServer*, because this is the class that controls the aggregate signaling session and all triggers for an increase or a decrease of the aggregate reservation should be handled by it The design details of the **qosAMU** is discussed in Section 5.3.3.

5.3.1 The NSLP statemachine

As stated in the previous section, the NSLP statemachine is responsible for the correct processing of NSLP signaling messages. A statemachine is used because it gives a clear view of how the signaling messages are processed. As a basis for our statemachine we will use the statemachine designed by Fu et.al. [37], in which QoS signaling messages are processed according to the message processing rules defined in [18]. A complete overview of this statemachine is given in [37], however in this section we will not discuss the entire statemachine. In our design a subset of this statemachine is used, containing only the states and transactions relevant for our prototype. This reduced statemachine and its differences with the statemachine defined in [37] are discussed in this section.

Before going into the details of this statemachine we will discuss how the statemachines are maintained. For every signaling session a statemachine is created and maintained at a QNE node. The creation and removal of these statemachines are coordinated by the QoS daemon (qosd) (see Figure 5.4).

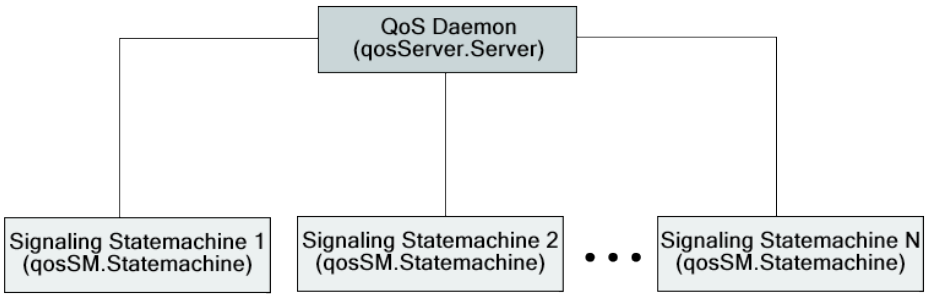


Figure 5.4: The QoS daemon.

The *qosd* listens for incoming messages from either a local application or from GIST. From a local application the *qosd* can receive triggers to setup or tear down a reservation or to query the network for resource availability. From GIST the *qosd* can receive one of the following the QoS signaling messages RESERVE, RESPONSE, QUERY or NOTIFY.

The *qosd* is implemented by the *qosServer.QOSServer* class. This class uses the GIST API (implemented by the *gistAPI.APIService* class) to read out the messages received from other nodes. More specifically the *RecvMessage* service primitive is used to pass on the signaling messages from the GIST layer to the NSLP layer. The *qosAPI* is used to read out the messages sent by a local application.

Whenever the *qosServer.QOSServer* receives a message, either from the local application or from GIST, it checks whether there is a statemachine present that can be used to process this message. Internally the *qosServer.QOSServer* maintains a lookup table for all the statemachines present at the QNE. In our implementation this lookup table is a Python dictionary named *statemachinedict*. For every signaling session a statemachine is created and a reference stored in this lookup table. These references are keyed by their session id (SID). In other words the SID can be used to lookup the corresponding statemachine for this session. When the *qosServer.QOSServer* receives a message it can use the received SID parameter to lookup the statemachine. If the lookup table *statemachinedict* has a reference to a statemachine for the SID then this statemachine is used. If *statemachinedict* does not have a reference to a statemachine for the SID then a new statemachine is created and the reference and SID stored in *statemachinedict*.

Once a statemachine instance has been found or created, a trigger for an event can be raised. Relevant events are:

TG_RESERVE: An external trigger to send a RESERVE message. This trigger is received from the local application.

RX_RESERVE: A RESERVE message received (from GIST).

TG_QUERY: An external trigger to send a QUERY message. This trigger is received from the local application.

RX_QUERY: A QUERY message received (from GIST).

RX_RESPONSE: A RESPONSE message received.

RX_NOTIFY: A NOTIFY message received.

TIMEOUT_RESPONSE: The Wait-Response timer has expired.

TIMEOUT_REFRESH: The Refresh timer has expired.

TIMEOUT_STATE_LIFETIME: The State lifetime timer has expired.

Of these events there are two (external) events that are triggered by an application, these are the TG_QUERY and TG_RESERVE events. The TG_QUERY event triggers the statemachine to send a QUERY message which can be used by a QNE to trigger a receiver initiated reservation request or to check the availability of resources. The TG_RESERVE event triggers the statemachine to send a RESERVE message which can be used to initiate a reservation.

The TIMEOUT events are internal events that are triggered due to the expiration of a particular timer. The TIMEOUT_RESPONSE event occurs when a RESERVE or QUERY message has been send and no RESPONSE message has been received within a predetermined interval. The TIMEOUT_REFRESH event is raised when the refresh timer expires. The TIMEOUT_STATE_LIFETIME event resembles the expiration of the reservations life time. It this timer expires and no refresh message has been received then the reservation state and persistent state are removed along with the current instance of the statemachine. In addition a tearing RESERVE message is sent.

Not all QNE's maintain all these timers. Table 5.1 shows the timers which are used by the different QNE's.

QNI	QNE	QNR
Refresh Timer Response Timer	Refresh Timer Response Timer StateLifetime timer	StateLifetime timer

Table 5.1: The timers at the QNI, QNE and QNR

The QNI only maintains the Refresh and Response timers. The Refresh timer is used to refresh the node's reservation state. Because the reservation state uses the soft state mechanism, it needs to be refreshed regularly. Whenever a QNI or QNE receives a trigger that the refresh timer has expired, a refreshing RESERVE message is sent peer-to-peer towards the QNR. The QNR on its turn send back a RESPONSE indicating whether the reservation state was refreshed successfully. Once a positive RESPONSE is received, the StateLifeTime time is restarted. If a negative RESPONSE is received then the reservation state is removed and a tearing RESERVE send to the QNR. Note that the QNR does not have a refresh timer, because it does not have a downstream neighbouring peer to which to send a RESERVE message.

The Response timer allows the QNI to check whether a neighbouring QNE has received a previously sent message. If the timer expires before the QNI has received a RESPONSE message then the QNI re-sends the message and restarts the timer. This process is repeated until a maximum retry is reached or when a RESPONSE message is received. The QNE also has a response timer which is used in the same manner. The QNR however does not have response timer. This is because it does not have a downstream peer and thus cannot send RESERVE or QUERY messages.

The StateLifetime timer is used to maintain the reservation state for a maximum period of time. If a reservation state has been active for more than the maximum lifetime then a trigger is released indicating

that the state lifetime has expired. In this case the reservation state is deleted and a tearing RESERVE sent to the QNR.

The events starting with RX correspond with messages received from GIST. The messages that can be received are RESERVE, QUERY, NOTIFY or RESPONSE messages. Note that there are other interactions between GIST and the NSLP layer which are not regarded in this statemachine. For example GIST can notify the NSLP layer of certain network information or the NSLP layer can instruct GIST to change its routing. A received message is processed based on the state of the statemachine and the role of the QNE. After the message has been processed a state transition is made and the running *qosServer.QOSServer* instance notified. If the statemachine is in an idle state when returning to the *qosServer.QOSServer* instance, it is deleted along with its reference in the *statemachinedict* lookup table.

Currently there are three states have been defined for the statemachine:

ST_IDLE: which is the idle state of the statemachine. In this state no reservation state have been installed. This is the initial state for new statemachines but it is also the state that triggers the *qosServer.QOSServer* instance to remove the statemachine.

ST_WAITRESP: in this state the statemachine is waiting for a response from a previous sent message. This is typically the case for receiver-initiated reservations or when resources are queried. In this state no reservation state is installed.

ST_INSTALLED: in this state a reservation state has been installed successfully.

In each of these states several events can be handled. An overview of the relevant events for every state is given in Table 5.2.

State	Event	Function
ST_IDLE	TG_QUERY	idle__tg_query
ST_IDLE	RX_QUERY	idle__rx_query
ST_IDLE	TG_RESERVE	idle__tg_reserve
ST_IDLE	RX_RESERVE	idle__rx_reserve
ST_IDLE	RX_RESPONSE	idle__rx_response
ST_WR	TG_QUERY	wr__tg_query
ST_WR	RX_QUERY	wr__rx_query
ST_WR	TG_RESERVE	wr__tg_reserve
ST_WR	RX_RESERVE	wr__rx_reserve
ST_WR	RX_RESPONSE	wr__rx_response
ST_WR	TIMEOUT_WAITRESP	wr__timeout_waitresp
ST_INST	TG_QUERY	inst__tg_query
ST_INST	RX_QUERY	inst__rx_query
ST_INST	RX_NOTIFY	inst__rx_notify
ST_INST	TG_RESERVE	inst__tg_reserve
ST_INST	RX_RESERVE	inst__rx_reserve
ST_INST	RX_RESPONSE	inst__rx_response
ST_INST	TIMEOUT_WAITRESP	inst__timeout_waitresp
ST_INST	TIMEOUT_REFRESH	inst__timeout_refresh
ST_INST	TIMEOUT_STATELIFETIME	inst__timeout_statelifetime

Table 5.2: All possible events for the different states in the statemachine.

For a complete overview and an detailed description of these functions the reader is referred to Fu et.al. [37]. In our design we will implement a subset of this statemachine. In this reduced statemachine only two state are used: the **IDLE** state and the **STATE_INSTALLED** state. In this statemachine the **STATE_WAIT_RESPONSE** is not included, because as stated earlier it is not used by our

QoSMs. A transition to the **STATE_WAIT_RESPONSE** state is made when a QNE wishes to set up a reservation using the receiver base reservation scenario or when the QNE wishes to send a query for particular resources. In both cases a QUERY message is send and the statemachine is set in the **STATE_WAIT_RESPONSE** state, indicating that the QNE is waiting for a RESPONSE message. The *ISCL-QOSM* and the *RMD-QoSM* used in our implementation do not support the querying of resources nor do they implement the receiver initiated reservation. Thus, it would useless to implement this functionality. Also because the described functionality is not supported, we will not implement the processing of QUERY messages. This type of message is only used in the receiver initiated reservation scenario or for the querying of available resources.

The IDLE state

When the statemachine is in the **IDLE** state it can receive a RESERVE or RESPONSE message from GIST or receive a trigger from a local application (see Figure 5.5). Receiving a RESPONSE message in the **IDLE** state is not very useful because no reservation state has been installed and thus the RESPONSE message does not seem have any meaning. If the Scoping flag (S-flag) is not set then the message need to forwarded upstream along the whole path. As a result only when the node in in the role of a QNE then must the message be forwarded further upstream. The statemachine will remain in the **IDLE** state after the message has been processed.

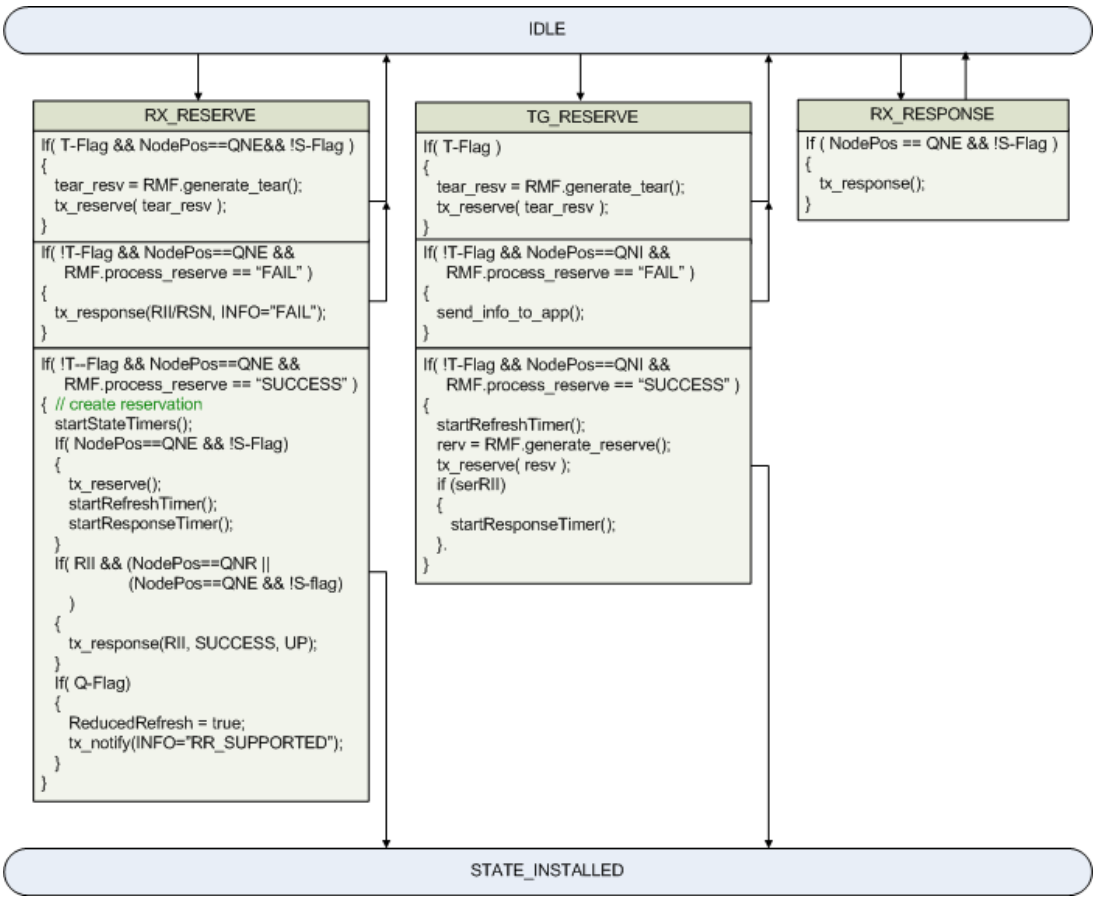


Figure 5.5: Transitions from the IDLE state.

Before we continue describing the transactions of this statemachine, the roles of the nodes in the RMD domain need to be explained. In the previous chapter we have defined some additional roles for the Ingress, Egress and interior nodes; the QNE-Ingress, the QNE-Egress and QNE-Core roles respectively. These

roles are quite similar to the three roles used in this statemachine, the QNI, QNE and QNR roles. The QNE-Ingress for example can be in one of these two roles depending on the signaling session associated to the statemachine. If the QNE-Ingress is processing an end-to-end signaling message then it plays the part of a QNE, whereas a signaling message received for an RMD signaling session is processed in the role of QNI. For the QNE-Egress this means that the node fulfills the role of an QNE in when processing an end-to-end signaling message and the role of QNR is assumed when processing RMD signaling messages. The QNE-Core always assumes the role of a QNE because it will only get to process the RMD signaling messages in which it plays the art of a QNE. End-to-end signaling messages are not processed by this node. The reason why the different roles QNE-Ingress, QNE-Egress and QNE-Core have been used was to avoid confusion.

In the **IDLE** state the statemachine may receive a RESERVE message from GIST. In this case the Tear flag (T-flag) is checked to see whether the RESERVE is a tearing RESERVE or not. If the T-flag is set then the RESERVE message should simply be forwarded because there is no reservation state that needs to be removed. Note that the message is only forwarded when the node is in a QNE role and the S-flag is not set. After the message has been forwarded a transition is made to the **IDLE** state.

If the T-flag has not been set then the supplied QSpec should be processed by the RMF. Here we have specified the specified RMF method which is called in our implementation. The QSpec is passed to the *RMF.process_reserve* method, which will return a 'SUCESS' message in case the QSpec has been processed correctly and the reservation state has been set up. A 'FAIL' message is returned when if the message could not be processed accordingly or when no reservation state has been set up (because the requested resources where not available). In this case a RESPONSE message is sent upstream ton indicate the failure and a transition is made to the **IDLE** state.

If the RESERVE message was processed correctly by the RMF a new updated QSpec is returned which is forwarded downstream in a newly created RESERVE message. This message is only forwarded downstream when the node is a QNE and the S-flag is not set. Notice how in our statemachine the *install_qos_state* has been removed. This is because in our design the reservation state is maintained by the RMF. After the reservation state has been installed the proper timers are started and the RESERVE message is processed further. The Q-flag is checked to see whether reduced refreshes should be used. If this flag is set then a NOTIFY message needs to be send upstream in order to confirm the usage of reduced refreshes. When using reduced refreshes no QSpec is added to a refreshing RESERVE message. This greatly reduces the size of the refresh message. Note however that the reduced refreshes can only be used when the NSLP operational state is installed at a node. If no operational state has been installed then a QSpec is needed to indicate which resources need to be refreshed.

In the received RESERVE message a RII object might have been included indicating that the QNI wishes to receive an explicit confirmation that the reservation state has been set up. In this case a RESPONSE message with a proper INFO-Spec is sent upstream. After the message has been processed completely a transition is made to the **STATE_INSTALLED** state.

The statemachine can also receive a trigger from a local application to initiate a reservation. Again here a T-flag is checked to indicate whether a reservation needs to be created or whether it needs to be torn down. If the T-flag is set then a tearing RESERVE message is sent downstream and a transition is mate to the **IDLE** state. If the T-flag is not set then the trigger is only processed if the node is in the QNI role. This is most likely to always be the case. Again here the RMF is requested to process the supplied QSpec and depending on its outcome a transition is made to either the **IDLE** state or the **STATE_INSTALLED** state. When the RMF returns a 'FAIL' message the local application will be notified of the failure and a transition is made to the **IDLE** state. If a 'SUCCESS' message is returned by the RMF then a RESERVE message is sent downstream, the relevant timers started and a transition made to the **STATE_INSTALLED** state. In our design the RMF is the one that will generate the RESERVE message, using the *RMF.generate_reserve* method. Strictly speaking this is not entirely correct, because the RMF should only generate a QSpec while the actual RESERVE message is constructed by the statemachine. The reason why this method is used in our implementation is because we did not want to change the common interface of the QoSM classes implemented in the **qosRMF**

module.

The STATE_INSTALLED state

In the **STATE_INSTALLED** state, reservation state has been set up for a particular signaling session. In this state several TIMEOUT events have been defined. In the beginning of this section the details of these TIMEOUT events have been explained and thus we will briefly discuss them here. In Figure 5.6 an overview of these events and their transactions is shown. The TIMEOUT_STATE_ LIFETIME event resembles the expiration of the reservations life time. When this timer expires, the reservation state is removed using the *RMF.remove_reservation* method. In addition all timers are stopped and a tearing RESERVE message is sent downstream (by all nodes other than the QNR). The TIMEOUT_RESPONSE event occurs when no RESPONSE message has been received within a predetermined interval, for a previously sent RESERVE or QUERY message. The RESERVE or QUERY message will be re-sent if the maximum number of retries has not yet been reached. Otherwise, the reservation state is removed using the *RMF.remove_reservation* method, the running timers stopped and a transition made to the **IDLE** state. The TIMEOUT_REFRESH event is raised when the refresh timer expires. Once the timer expires a refreshing RESERVE message is sent downstream and the Response and Refresh timers (re)started. Based on the parameters stored at this statemachine the RESERVE message is sent with or without a RII or QSpec object.

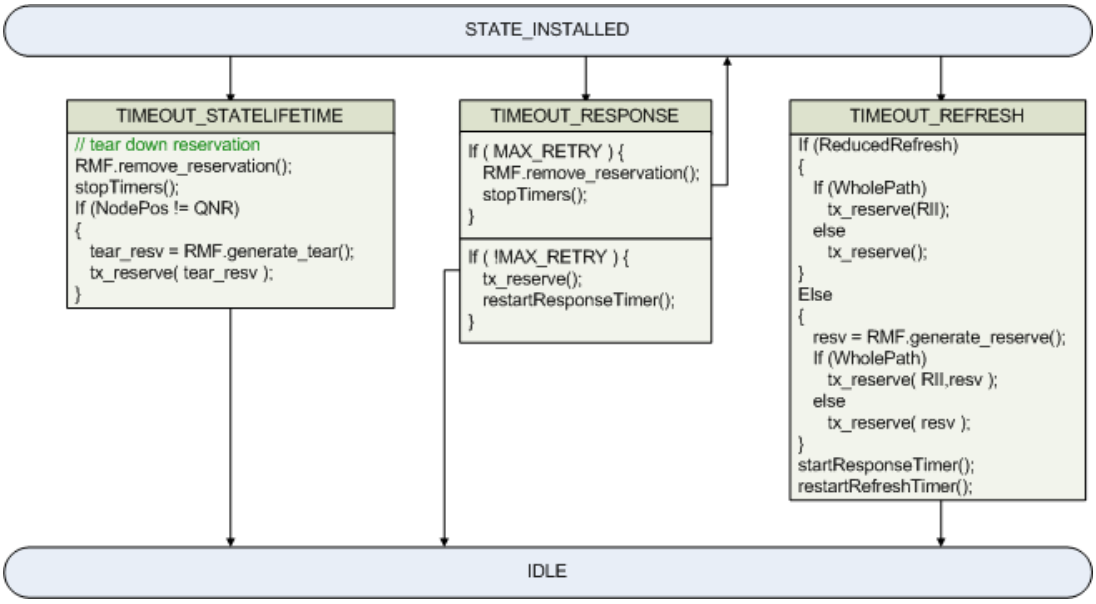


Figure 5.6: Transitions from the STATE_INSTALLED state.

The other events that have been defined for the **STATE_INSTALLED** state are shown in Figure 5.7. In this statediagram two events are shown that have not been defined in Fu et.al. [37]:

1. **TG_INCREASE:** An external trigger to increase the reservation. This trigger is received from the AMU.
2. **TG_DECREASE:** An external trigger to decrease the reservation, also received from the AMU.

These events are needed to add support for aggregation of flows to the statemachine. The original statemachine did not have any means of updating the reservation state. A reservation could only be set up, torn down or refreshed. In our statemachine, when a **TG_INCREASE** is received by the statemachine in the **STATE_INSTALLED** state, it will send a request for extra bandwidth to the

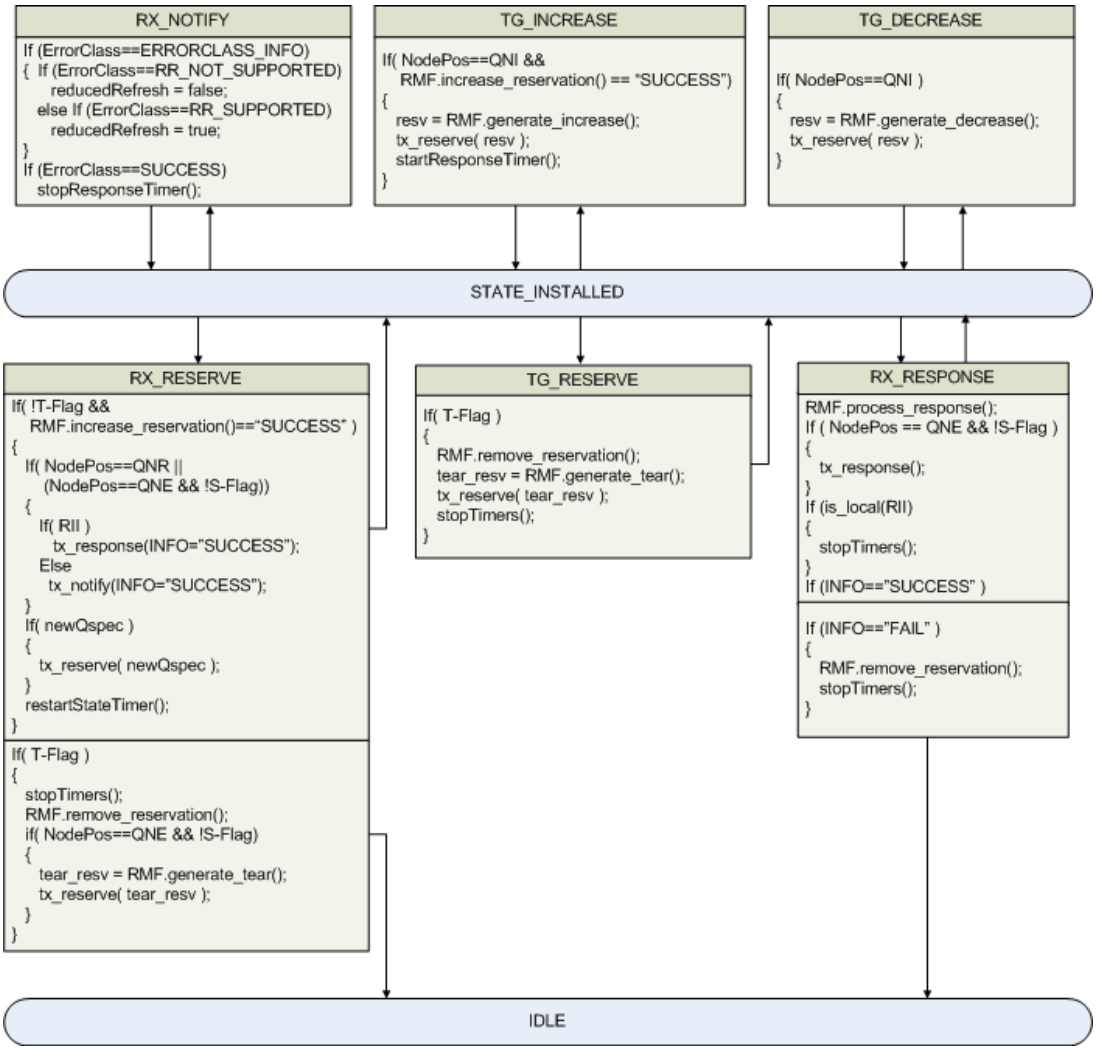


Figure 5.7: Transitions from the STATE_INSTALLED state.

RMF only when the node is in the role of a QNI. In other words, only the QNI is allowed to update the reservation state. First the RMF is queried for extra bandwidth using the *RMF.increase_reservation* method. This method is similar to the *RMF.increase_reservation* method, with the exception that no reservation state needs to be set up. The *RMF.increase_reservation* method checks with the AMU whether the requested bandwidth can be granted and if so a 'SUCCESS' message is returned. Note that at this point the resources will be claimed for the (aggregate) signaling session but they cannot be used yet. Only after a confirming RESPONSE message is received can this bandwidth be used. It is the task of the AMU to keep track of the amount of reserved bandwidth. This means that the admission control for new end-to-end flows, performed at the AMU, should be done using the previous reservation size until a confirming RESPONSE has been received.

If a 'SUCCESS' message has been received from the *RMF.increase_reservation* method, the statemachine will instruct the RMF to generate a RESERVE message, using the *RMF.generate_increase* method. The newly created RESERVE message is sent downstream and the Response timer is started. At the downstream nodes the RESERVE message is received and processed according to the **RX_RESERVE** transition (shown in Figure 5.7). Here the *RMF.process_reserve* method is called which will attempt to reserve the requested bandwidth. If the request is granted, a 'SUCCESS' message is returned. The *RMF.process_reserve* method may also construct a new QSpec, denoted by *newQspec* in the statediagram,

based on the received QSpec. If this is the case then this updated QSpec is sent further downstream until it reaches the QNR.

Note that the refresh procedure defined in the statemachine designed by Fu et.al. [37] is too simple for our prototype design. The problem here is that in Fu et.al. the assumption is made that a RESERVE message received in the **STATE_INSTALLED** state is either a refreshing RESERVE or a tearing RESERVE. In the first case a RESPONSE or NOTIFY message is sent upstream, while in the latter case the reservation state is torn down and the statemachine deleted. In both cases the QSpec is not sent to the RMF for processing. This is particularly troublesome for our design because our QNE-Ingress, QNE-Egress and QNE-Core nodes use the QSpec for the RMD-specific refresh and reserve procedures. This is why we have added the *RMF.process_reserve* method to the **RX_RESERVE** transition. Note that in our implementation the QNE-Core node will also use a statemachine for message processing but will not set up any operation state.

Eventually the increasing RESERVE message is received by the QNR. The QSpec is sent to the RMF for processing and based on the outcome a RESPONSE or NOTIFY message is sent back to the QNI. A RESPONSE message is send if the original QSpec carried an RII object and a NOTIFY message is send in case the RII was not included in the RESERVE message.

The **TG_DECREASE** indicates that the AMU wishes to decrease the reservation. This functionality is less complicated than increase because decreasing the reservation occurs instantaneously while increasing the reservation requires an explicit confirm. When this trigger is raised the statemachine will check whether it is in the role of a QNE. If this it the case, the *RMF.generate_decrease* method is called which will generate a RESERVE message containing a QSpec used for the decrease of the reservation state. Note that no tearing RESERVE is send because this would result in the NSLP operation state being torn down, which is not what we want. In the *RMF.generate_decrease* method a call is made tot the AMU which will perform the actual update on the reservation state.

Another trigger which can be received by the statemachine is the **TG_RESERVE**. This trigger, received from a local application, instructs the node to tear down the previously set up reservation state. Note that if the T-flag is not set, this trigger is simply ignored. If such a trigger is received then the RMF is instructed to remove the installed reservation state using the *RMF.remove_reservation* method. The timers are then stopped and a tearing RESERVE message is sent downstream. This tearing RESERVE message is created by calling the *RMF.genereate_tear* method.

The **RX_NOTIFY** event corresponds to a NOTIFY message being received from GIST. The INFO_Spec object included in this message is inspected and based on the supplied *ErrorClass* and *ErrorCode* parameters some internal parameters are set. In this design the NOTIFY message will be used to inform QNEs whether the reduced refresh procedure is supported or whether a RESERVE message was processed successfully.

To summarize, an overview of the functions that need to be implemented for our reduced statemachine is given in Table 5.3.

5.3.2 Resource Management Function

In the previous section we have seen how the RMF is used in the NSLP statemachine. Here we will describe the details of the method used by the statemachine.

For each QoSM used in a node, an instance of the relevant class in the *qosRMF* module is maintained. The current implementation is set up in such a way that a QoSM class in the *qosRMF* module is associated to a particular network interface. In the *qosOptions* class one can specify which QoSM to use on which network interface. For example our QNE-Ingress node has its external interface (relative to the RMD domain) configured to use the *ISCL-QoSM*, while the internal interface is configured to use the *RMD-QoSM*. The QoSM class implementing a particular QoSM specification must use the common interface defined by the **qosRMF** module. This way the *qosServer.QOSServer* can use the same methods for

State	Event	Function
ST_IDLE	TG_RESERVE	idle__tg_reserve
ST_IDLE	RX_RESERVE	idle__rx_reserve
ST_IDLE	RX_RESPONSE	idle__rx_response
ST_INST	TG_DECREASE	inst__tg_decrease
ST_INST	TG_INCREASE	inst__tg_increase
ST_INST	RX_NOTIFY	inst__rx_notify
ST_INST	TG_RESERVE	inst__tg_reserve
ST_INST	RX_RESERVE	inst__rx_reserve
ST_INST	RX_RESPONSE	inst__rx_response
ST_INST	TIMEOUT_WAITRESP	inst__timeout_waitresp
ST_INST	TIMEOUT_REFRESH	inst__timeout_refresh
ST_INST	TIMEOUT_STATELIFETIME	inst__timeout_statelifetime

Table 5.3: All possible events for our statemachine.

different QoSM classes. The methods that must be implemented are:

- **generate_reserve**: This method instructs the RMF to construct an initiating reserve message with a QSpec relevant to the QoSM used. This method is generally called as a result of an incoming application API call.
- **process_reserve**: This method is called when a RESERVE message is received from GIST. Here the QSpec is processed according to rules defined in the QoSM specification. Depending on the values of the QSpec, a reservation can be set up, torn down or refreshed. If the QSpec is processed correctly an updated QSpec based on the received one is returned. This QSpec will be propagated further downstream. In case the QSpec was not processed correctly, an exception is thrown, resulting in the *qosServer.QOSServer* sending a RESPONSE message upstream.

  At a QNR this method behaves slightly different. In this case a RESPONSE message is created and send back towards the QNI confirming the reservation setup or failure.
- **process_response**: This method is called by the *qosServer.QOSServer* upon receiving a RESPONSE message from GIST containing a QSpec. After having inspected the QSpec the RMF can conclude whether the reservation was successful or not. In the case that the reservation has failed, a tearing RESERVE is sent downstream for the removal of states previously installed at some nodes.
- **generate_refresh**: This method instructs the RMF to construct a refreshing RESERVE message for a particular session. This method is called by the *qosState.PersistentState* class due to the expiration of the refresh timer maintained in this state.
- **generate_tear**: This method instructs the RMF to generate a tearing RESERVE message for a particular session. This method is called whenever an application wishes to tear down a reservation or due to error handling.
- **remove_reservation**: This method instructs the RMF to remove the reservation state installed at this node for a particular session. This method is called when a tearing RESERVE message is received by the *qosServer* or an application has instructed the *qosServer.QOSServer* to remove a reservation explicitly or whenever the state lifetime timer maintained at the *qosState.PersistentState* has expired.

The methods defined above do not cover all the interactions between the statemachine and the RMF. For example the QUERY and NOTIFY messages can also contain a QSpec object which will need to be processed by the RMF according to the rules of a particular QoSM. In this case methods like **process_query** and **process_notify** would have to be implemented. For the scope of this assignment

we will not implement these methods, because they are not used by our QoSM. The *ISCL-QoSM* only uses the RESERVE message to set up or tear down a reservation, and a RESPONSE message to notify the initiating node of a successful or unsuccessful reservation setup. The *RMD-QoSM* also uses RESERVE and RESPONSE messages for all its actions (see Chapter 4). In the signaling statemachine we see that the NOTIFY message is used to inform other nodes whether the reduced refresh procedure is supported for example. These NOTIFY messages however do not carry a QSpec and thus do not need to be processed by the RMF.

Aside from the methods defined in the common interface the following methods will be implemented in our *RMF.RMD* class:

- **increase_reservation**: This method instructs the RMF to increase the amount of reserved bandwidth for a particular session. After the QSpec has been processed, a call is made to the AMU which is responsible for the maintenance of the reservation state. If the extra bandwidth for the increase can be granted at locally, a new RESERVE message is constructed which will be sent downstream to request the extra bandwidth at the downstream nodes.
- **decrease_reservation**: This method instructs the RMF to decrease the amount of reserved bandwidth for a particular session. This maintenance of the aggregate reservation state is done by the AMU, so after the QSpec has been processed the AMU is instructed to update the reservation state accordingly. The method returns a RESERVE message containing a QSpec for the decrease of the aggregate reservation state. Note that no tearing RESERVE message is sent because the NSLP operational state needs to be kept intact.
- **generate_increase**: This method instructs the RMF to create a RESERVE message for the increase of the (aggregate). Note that this method is specific for our aggregation based RMD-QoSM.
- **generate_decrease**: This method instructs the RMF to create a RESERVE message for the decrease of the (aggregate). Note that this method is specific for our aggregation based RMD-QoSM.

The methods defined above result into a call to the AMU. As stated before, the *qosRMF.RMD* class is not allowed to manipulate the (aggregate) reservation state directly. For that the AMU should be used. In the next section the details of the AMU are discussed.

5.3.3 Aggregate management unit

The *Aggregate Management Unit* (AMU) is responsible for controlling the aggregate reservation. It needs to be able to setup an aggregate reservation, increase and decrease the aggregate reservation size. Last but not least it should also be able to tear down the aggregate reservation. Aside from controlling the aggregate reservation is can also be used for admission control for end-to-end flows joining the aggregate reservation.

As shown in Figure 5.3, the AMU is located in the NSLP layer which is slightly different from the approach taken in the '*NSLP for Quality-of-Service Signaling*' draft [18]. Here an 'aggregate management application' is positioned in the application layer. The reason why the AMU in our design is located in the NSLP layer because of its close relationship with the aggregation based RMD-QoSM. If the AMU is modeled as an application is would required the implementation of additional communication means. A socket connection would have to be set up between the RMF and the AMU and the messages would have to be encoded and decoded at both ends. Also the RMF would have to implement methods that can verify the presence of the AMU. In case no AMU application would be present, proper error handling would have to be implemented. All these issues makes implementing the AMU as an application unnecessarily complex.

The AMU is mainly used by the QNE-Ingress. It is here that end-to-end flows are aggregated and the increase or decrease of the aggregate reservation initiated. Before the end-to-end flows can be aggregate

an aggregate reservation needs to be present. The AMU is responsible for the setup of this reservation. In our implementation we have chosen to set up the aggregate reservation immediately after the *qosServer.QOSServer* has been started. As stated in Section 4.3.1 the initiation of this reservation can be configured in several other ways, of which the automatic set up has been chosen. The setup of the aggregate reservation is done by raising the **TG_RESERVE** at the *qosServer.QOSServer* instance. The server will then create a statemachine and raise the **TG_RESERVE** instructing the statemachine to trigger the set up of the aggregate reservation.

Once this reservation has been set up it will be increased or decreased by our AMU using an specified aggregate update policy. In our implementation only one aggregate update policy has been implemented, the one specified in our proposed solution (see Section 2.2). The aggregate update policy is implemented in the *qosAMU.UpdatePolicy* class.

In our *qosAMU.AggregateManagementUnit* class the following methods have been implemented:

create_aggr: This method instructs the AMU to setup an aggregate reservation state. The setup of the aggregate reservation is done according to the procedure defined in Section 4.3.1.

remove_aggr: This method instructs the AMU to remove an aggregate reservation state.

incr_aggr: This method is used to notify the AMU of a successful increase of the aggregate reservation. In this case the aggregate reservation state is updated by increasing the reserved bandwidth for the aggregate reservation with the specified amount of bandwidth. This is done according to the procedure defined in Section 4.3.3.

decr_aggr: This method is used to notify the AMU of a successful decrease of the aggregate reservation. The aggregate reservation state is updated and the reserved bandwidth for the aggregate is decrease with the specified amount of bandwidth. This is done according to the procedure defined in Section 4.3.4.

add_bw: This method is used to notify the AMU of an increase in the aggregate traffic demand (r_a). In other words an end-to-end flow wishes to join the aggregate. This method will check whether the aggregate reservation has enough bandwidth to grant the request of the end-to-end flow. The end-to-end flow can only be added if $r_a + x_{bw} \leq C_a$, where x_{bw} is the bandwidth requested by the end-to-end flow and C_a the bandwidth that has been reserved for the aggregate. If this is the case then this method returns a positive acknowledgment. If the request could not be granted a negative acknowledgment is returned.

rem_bw: This method is used to notify the AMU of a decrease in the aggregate traffic demand (r_a). In other words an end-to-end flow has left the aggregate.

In the *qosAMU.AggregateManagementUnit* class the aggregate traffic rate (r_a) is maintained. The RMF uses the **add_bw** and **rem_bw** methods to notify the *qosAMU.AggregateManagementUnit* class of the changes in the aggregate traffic demand. An end-to-end flow arriving at the QNE-Ingress it will have to pass the admission control check before being admitted to the aggregate. This is done in the **add_bw** method. If the flow has been admitted then r_a is updated as follows: $r_a = r_a + x_{bw}$ where x_{bw} is the bandwidth request of the end-to-end flow. The remove the **rem_bw** method is used to remove the bandwidth of end-to-end flow from the aggregate traffic demand. Here the parameter r_a is updated as follows: $r_a = r_a - x_{bw}$. Once r_a has been updated, the *qosAMU.AggregateManagementUnit* checks whether the new value for r_a still lays within the thresholds. If the new value for r_a crosses the threshold boundaries, an increase or decrease of the aggregate reservation is triggered.

The updating of the aggregate reservation size is done using the threshold based update policy discussed in Section 2.2. The aggregate reservation size is updated whenever the aggregate traffic crosses the upper or lower threshold. Internally the *qosAMU.AggregateManagementUnit* class has two thresholds which are updated according to the update policy specified in 2.2.1. This update policy is implemented in the *qosAMU.UpdatePolicy*. Periodically the *qosAMU.AggregateManagementUnit* can ask the

AMU.UpdatePolicy class for a new set of thresholds using the **calc_thresholds** method implemented b the update policy. Here the thresholds are computed according to Equations (2.20) and (2.19) for the upper and lower threshold respectively. If a AMU needs to increase the aggregate reservation, it queries the *AMU.UpdatePolicy* class for the new aggregate reservation size (C_a) using the **calc_ca** method. This method will compute the new value for C_a according to Equation (2.18). After the new aggregate reservation size has been computed, the **TG_INCREASE** trigger is raised to notify the QNE-Ingress node of the increase and the amount of bandwidth that needs to be reserved. This trigger is send to the *qosServer.QOSServer* which will handle the further processing of the trigger.

Chapter 6

Experimental Evaluation

In the previous chapter we have discussed how the prototype can be extended in order to support aggregation based RMD. Our prototype implementation has to be tested for bugs and compliance to the RMD QoSM specification discussed in Chapter 4. The experiments that serve the purpose of testing the implementation for bugs and compliance are categorized as *Functional Experiments*. Aside from the functional experiments, *Performance Experiments* have been conducted in order to measure how well our aggregate update policy works. In this chapter the *Functional* - and *Performance Experiments*, including their results are discussed.

6.1 Test environment

For the experiments on our implementation we have set up a test environment to mimic a real-life scenario on a small scale. In our test environment we have set up a network with five routers, each having the role of either QNI, QNE-Ingress, QNE-Interior, QNE-Egress or QNR. Due to the limited amount of resources our test environment has been recreated using several virtual machines. Virtual machines allows us to run multiple instances of the Linux operating system on the same machine. The topology of the network used for testing is shown in Figure 6.1.

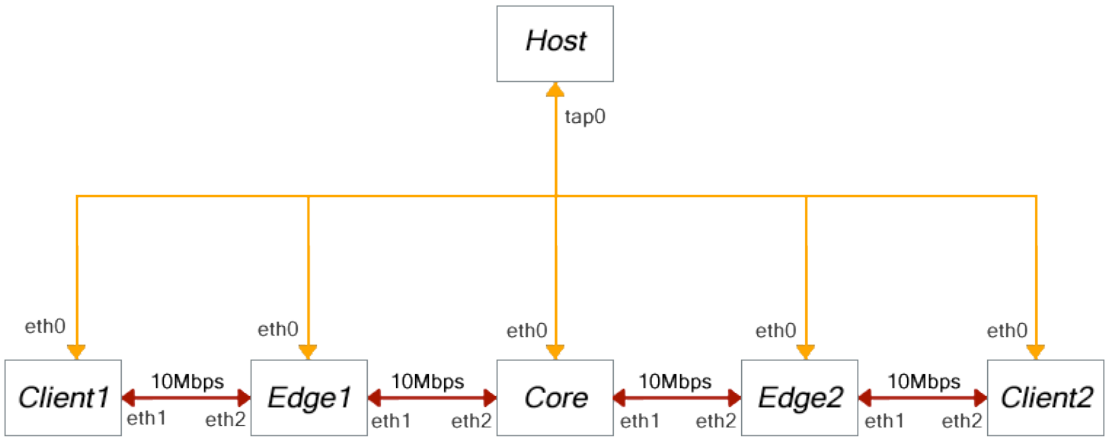


Figure 6.1: The test environment.

Here we have five virtual machines connected to each other, each representing a different role for the QNE. The virtual machines are User-Mode Linux (UML) clients running on our host Linux server. The host server runs Gentoo Linux version 3.4.5 on a Intel Pentium 800 MHz with 384 Mb of memory. The Linux virtual machines are configured to run the same Linux kernel as the host server (version 2.6.14-gentoo-r3) with 64 Mb of (virtual) memory. The UMLs run as separate processes in our host system. Every virtual

machine has one or more virtual network interfaces which connects it to either the host system (via the *eth0* network interface) or a neighboring virtual machine. The nodes in the RMD domain, *Edge1*, *Core* and *Edge2* have three network interfaces. One connects the node to the *Host* Linux server and the others connect the node to its neighbors. The QNE-Ingress is located at router *Edge1*. It is connected to the QNI, which is located at the *Client1* router, and with the QNE-Interior, which is located at the *Core* router. The QNE-Egress is located at the *Edge2* router which is connected to the QNE-Interior and the QNR located at the *Client2* router.

For our functional experiments we will instruct the QNE-Ingress router to setup, increase, decrease and refresh an aggregate reservation within the RMD domain. Here we will test whether the aggregate reservations are set up and maintained according to the specification defined in Chapter 4. For the performance experiments we will analyze how the aggregate reservation is updated for end-to-end flows joining and leaving the aggregate. This is achieved by instructing the QNI to set up and tear down end-to-end flows between the QNI and QNR according to some predefined traffic model. The setup of these experiments is discussed in more detail in the next sections.

6.2 Functional experiments

To test whether our implementation complies with the RMD-QoSM specification discussed in Chapter 4 we will conduct the following functional experiments:

1. *The successful setup of an aggregate reservation.*
2. *The unsuccessful setup of an aggregate reservation.*
3. *The successful increase of an aggregate reservation.*
4. *The unsuccessful increase of an aggregate reservation.*
5. *The successful decrease of an aggregate reservation.*
6. *The successful refresh of an aggregate reservation.*

In addition to the scenarios listed above some additional testing is done for some specific cases which are relevant for our performance experiments. These additional tests are needed so that the performance experiments can be performed reliably.

To observe the behavior of our prototype implementation, we rely on the information stored in log files. At every UML a log file is maintained which stores information regarding the signaling messages sent or received, the reservation state which has been created or torn down, and the (NSLP and GIST) statemachine interactions. Whenever the GIST server is started the log file is cleared and information is stored for as long as the server is active. For example, if an application wishes to set up a signaling session, information regarding the IP address and port numbers of the sender and receiver, along with the NSLP operational state information (such as the RSN, RII and SID) and reservation state, is recorded in the log file. If a GIST statemachine is created or an existing statemachine manipulated, this information is stored to the log file. Also whenever a signaling message is sent or received, the complete message is written to the log file.

6.2.1 Successful setup of an aggregate reservation

Our first functional experiment checks whether the aggregate reservation has been set up correctly (see Section 4.3.1 for the details). First we instruct our QNE-Ingress to initiate the setup of the aggregate reservation using the *qosConsole* application. This is basically the manual setup of the aggregate reservation. At every node logging is performed in which state transitions, sent and received messages are

stored in a file. From the log file at the QNE-Ingress we see how an intra-domain RESERVE message is sent towards the QNE-Egress. After some time a RESPONSE message is received by the QNE-Ingress indicating that the reservation was set up successfully.

Aside from the QoS signaling messages the state transitions of the GIST state machine are shown. From the GIST interaction we can see that the RESERVE message sent by the QNE-Ingress is done in GIST-Query mode. Here a GIST-Query message is sent towards the QNE-Egress with the RESERVE message as its payload. After the GIST-Query has been sent a *Query State Machine*, in state *Awaiting Response*, is created at the QNE-Ingress. After some time the GIST-Response message is received and the state machine is set to the state *Established*. The GIST then sends a GIST-Confirm message in order to confirm the setup of the GIST state.

The log file of the QNE-Core node (at the Core UML) shows the signaling messages that have been received by the QNE-Core node. Here we see how a GIST-Query message, with an intra-domain RESERVE message as payload, is received. The RESERVE message is passed on to the NSLP layer where it is processed. Here the RMD reservation state is setup and an updated RESERVE message forwarded in a GIST-Query message. The next GIST message received by the QNE-Core node is a GIST-Reponse message which is simply forwarded to the QNE-Ingress. This is because the QNE-Core node does not store any GIST state. This message is followed by the GIST-Confirm message send by the QNE-Ingress. Finally the intra-domain RESPONSE message is received and processed by the QNE-Core and forwarded to the QNE-Ingress.

At the QNE-Egress the intra-domain RESERVE message is received over a GIST-Query message. The GIST-Query triggers the creation of a *Responder State Machine* which is set to state *Awaiting Confirm*. The RESERVE message is passed to the upper NSLP layer and processed by the QNE-Egress. Because the reservation was setup successfully a RESPONSE with an INFO_Spec object set to *Error Code: Reservation Successful* and *E-Class: Success* is send to the QNE-Ingress. The RESPONSE message also includes a PDR object set to code a *PDR Reservation Report*. Finally the QNE-Egress receives the GIST-Confirm message sent by the QNE-Ingress to complete the GIST state setup.

In the log files at the QNE-Ingress, QNE-Core and QNE-Egress, information regarding the reservation state is stored. Here we can see how an aggregate reservation has been set up. The correct messages are send in the order described in Section 4.3.1 and processed accordingly. From these log files we can also conclude that the transfer of these signaling message is done according to the procedure described in Section 4.2.1 (*Transport of signaling messages*).

6.2.2 Unsuccessful setup of an aggregate reservation

In order to test the unsuccessful setup of an aggregate reservation the QNE-Core is modified such that it will deny any reservation request (see Section 4.3.1 for the details). At the QNE-Ingress the setup of the aggregate reservation is initiated using our *qosConsole* application. A RESERVE message is sent from the QNE-Ingress to the QNE-Core. At the QNE-Core the request is denied and from the logs we see how a RESERVE message with the M-flag in the PACKET_CLASSIFIER object set is sent to the QNE-Egress node.

At the QNE-Egress the failure is recognized and a RESPONSE with the an INFO_Spec object set to: *Error Code: Reservation failure* and *E-Class: Transient Failure* is sent back to the QNE-Ingress. This RESPONSE contains a PDR object set to code a *PDR Reservation Report*. In the PDR object the *Max Admitted Hops* flags is set to the value of the *Admitted Hops* parameter in the received PHR object. Because the reservation setup has failed at the first hop this value is zero.

The QNE-Ingress node receives the RESPONSE message and tears down the previously set up reservation state. Because the *Max Admitted Hops* parameter in the PDR object is set to zero no reservation states needed to be released and thus no tearing RESERVE message is send downstream. In order to test the *RMD partial release procedure* (see Section 4.3.1) we have configured our QNE-Egress to deny the request, instead of the QNE-Core. In this case the QNE-Egress sends a RESPONSE message with an

INFO_Spec set to code the reservation failure and a PDR with the *Max Admitted Hops* set to 1, because reservation state has been set up at the QNE-Core node.

At the QNE-Ingress the RESPONSE message is received and processed. The reservation state is torn down and a new RESERVE message is created in order to release resources that have been reserved at the QNE-Core. This RESERVE message carries a RMD-QSpec with the PHR parameter set to *PHR Release Request* and the M-flag set. In addition to the PHR a PDR parameter is included in the QSpec.

At the QNE-Core node the RESERVE message is received and the reservation state release. An updated RESERVE message is sent to the QNE-Egress. In the new RESERVE the I-flag is set indicating that the release procedure has been terminated and that the following nodes should not tear down their reservation states.

From log files stored at the QNE-Ingress, QNE-Core and QNE-Egress, we can conclude that in our prototype implementation the *RMD partial release procedure* has correctly implemented. It can be seen that the notification of failure to reserve resources is correctly processed by the QNE-Egress node and the clean up of previously installed reservation state is done correctly at the QNE-Ingress and QNE-Core nodes. The log file at the QNE_Ingress shows that reservation state has been created for the aggregate and after the RESERVE message is sent, a RESPONSE message is received. This RESPONSE message contains an INFO_Spec with the failure and as a result the reservation state is deleted and a RESERVE message is sent. In the case when reservation state was set up only at the QNE-Ingress, no RESERVE message is sent to release bandwidth at the other nodes. This is in accordance with the NSIS specification [18].

6.2.3 Successful increase of an aggregate reservation

This scenario for increasing the aggregate reservation is quite similar to the *Successful setup of an aggregate reservation* scenario discussed above so we will not explain it in detail. From the log files stored at the QNE-Ingress, QNE-Core and QNE-Egress, we can conclude the following:

- The RESERVE message sent by the QNE-Ingress, used to instruct the node to increase the aggregate reservation, is sent in GIST-Query mode. Note that when the aggregate reservation has been setup so has the GIST routing state. This means that the QNE-Ingress and QNE-Egress router can send each other messages directly using the GIST-Datagram mode. When increasing the aggregate reservation the RESERVE message must be sent to and processed by the QNE-Core node. Thus in this case the RESERVE message is sent in GIST-Query mode.
- After the RESERVE message has been processed the reservation state is updated correctly. At the edge nodes the SID is used to lookup the statemachine for the signaling session which updates the amount of reserved resources accordingly. At the QNE-Core node this is slightly different because only per traffic class reservation state is stored. Thus the traffic class for the RESERVE message has to be extracted in order to find and update the reservation state accordingly. From our logs we can conclude that this procedure works correctly.

Our experiment shows that the aggregate increase procedure, discussed in Section 4.3.3, has been implemented correctly.

6.2.4 Unsuccessful increase of an aggregate reservation

In order to fully test this scenario we have set a maximum amount of resources which can be reserved at the QNE-Egress node. When this maximum is reached the QNE-Egress node will reject the request for extra bandwidth and notify the QNE-Ingress node of the failure in order to set the RMD partial release procedure in motion. At the QNE-Ingress we have set up an aggregate reservation which will be

increased with some extra bandwidth. The maximum amount of resources which can be reserved at the QNE-Egress is set such that it will reject this extra bandwidth. The RESERVE message for the extra bandwidth is send by the QNE-Ingress to the QNE-Core, which will increase the aggregate reservation and forward the message to the QNE-Egress. At the QNE-Egress the request for extra bandwidth is denied and a RESPONSE with an INFO_Spec set to code the reservation failure is sent back to the QNE-Ingress.

At the QNE-Ingress the aggregate reservation state is not increased due to the reservation failure. Additionally the QNE-Ingress inspects the *Max Admitted Hops* parameter to check whether the reservation state other nodes needs to be released. In this case the reservation state at the QNE-Core node needs to be decreased and thus the QNE-Ingress node sets the RMD partial release procedure in motion.

This experiment shows that the unsuccessful aggregate increase procedure, discussed in Section 4.3.3, has been implemented correctly. The notification of the reservation failure is done correctly and the RMD partial release procedure is initiated accordingly. In the RMD partial release procedure the release of the extra bandwidth at the QNE-Ingress and the QNE-Core nodes is done correctly. Note that in the scenario where end-to-end reservations are released, receiving a tearing RESERVE message would lead to the tear down of the QoS NSLP operational and reservation states. However in the case of aggregation the QoS NSLP operation state at the edge nodes should remain intact and the aggregate reservation at the interior nodes decreased with the extra bandwidth that has been reserved previously. This is done correctly in our prototype implementation.

6.2.5 Successful decrease of an aggregate reservation

The decrease of the aggregate reservation is quite similar to the RMD partial release procedure and thus we will not discuss it in great detail. After an aggregate reservation has been set up the QNE-Ingress router is instructed to decrease the aggregate reservation. First a RESERVE message is constructed at the QNE-Ingress which is sent towards the QNE-Egress node. The signaling message is sent in GIST-Query mode because it needs to be processed by the QNE-Core node. The RESERVE message includes a RMD-QSpec containing a PHR object set to code "*PHR Release Request*". In the PHR object the M-flag is also set to 1. This RESERVE message is processed properly at every node in the RMD domain and the reservation state is updated accordingly.

Our experiment shows that the aggregate decrease procedure, discussed in Section 4.3.4, has been implemented correctly.

6.2.6 Successful refresh of an aggregate reservation

Again, the refreshing of the aggregate reservation state is similar to the setup and increase of the aggregate reservation. From our test we can conclude that the refreshing of the aggregate reservation works according to the specification discussed in Section 4.3.5.

In order to test whether the refreshing procedure is implemented correctly we have setup an aggregate reservation and waited about 10 minutes before checking the reservation state. Logging information show that indeed (refreshing) RESERVE messages have been sent between the QNE-Ingress, QNE-Core and QNE-Egress nodes. Also, the size of the aggregate reservation has remained the same at all the nodes in the RMD domain.

After we have tested whether the reservation state is indeed refreshed we shut down the QNE-Ingress node. Shutting down the QNE-Ingress node will not cause the aggregate reservation state at the QNE-Core and QNE-Egress nodes to be refreshed. Indeed after the refresh period has passed (in our tests set to 30 seconds) the reservation state is torn down at the QNE-Core and QNE-Egress nodes. The QNE-Egress node will also send a RESPONSE message to the QNE-Ingress notifying it of the tear down of the aggregate reservation state.

Note that due to limited resources we were not able to test whether the reservation state at the QNE-Core is refreshed/updated correctly when using multiple Ingress-Egress pairs. In the case of multiple Ingress-Egress pairs the reservation state at the core nodes consists of bandwidth reserved for all the Ingress-Egress pairs that traverse the core node. Thus when an Ingress-Egress pair refreshes the reservation state at this core node only part of the reservation state is refreshed. In the RMD QoSM specification [1] a refresh procedure has been defined which correctly refreshes the reservation state at the core nodes. Here the refreshed resources are set to zero at the beginning of the refresh period and and incremented with every refresh message. At the end of the refresh period the reserved resources are set to the amount of refreshed resources for that refresh period. This process is repeated for every refresh period. Due to our limited resources at our disposal, this refresh procedure could not be tested. This however has no impact on our performance tests.

6.2.7 Additional testing

In addition to the functional tests above we have also tested whether our QNE-Ingress nodes rejects end-to-end flow request when the aggregate reservation does not have sufficient resources to grant the request. This test is particularly important because in the performance experiments, discussed in the next section, we assume that this admission control check works. By checking whether the flows are truly rejected in the case of insufficient bandwidth the correct measurement of our blocking probability is ensured.

6.3 Performance experiments

The performance experiments focus on how our aggregate update policy performs according to the constraints defined in our problem definition. Our aggregate update policy has two parameters that can be set which will influence the performance of the update policy. These parameters are:

1. the desired pipe size modication period or desired inter-update period T
2. the maximum blocking probability ϵ.

The desired inter-update period T tells our aggregate update policy how often we want our aggregate reservation to be updated. Setting this parameter to a high value will instruct the aggregate policy to reserve more bandwidth so that the aggregate reservation is updated less frequently. However this will also result in a lower fraction of the reserved bandwidth to be utilized by the flows in the aggregate. If we set T to a small value then the aggregate reservation is updated more frequently but its utilization will also be higher.

The maximum blocking probability ϵ allows our update policy to reject a certain maximum number of flows when the aggregate reservation needs to be increased. Although this parameter has no major impact on the performance it is of importance to our update policy. Setting ϵ to a high value allows our update policy to be more relaxed in rejecting flows. As a result our update policy will wait longer before initiating an increase of the aggregate reservation. Because the aggregate update policy is allowed to reject more flows it will postpone the increasing of the aggregate reservation. If ϵ is set to a low value then our aggregate update policy will not take much risk and increase the aggregate at an earlier stage.

The impact of these parameters on the performance of our aggregate update policy is what we want to examine. In order to measure the performance for different values of T and ϵ we need to define (measurable) performance indicators which express how well our aggregate update policy works. The performance indicators for our aggregate update policy have been defined in Chapter 1 (Section 1.4.2). In our problem statement we have defined the following constraints: the signaling load $S(t)$, the normalized bandwidth inefficiency $BI'(t)$ and the blocking probability P_{bl}. These constraints will be used to measure

the performance of our aggregate policy. Our aggregate update policy should be configured such that the these constraints are as small as possible.

In this section the setup of the performance experiments is discussed. It is subdivided as follows:

- **Goals**: here we will discuss the goals of our performance experiments.
- **Measurements**: here we will discuss what we want to measure in our experiments and how we these measurements are conducted.
- **Assumptions**: here the assumption we have made for our parameters and our performance indicators are discussed.
- **Scenario**: here the test environment for our experiments is discussed.
- **Traffic model**: here the traffic model used in our experiments is discussed.
- **Results**: here the results of our performance experiments are discussed.

6.3.1 Goals

As stated in the previous section we want to examine the effects of T and ϵ have on the performance of our aggregate update policy. The performance of our aggregate update policy is expressed by the following constraints:

- the signaling load $S(t)$
- the (normalized) bandwidth inefficiency $BI'(t)$.
- the blocking probability P_{bl}

For different values of T and ϵ we need to measure $S(t)$, $BI'(t)$ and P_{bl}. Recall that we need to set T and ϵ such that these constraints are as small as possible. Because these constraints complement each other, an optimal solution need to be found. This can be achieved using the cost function defined in Section 1.4.1:

$$c = \kappa_1 \times S(t) + \kappa_2 \times BI'(t) + \kappa_3 \times P_{bl} \tag{6.1}$$

where κ_1, κ_2 and κ_3 are constants used to normalize the signaling load $S(t)$, the normalized bandwidth inefficiency $BI'(t)$ and the blocking probability P_{bl}. The constants κ_1, κ_2 and κ_3 are weights that are assigned to the different constrains indicating their importance. As an additional requirement for c we will define that the sum of the weights κ_1, κ_2 and κ_3 must be 100%.

The main goal of our performance experiments is to find some optimum value for T and ϵ such that the cost function c is minimal. Note that we can only calculate the minimal cost of our update policy for a given set of values for κ_1, κ_2 and κ_3. For the calculation of the cost in our performance experiments we will set the values of κ_1, κ_2 and κ_3 to be equally important. In other words $\kappa_1 = 33.3\%$, $\kappa_2 = 33.3\%$ and $\kappa_3 = 33.3\%$. The effects of using different values for κ_1, κ_2 and κ_3 will not be examined.

6.3.2 Measurements

If we want to compute the optimum value for c we need to measure the signaling load $S(t)$, the normalized bandwidth inefficiency $BI'(t)$ and the blocking probability P_{bl}. This is done as follows:

- The signaling load $S(t)$ is defined as follows:

$$S(t) = \frac{n_u(t)}{n_a(t) + n_d(t)} \tag{6.2}$$

where $n_u(t)$ is the number of update messages sent up to a certain time t, $n_a(t)$ is the number of reservation requests received up to time t, and $n_d(t)$ the number of flows that have left the aggregate up to time t. Thus, for the signaling load $S(t)$ we have to measure:

(i) $n_u(t)$ which is the number of update messages sent up to a certain time t,

(ii) $n_a(t)$ which is the number of reservation requests received up to time t, and

(iii) $n_d(t)$ which is the number of flows that have left the aggregate up to time t.

Note that in this definition of the signaling load $S(t)$ refresh messages are not taken into account. If we include the refresh messages in our definition of the signaling load then the performance gain compared to an end-to-end signaling scenario without aggregation would be much greater. This is because in an end-to-end signaling scenario every reservation needs to be refreshed individually. When using aggregation only the aggregate reservation needs to be refreshed and not the end-to-end flows in the aggregate. However in our experiments we are not interested in the number of (aggregate) refresh messages because they remain the same for different values of T and ϵ. The refreshing of the aggregate reservation is an RMD specific procedure over which our aggregate update policy does not have any influence.

If we want to measure $n_a(t)$ then we need to count the number of (end-to-end) RESERVE messages that cause the setup of an end-to-end reservation in time interval [0,t]. For the measurement of $n_d(t)$ we need to measure the number of end-to-end tearing RESERVE messages that cause the tear down of an end-to-end reservation in time interval [0,t]. The RESERVE messages that are used to refresh the end-to-end reservation states are thus ignored. The parameter $n_u(t)$ is measured by counting the amount of intra-domain RESERVE messages over time that cause the increase or the decrease of the aggregate reservation during time interval [0,t].

- The normalized bandwidth inefficiency $BI'(t)$ can be computed using the following equation:

$$BI'(t) = \frac{BI(t)}{\int_{x=t_0}^{x=t} C_a(x)dx} \tag{6.3}$$

where $BI(t)$ is the bandwidth inefficiency for the aggregate reservation up to time t, defined as follows:

$$BI(t) = \int_{x=t_0}^{x=t} C_a(x) - r_a(x)dx \tag{6.4}$$

where $C_a(x)$ is the aggregate reservation size and $r_a(x)$ is the aggregate traffic rate. The bandwidth inefficiency parameter defined above expresses the amount of bandwidth reserved for the aggregate reservation that was not utilized, in (mega)bits. In order to compute the normalized bandwidth inefficiency $BI'(t)$ we have to measure:

(i) $C_a(t)$ which is the aggregate reservation size at time t, and

(ii) $r_a(t)$ which is the aggregate traffic rate at time t.

$C_a(t)$ and $r_a(t)$ are functions that express the fluctuations of the aggregate reservation size and the aggregate traffic rate respectively over time. Because these functions a piecewise step functions their integral can be measured quite easily. For example the aggregate reservation size function $C_a(t)$ can be measured by storing the different reservation sizes used during the experiments and the intervals at which the aggregate reservation size was updated. In other words whenever the aggregate reservation is updated the old size is stored along with the time stamp at which the reservation was updated. In a similar manner the aggregate traffic rate $r_a(t)$ is measured.

- The blocking probability P_{bl} can be expressed using the following equation:

$$P_{bl}\{r_a(t') > C_a(t)\} \leq \epsilon \tag{6.5}$$

where t' denotes the current time plus the reservation latency delay ($t' = t + \Delta t_{rl}$), $r_a(t')$ is the future aggregate traffic rate at the end of the reservation latency period, $C_a(t)$ is the current aggregate reservation size and ϵ is a small error term. In other words the probability that our future bandwidth over the period $[t, t+\Delta t]$ exceeds the current aggregate reservation size may be no more than ϵ.

In section 2.2.1 we have defined P_{bl} to be a counting process and thus Equation (6.5) can be rewritten as follows:

$$P\{N(t') - N(t) \geq k\} \leq \epsilon \tag{6.6}$$

where $N(t')$ describes the number of flows in the aggregate at time t', $N(t)$ describes the number of flows in the aggregate at time t and k is the number of flows that fit in the cushion. Notice that the blocking probability only applies to the flows arriving at the aggregator during the reservation latency period. All other flows arrivals are discarded. This makes measuring the blocking probability slightly complex and inaccurate.

If we want to measure the blocking probability we need to measure the flow arrivals during a reservation latency period. At the start of the reservation latency, when the QNE-Ingress is triggered to start the reservation of extra bandwidth, we should start counting the number of flow arrivals. In addition to the number of requests arriving at the QNE-Ingress, the number of rejected requests should be counted. At the end of the reservation latency period, thus when the QNE-Ingress has been notified that its request for extra bandwidth has been granted, the measuring should be stopped and the results stored. At the end of our simulation the number of end-to-end flows request that have arrive at the QNE-Ingress during a reservation latency should be counted along with the number of requests that have been rejected. Our measured blocking probability $\tilde{P}_{bl}$ is then computed as follows:

$$\tilde{P}_{bl} = \frac{\dot{n}_r}{\dot{n}_a} \tag{6.7}$$

where $\dot{n}_r$ is the total number of flow requests that have been rejected during the reservation latency periods and $\dot{n}_a$ is the total number of flows request that have arrived at the QNE-Ingress during the reservation latency periods.

In order to get an accurate value for $\tilde{P}_{bl}$ we need a large data set. For our simulations however we do not expect to have such a large data set. This is because the reservation latency periods are relatively small and the number of flows arriving during these periods is not expected to be large. In a real life scenario the number of flows arriving at the Ingress during the reservation latency period would be much greater hence making this parameter more meaningful.

- After the signaling load, the bandwidth inefficiency and the blocking probability have been measured the cost can be calculated:

$$c = \kappa_1 \times S(t) + \kappa_2 \times BI'(t) + \kappa_3 \times P_{bl} \tag{6.8}$$

where κ_1, κ_2 and κ_3 are predefined weights. As stated in the previous section we will set the values for κ_1, κ_2 and κ_3 to 33.3% so that all the parameters are evenly distributed over the cost function. For our performance experiments we will search for the minimum value of c for a given traffic model. The traffic model is discussed in more detail in Section 6.3.5. Using this traffic model we will set the values for T and ϵ of our aggregate update policy to different values and measure the signaling load, the normalized bandwidth inefficiency and the blocking probability. With these measured values we can then computed the minimal cost.

Note that if we want to accurately compute the minimum value for c then we need to measure the signaling load, bandwidth inefficiency and blocking probability for a large number of scenarios. Due to a time constraint and the limited amount of resources we cannot perform that many experiments and thus we need to define a minimal set of experiments that will be conducted. In our minimal set of experiments we will test how our aggregate update policy performs for:

(i) for one particular traffic model (which we will specify in Section 6.3.5),

(ii) a fixed value for $\epsilon = 0.005$ and $T = 24$ hours, 12 hours, 6 hours, 3 hours, 2 hours, 1 hour, 30 minutes, 20 minutes, 10 minutes, 5 minutes, 2 minutes and 1 minute and,

(iii) a fixed value for $T = 5$ minutes and $\epsilon = 0.25$, 0.1, 0.05, 0.025, 0.01, 0.005 and 0.001.

Compared to the latter experiments we will conduct more experiments for a fixed value of ϵ because we expect the T to have a significant impact on the performance whereas ϵ is expected to have little impact on the performance of our aggregate update policy. The choice of the values for T is exponentially because for large values of T we don't expect the results to differ much. It is the small values for T that will produce interesting result. For the small values of T is will be interesting to see whether the desired target inter update is reached in our implementation. Also, it will not be likely that the target blocking probability ϵ will be reached, for these values of T.

6.3.3 Assumptions

In Chapter 2 (Section 2.2.2) we have made the following assumptions:

- In our experiments the aggregation of only constant bit rate (CBR) flows is investigated. Thus we are not considering the fluctuation of the (data) traffic rate of an individual flow. Whenever an individual flow requests a certain amount of bandwidth and this request is granted then its rate will not exceed this limit. The data rate for this flow does not change during the flow's lifetime.
- The arrival process for flows can be described as a Poisson process. Poisson models are typically used to describe in telephony model to describe the call-arrival process. In these models calls are assumed to arrival according to a Poisson distribution and their lifetime is assumed to be exponentially distributed.
- The length of the reservation latency period is very small and during this period no flows leave the aggregate. This assumption is made in order to simplify the calculation of the cushion. Namely, if no flows leave the aggregate we simply have to calculate the amount of bandwidth needed to cope for the arriving during the reservation latency period. Flows leaving the aggregate during this period are not considered. For small values of the reservation latency this assumption might hold but for large value of the reservation latency period this certainly is not true. Note that due to this assumption our calculation of the cushion size is slightly pessimistic.
- The rate parameter λ of the Poisson process does not change during the reservation latency period. The fluctuations of the aggregate traffic rate on a large time scale are caused by the change in the intensity of the flow arrivals. This intensity is represented by the rate parameter λ. Increasing λ causes the (average) number of flows in the aggregate to grow while decreasing λ causes the (average) number of flows in the aggregate flow to shrink. This process however occurs on a large time scale. For the calculation of the cushion however a small time scale is considered and thus we can safely assume that the rate parameter λ remains constant during the reservation latency period.

In addition to the assumptions defined here we will not take congestion into account. Congestion has not been considered in our aggregate update policy and thus simulating congestion would have a serious negative impact on our aggregate update policy. This assumption is not a very restrictive one because in a real life scenario the links in a backbone network are kept under-utilized using engineering rules. Nevertheless, the impact of congestion on our aggregate update policy is a good subject for future work.

6.3.4 Scenario

Our test environment is set up to represent a basic signaling scenario in which the a sender and a receiver reserve multiple flows, according to an end-to-end QoSM, that traverse a RMD domain. In the RMD

domain an aggregate reservation has been set up onto which the end-to-end flows are mapped. The aggregate reservation is adjusted according to the flows joining or leaving the aggregate reservation. The topology of the test environment is displayed in Figure 6.2.

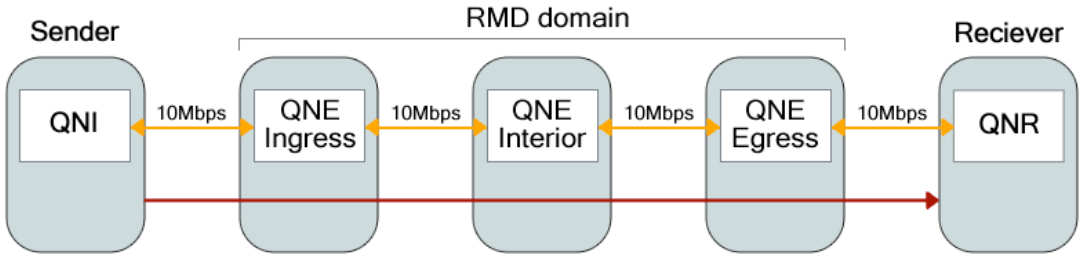


Figure 6.2: The topology of our test environment.

The QNI and QNR represent the sender and receiver of the end-to-end flows respectively. The RMD domain in our test environment consists of three nodes: the QNE-Ingress node, the QNE-Interior node and the QNE-Egress node. During our experiments end-to-end flows are set up and torn down by the QNI. At the QNE-Ingress the admission control for the end-to-end flows is performed. The QNE-Ingress is also the responsible for adjusting the aggregate reservation size according to the traffic demand generated by the end-to-end flows. For our performance measurements we need to measure the number of end-to-end flows joining and leaving the aggregate and the number of aggregate update message. All of these parameters can be measured at the QNE-Ingress and so data collection is done here.

6.3.5 Traffic model

In order to test the performance of our aggregate update policy we need to set up a traffic model which can be used to simulate real-time traffic behavior. Our implementation is intended to be used for real time application like voice and video and in order to simulate this behavior we need to create a traffic model which gives a correct representation of such applications. The basis for our traffic model will be the model for telephony systems. In such models the arrival of flows are modeled as a poisson process. This means that the inter arrival times of the flows is exponentially distributed. The holding times for these flows are also exponentially distributed.

In our traffic model we assume that the call arrival rate fluctuates over time. This fluctuation occurs on a larger time scale than the flow inter arrivals and we will refer to it as the *long term fluctuation*. This long term fluctuation can be simulated by adjusting the call arrival rate according to some long term behavior. For example the average number of telephone calls during business hours is higher than the average number of calls outside business hours. This is also true for our traffic model. The long term fluctuation used in our traffic model is comparable to the one described in Wang et al. [36] (also see Section 2.1.2). In Figure 6.3 the (normalized) call arriving rate during a period of a day is shown.

In Figure 6.3 we see the long term fluctuation over a period of a day used in Wang et al. [36]. In the beginning there is little traffic and the call arrival rate remains low. After 6 o' clock we see a steep increase of the call arrival rate. The rate drops to a moderate level at around 12 o' clock for an hour or so and then increases to a second peak. This peak lasts until 18 o' clock, after which it drops to a moderate level and finally at 23 o' clock it will drop to the level it started which.

For our experiments we will use this long term fluctuation. In our traffic model we assume that flows arrive according to a heterogeneous Poisson process with parameter $\lambda(t)$. In such a process the arrival intensity λ fluctuates over time. The behavior of our $\lambda(t)$ is defined by the long term fluctuation described

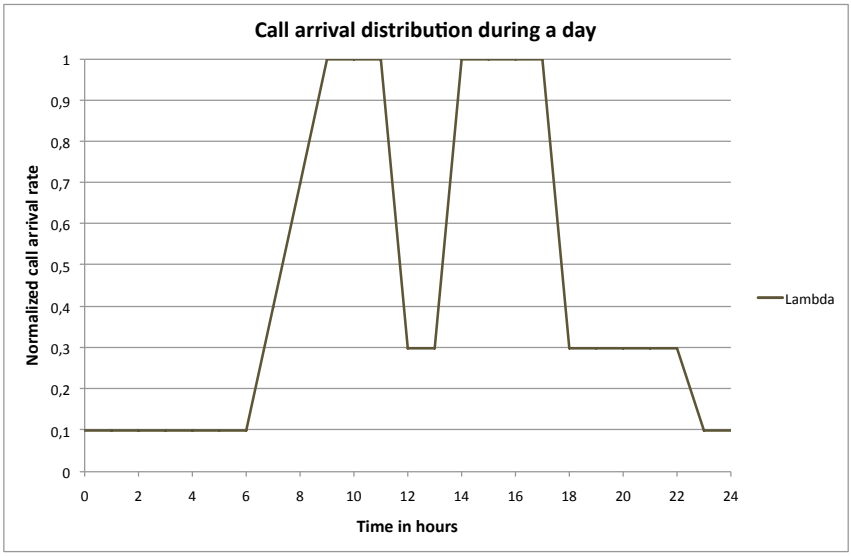


Figure 6.3: The call arrival distribution during a day.

above. In equation form $\lambda(t)$ is defined as follows:

$$\lambda(t) = \begin{cases} 0.1 & 0 \leq t < 6 \text{ and } 23 \leq t < 24 \\ 0.3t - 1.7 & 6 \leq t < 9 \\ 1.0 & 9 \leq t < 11 \text{ and } 14 \leq t < 17 \\ -0.7t + 8.7 & 11 \leq t < 12 \\ 0.3 & 12 \leq t < 13 \text{ and } 18 \leq t < 22 \\ 0.7t - 8.8 & 13 \leq t < 14 \\ -0.7t + 12.9 & 17 \leq t < 18 \end{cases} \tag{6.9}$$

where t is the time in hours. We are particularly interested in how our aggregate update policy adapts to the flow arrivals during the transition periods of $\lambda(t)$. The period between the two peaks (11 o' clock to 13 o' clock) is especially interesting because our update policy will not have much time to adjust to the drop in the flow arrivals.

Other parameters in our traffic model that we need to define are:

- The **actual call arrival rate** $\dot{\lambda}(t)$. In the previous section we have discussed how the long term fluctuations can be modeled using a heterogenous Poisson process. However only the normalized flow arrival rate $\lambda(t)$ has been discussed. The actual flow arrival rate is some scalar times $\lambda(t)$. In Wang et al.[36] the actual peak rate of the call arrival was not given. Instead two scenarios where used for testing: (1) a scenario with a peak arrival rate of 20 calls per minute and (2) a peak rate of 200 calls per minute. The average holding time for a call was set to 5 minutes. Thus this would result in the average number of calls in (1) being 100 during the peak period and in (2) this would be a 1000 calls. The difference between the two scenarios is the burstiness of the traffic. In scenario (1) the traffic is more bursty compared to scenario (2). This is because for a Poisson arrival/departure process, the variance decreases when the average number of calls increases. For our experiments we will simulate the scenario (1) and define the actual call arrival rate $\dot{\lambda}(t)$ (in calls per minute) to be:

  $$\dot{\lambda}(t) = 20 \times \lambda(t) \tag{6.10}$$

  where $\lambda(t)$ is the long term fluctuations as discussed previously. The equation above states that during the peak(s) of the long term fluctuation the average number of flow arrivals would be 20 flow arrivals per minute. Note that the value for $\dot{\lambda}(t)$ has been chosen such that the amount of

flows generated using this traffic model can be supported by the links in our network. Along with the bandwidth requirement for an end-to-end flow, $\dot{\lambda}(t)$ is chosen such that the load on our network links is never more than 80% of the link capacity.

- The **average holding time** for an end-to-end flow. We have stated above that the end-to-end flows have exponentially distributed holding times. In order to calculate the values for the holding times used in our simulation we have to specify the average holding time parameter λ_{ht} which is used in our exponential distribution. Varying this parameter has a large impact on the fluctuation of our aggregate traffic demand. Setting λ_{ht} to a high value will cause flows to stay longer in the aggregate which will result in an higher aggregate demand. Setting λ_{ht} to a low value will cause flows to leave the aggregate much sooner. For our simulations we will set the average holding time for and end-to-end flow (λ_{ht}) to 5 minutes. This value is also used for the average holding time for the flows used in the simulation of Wang et al. [36].

- The **bandwidth requirement** of an end-to-end flow. In our traffic model we will only use CBR flows. In addition we will use flows with equal bandwidth requirements. Again setting the bandwidth requirement for an end-to-end flow to different values has an impact on the performance of our aggregate update policy. If flows with large bandwidth requirements are used then the fluctuations of the aggregate traffic demand will be larger. In our traffic model we will not fully explore the effects of using different bandwidth requirements. Here we will simply use flows with a bandwidth requirement of 50 kbps (kilo bits per second). The reason why 50 kbps was chosen is because the maximum throughput of our network links is 10 Mbps. If we don't want our network to get congested we should never set the load to be more than 80% of the link capacity. In our case this would be 8 Mbps. In the previous paragraph we have set the peak call arrival rate to be 20 calls per minute. The average flow holding time is set at 5 minutes and thus during the peaks in our traffic model we would have an average of 100 flows in the system. In other words on average our aggregate traffic rate would be about 5 Mbps during peak times.

Using the parameters discussed above we have created a script that generates an aggregate traffic rate for our simulations. The generated aggregate traffic rate used in our experiments is shown in Figure 6.4. The characteristics for this traffic is stored in a file which is loaded into our client at the beginning of every experiment in order to start the simulation.

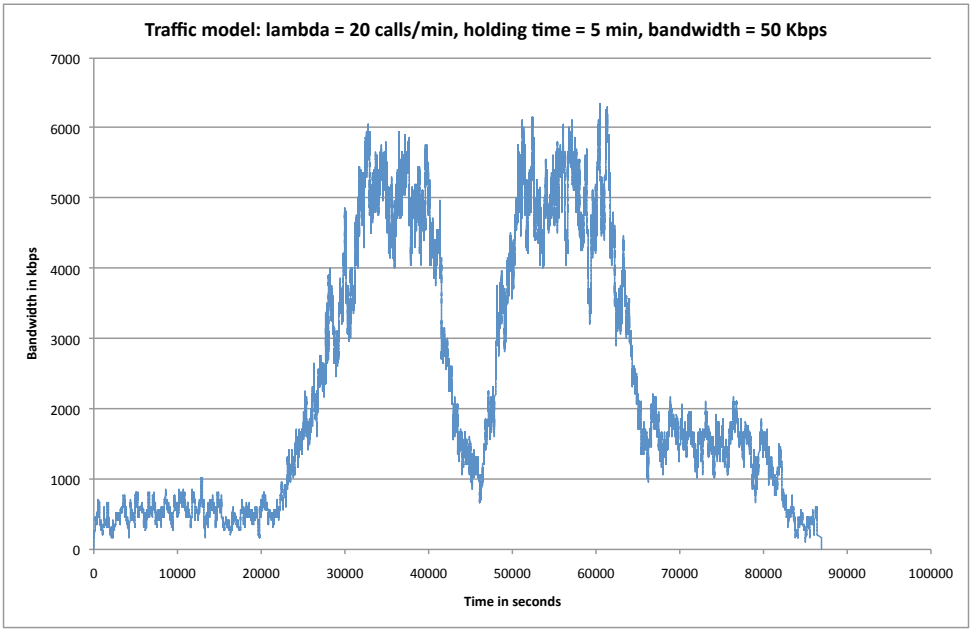


Figure 6.4: The aggregate traffic demand used in our simulations.

In Figure 6.4 the aggregate traffic rate (r_a) is shown. Here we see how the long term fluctuation follows the fluctuation of $\lambda(t)$. The aggregate traffic rate is rather low for the first 6 hours (= 21600 seconds), followed by two peaks, the first between the 6^{th} and 12^{th} hour (= 43200 seconds) and the second between 13^{th} (= 46800 seconds) and 18^{th} hour (= 68400 seconds). After the peaks the aggregate traffic rate returns to the level it started out with. The maximum peak rate of the aggregate traffic rate is measured to be 6350 kbps. On a smaller timescale we see how the Poisson arrivals result in the burstiness of the aggregate traffic rate. The aggregate traffic rate is not smooth but has minor fluctuations that are caused by the flows joining or leaving our network. Notice how during peak times these bursts are larger than during less busy hours. This aggregate traffic rate will be simulated in our experiments and the results are discussed in the next section.

6.3.6 Results

In the following sections we will discus the results of our performance experiments. First we will discuss the effects of choosing different values for T for a given ϵ (= 0.005) followed by a discussion of the impact of changing ϵ for a given value for T (= 300 seconds). The results of our performance experiments are shown in Appendix D. Before discussing the details of our results we will present some preliminary findings:

- From initial testing it became apparent that the results are sensitive to the chosen bootstrapping values for our aggregate reservation. This is especially true for large values of T. When T is set to a rather large value (e.g. 12 hours) then the aggregate update policy will attempt to set the width of the operating region (OR) such that the size of future update intervals is in the vicinity of T. If we set the initial values for our aggregate reservation (C_a) too low then the first update will occur very soon. The update policy will assume that its OR is too small and increase the width of the new OR drastically. This value can even be larger than the maximum link capacity[1]. For small values of T this is not an issue because the correction on the OR will not be as much and because T is small any miscalculation will be corrected within the next update interval such that the new update interval size is close to T.

- In our proposed solution we have set our cushion to be constrained a the maximum value. The maximum value for the cushion is set to be $\frac{OR}{2}$. However during our testing we have concluded that the cushion should be constrained with w a minimum value. If the size cushion is less than the bandwidth demand of one individual flow, it will not be possible to increase the aggregate reservation. This is because our high threshold Δ_{high} is set to $C_a - \Delta_{cushion}$. An increase of the aggregate reservation is triggered whenever the aggregate traffic rate r_a crosses Δ_{high}. However the value of r_a can never be more than C_a, thus an increase it triggered when $\Delta_{high} < r_a < C_a$. Now if the cushion $\Delta_{cushion}$ is smaller than the bandwidth demand of a single flow, the probability of r_a falling in between C_a and $\Delta_{cushion}$ gets very small. In order to avoid this problem we have constrained the value of our cushion $\Delta_{cushion}$ to be between $\frac{OR}{2}$ and the bandwidth requirement for one flow.

- The performance of our aggregate is not very satisfying for large values of T. When T is set to some large value our aggregate update policy will attempt to reserve extra bandwidth such that the average inter update time is close to this value. Because this value is rather high our algorithm will overshoot its target and attempt to increase the aggregate reservation to a significantly large value. The request for this extremely large amount of bandwidth is rejected in many cases because if granted the aggregate reservation will exceed the maximum link capacity. When the request for extra bandwidth has been rejected the update policy will still attempt to reserve some extra bandwidth. For the computation of the new OR the same values are used except the value for the current inter update T_{curr}. This value will be slightly larger because the time passed during the rejection of the previous request is added. Because the values for the parameters in the computation

[1] In this case the request for extra bandwidth will be rejected.

have not changed significantly, the newly computed value for OR will be slightly less than the previously computed OR. The request for this extra bandwidth is most likely to be rejected again which brings us back to the beginning of the process. As time passes, the value of T_{curr} keeps increasing resulting in a decreasing OR. Eventually the request for extra bandwidth will be granted and the reservation size is set to a value just below the maximum link capacity. In the mean time many end-to-end flows will have to be rejected because the aggregate reservation could not be increased.

Also because a symmetric OR is used the lower threshold will be set to an extremely low value. In many cases this will result in a negative lower threshold, which will be set to zero by our update policy because negative thresholds are not allowed. When the lower threshold is set to zero it will be impossible to decrease the aggregate reservation.

Varying T for $\epsilon = 0.005$

For the first set of experiments we have set the ϵ parameter of our aggregate update policy to a fixed value (0.005). The second parameter of our aggregate update policy, the target inter update T is set to the following values: 24 hours, 12 hours, 6 hours, 3 hours, 2 hours, 1 hour, 30 minutes, 20 minutes, 10 minutes, 5 minutes, 2 minutes and 1 minute. For every value of T the generated traffic rate is fed into our network and the performance parameters (discussed above) measured. Note that for every T the experiment is run only once, due to time constraints. This implies that we can not draw detailed conclusion based on the results but only spot trends in the graphs. Based on the measurements of the signaling load $S(t)$, the bandwidth inefficiency $BI(t)$, the (measured) blocking probability P_{bl} and the cost c are calculated. The results can be found in Appendix D.1. Based on these results we have generated graphs for $S(t)$ (see Figure 6.5), $BI(t)$ (see Figure 6.7), P_{bl} (see Figure 6.8) and c (see Figure 6.10). For the cost function two graphs have generated because it seems that the range of the $S(t)$ and P_{bl} is very small compared to the range of $BI(t)$. As a result the cost function closely resembles $BI(t)$. In order to increase the contribution of $S(t)$ and P_{bl} on the cost we have created a second normalized cost function. The details are discussed in the relevant paragraph below.

Signaling load. The signaling load for the different values of T is shown in Figure 6.5.

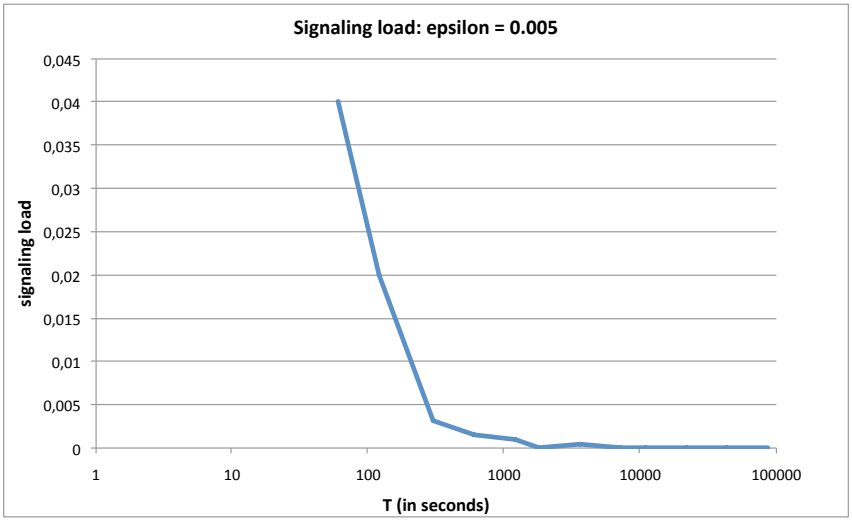


Figure 6.5: The measured signaling load.

The first thing to note is the logarithmic scale used in the graph. We have assumed that for large values of T the results are expected to be quite similar. It is the smaller values for T that should show some

interesting results. Because of the chosen values for T we can best display our results in a graph with a logarithmic x-axis. This logarithmic scale is also used in the other graphs.

From Figure 6.5 we can conclude that the signaling load behaves as expected. For small values of T the signaling load is high while for large values for T the signaling load is low. When using small values for T the measured values for the signaling load correspond to the signaling load one would expect when using an IntServ architecture. Large values for T result in a more static aggregate update policy, having a low signaling load, which can be found in networks using the DiffServ architecture. As expected we see a large drop in the signaling load when increasing T from 60 seconds. The decrease in the signaling load is initially quite large and eventually flattens out for large values for T. Although this is the expected result some remarks can be made:

- First the maximum value for $S(t)$ in the graph is around 0.04. This is significantly lower then our stipulated maximum value of $S(t)$ which is 1. In Section 1.4.2 we have assumed that if we update the aggregate for every arriving or departing flow we would get a signaling load of 1. This is the maximum value for our signaling load and corresponds to the performance when using an IntServ architecture. This however is not possible in our prototype because of a waiting timer we have implemented in our prototype. After every request for extra bandwidth the timer is started and while the timer is running no increase of the aggregate reservation is allowed. The value of this timer is set to 30 seconds. This implies that the value for T in our prototype can not be lower than 30 seconds. Thus the maximum amount of increases (over a period of a day) that can be realized with our prototype is $\frac{86400}{30} = 2880$.

  Of our prototype implementation a maximum value of 1 for $S(t)$ is not a realistic value. If we want to calculate a more realistic maximum value for $S(t)$ then we not only have to know the maximum amount of increase possible with our prototype but also the maximum amount of decreases. The number of decreases is however not constrained by our timer but let's assume that the number of decreases is more or less equal to the number of increases. In this case we get a maximum signaling load of $\frac{5760}{13155+13155} \approx 0.21892816$. This is still relatively small compared to our maximum value of 1.

- Initially there is a large drop in the signaling load when increasing T beyond 60 seconds. Although this was the expected behavior we cannot help but wonder if such a large decrease is realistic. In order to verify whether the measured increase is in line with the expected signaling load we will compute an 'ideal' signaling load an compare it to our measured signaling load. Our 'ideal' signaling load is calculated based on the assumption that if T is set to a particular value then the inter update times are indeed T. Thus for example if T is set to 24 hours then in our ideal situation the number of updates would be 1, whereas is T is set to 5 minutes then the amount of updates would be 288. In order to calculate the signaling load according to Equation 1.4 in Section 1.4.2 all we need to know is the amount of flows that have been admitted to the aggregate and the amount of flows that have departed from the aggregate. If we assume that all the flows in our aggregate traffic rate are admitted to the aggregate and at the end of our simulation have departed from our aggregate reservation then we can easily calculate the 'ideal' signaling load $\dot{S}(t)$ as follow:

$$\dot{S}(t) = \frac{\tilde{n}_u}{\tilde{n}_a + \tilde{n}_d} \tag{6.11}$$

  where $\tilde{n}_u$ is our stipulated 'ideal' amount of update, $\tilde{n}_a$ the amount of flows that have been accepted (which is 13155) and $\tilde{n}_d$ the amount of flows that have been rejected (which is 13155). The results for our 'ideal' signaling load is shown in Figure 6.6.

  Looking at Figure 6.6 we see that the measured signaling load closely resembles our ideal signaling load. Both graphs show an exponential decrease when increasing T and the difference between the two is minimal. The maximum difference between our ideal signaling load function and the measured signaling load is 0.014636576. The minimum difference and average difference is 4.7095×10^{-18} and 0.003128968 respectively. In other words our aggregate update policy performs close to ideal in terms of signaling load.

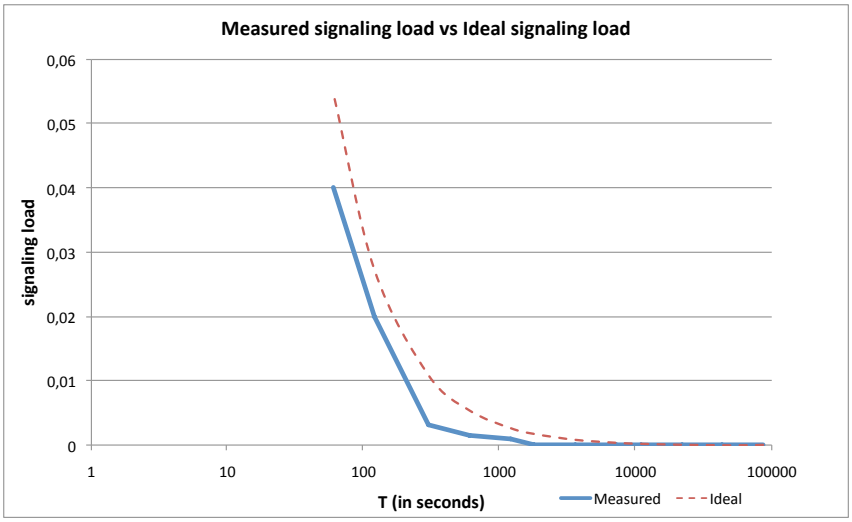


Figure 6.6: The measured signaling load vs the 'ideal' signaling load.

Bandwidth inefficiency The bandwidth inefficiency for different values of T is shown in Figure 6.7.

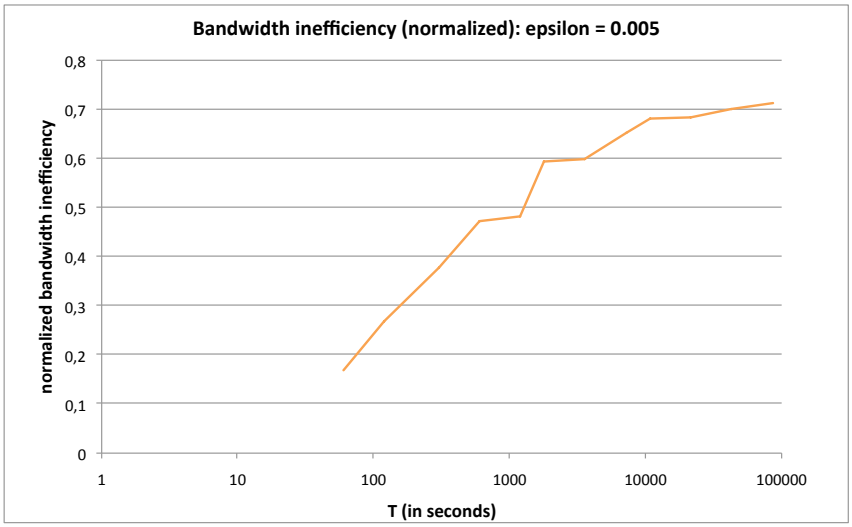


Figure 6.7: The measured (normalized) bandwidth inefficiency.

In Figure 6.7 a low bandwidth inefficiency is shown for small values of T and for large values of T a high bandwidth inefficiency is shown. Thus in other words, the utilization is high when using a small value for T and low when using a large value for T. This is logical because for high values of T the aggregate reservation is not updated very often meaning that a large amount of extra bandwidth is reserved. The amount of the reserved bandwidth that is actually utilized in this case would be rather low resulting in a high bandwidth inefficiency. When a small value for T is chosen, the aggregate update policy would update the aggregate reservation more often. The amount of extra bandwidth being reserved in this case would be much lower resulting in a higher utilization and thus a lower bandwidth inefficiency.

Furthermore we can see a somewhat linear function for the bandwidth inefficiency. Note that the graph has a logarithmic scale x-axis and thus the function cannot be considered linear. But for an exponentially increasing T the (normalized) bandwidth inefficiency $BI'(t)$ increases somewhat linearly. This is particularly interesting because the signaling load decreases exponentially for increasing T. In other words the if we slightly increase T from 60 seconds we get a large decrease in the signaling load while only getting

a small increase in our bandwidth inefficiency.

Looking at Figure 6.7 we see how the linear trend weakens for the larger values of T. One of the reasons why the trend weakens is because the aggregate reservation size C_a is restricted by the maximum link capacity of our network. In other words C_a can never be more than the maximum link capacity, which is 10 Mbps in our network. As we shall see in the next paragraph, for large values of T our update policy will attempt to reserve the maximum available bandwidth. When this maximum is reserved the update policy will not be able to increase the aggregate and thus reaching the maximum bandwidth inefficiency. The absolute maximum value for $BI'(t)$ can be computed as follows:

- Let's assume that at the start of the experiment the aggregate reservation size is set to the maximum link capacity, which is 10 Mbps. Further more we assume that the aggregate reservation size C_a is not decrease during the experiment. In this case we can compute the area for $C_a = 86400 \times 10000 = 864000000$[2] Kb.
- In our experiments the measured area of r_a in the case that no flows were rejected is 198186000 Kb. Using Equation (1.6) we can now compute the maximum (normalized) bandwidth inefficiency $BI'(t)$:

$$BI'(t) = \frac{\int_{x=t_0}^{x=t} C_a(x) - r_a dx}{\int_{x=t_0}^{x=t} C_a(x) dx} = \frac{864000000 - 198186000}{864000000} \approx 0,77061805555 \tag{6.12}$$

It is clear from Figure 6.7 that the larger values for T, the (normalized) bandwidth inefficiency approach our calculated maximum. Note that if our aggregate reservation size would not have been restricted by the maximum link capacity of our network then most likely we would also see a linear trend for $BI'(t)$ for the larger values of T.

Blocking probability The blocking probability for different values of T is shown in Figure 6.8.

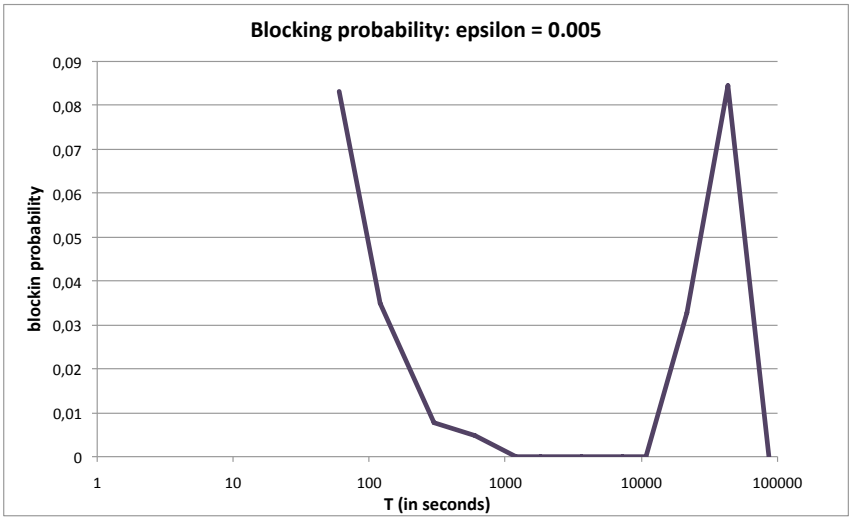


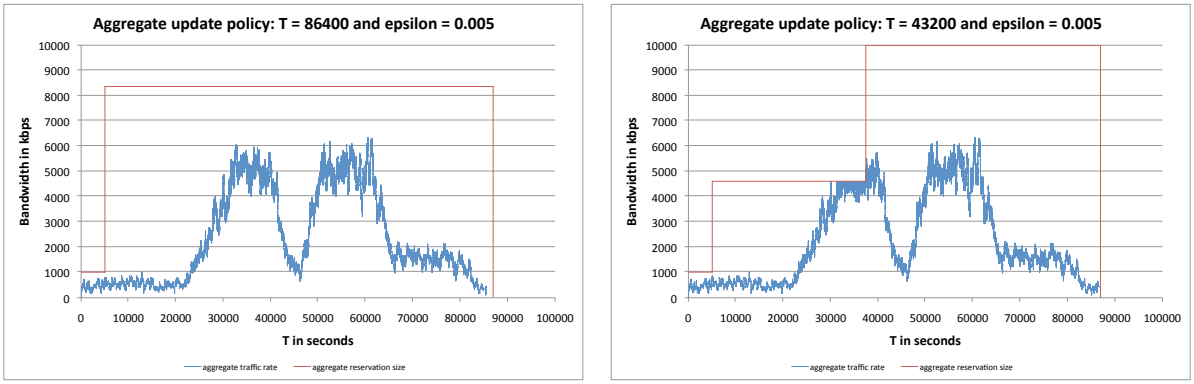
Figure 6.8: The (measured) blocking probability.

Figure 6.8 shows some surprising results. For large values of T we see a large peak in our graph while for the largest value of T (= 86400 seconds) the measured blocking probability is 0. For small values of T the blocking probability shows an exponential drop for increasing T. This can be explained as follows:

[2]For this calculation we will state that 1 MB = 1000 Kb.

- Flows are usually rejected when the aggregate reservation needs to be increased. It is during the reservation latency period that the end-to-end flows can be rejected. When T is small the aggregate reservation is often updated, increasing the probability of having to reject flows. This is because there will be more updates and thus more reservation latency period in which flows may have to be rejected. For large values of T there a less aggregate updates and thus less reservation latency period in which flow can be rejected.

For $T = 21600$ and $T = 43200$ seconds we see an anomaly in our graph. Here the measured blocking probability is 0.032817924 and 0.084514874 for $T = 21600$ seconds and $T = 43200$ seconds respectively. Strangely enough the measured blocking probability for $T = 10800$ and $T = 86400$ seconds is 0. In order to investigate this phenomenon we have drawn the aggregate traffic and aggregate reservation in one figure. In Figure 6.9 the aggregate reservation for $T = 84600$ and $T = 43200$ is shown.



(a) (b)

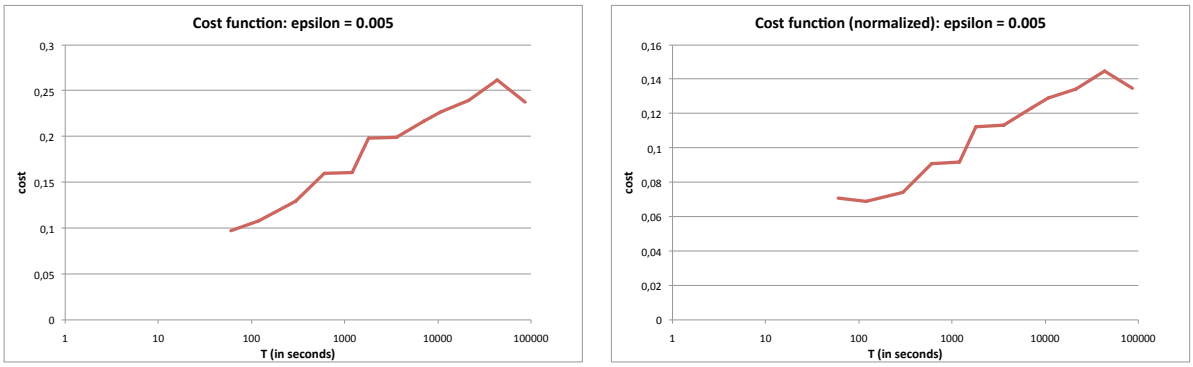
Figure 6.9: Aggregate update for T = 86400 (a) and T = 43200 (b)

From Figure 6.9(a), where $T = 86400$ seconds, we can conclude that the first increase for the aggregate reservation is enough to grant all flow arrivals in our aggregate traffic rate. After 5100 seconds in the experiments the first increase is triggered. The new value for the aggregate reservation is set to 8366 kbps which is more than 8 times the initial value of the aggregate reservation, which was 1000 kbps. Note that in order to achieve such a large increase the newly calculated value for new operating region (OR) would be about 14732 kbps ($= 2 \times 7366$). As a result the lower threshold Δ_{low}, which is calculated as follows $\Delta_{low} = C_a - \frac{OR}{2}$, will be set to zero because is cannot assume a negative value. If our Δ_{low} it means that a decrease of our aggregate reservation will never be triggered. This can have a serious impact on the performance of our aggregate update policy, with respect to the bandwidth inefficiency.

If we look at Figure 6.9(b), in the case of $T = 43200$ seconds, the first increase is not enough to cope with all flow arrivals over the entire simulation. The second increase sets the reserved bandwidth for the aggregate reservation just under the maximum link capacity allowing all the incoming flows to be admitted. After that the aggregate reservation is not updated anymore. The first increase is triggered around 5000 seconds in the simulation, meaning that the size of the first update interval is about 5000 seconds. This is significantly lower than the 43200 seconds which is set as the targeted inter update time. For the computation of the OR for the next interval the update policy will use the current values of the OR and inter update time. The update policy assumes that the size of the inter update interval is mainly determined by the size of the OR. Because the size of the previous inter update interval is smaller compared to the targeted interval, the update policy will compute a larger value for OR in order to compensate for the next inter update interval. Now because the difference between the measured inter update interval and the target T is rather large the new value for OR will sky rocket. However the maximum link capacity for the routers in the test network is 10 Mbps. Thus the request for extra bandwidth, which would result in a aggregate reservation size of more than 10 Mbps, will fail. After the request has failed the update policy will keep trying to increase the aggregate reservation. Note that the

requests are controlled by a timer. Thus after the increase has failed, the update policy will have to wait at least 30 seconds before attempting to increase the aggregate reservation again. The calculation of the new values for OR are still calculated based on the previous and current inter update intervals an thus will be rather high. However as time passes, the calculated value for OR will keep decreasing. This is because the current inter update interval keeps increasing with every failed increase of the aggregate reservation. Eventually an increase will be calculated which sets the new value of the aggregate reservation just below our maximum link capacity. In this case the bandwidth for the increase is granted and the aggregate reservation updated. Flows arriving at the our Ingress node during this period, are rejected because the aggregate reservation has not enough room for them. During the time period of 3500 seconds and 4000 seconds (in Figure 6.9(b)) we can clearly see that many flows have been rejected due to the failure of increasing the aggregate reservation. This explains the peak in our measured blocking probability graph.

Cost The cost for different values of T is shown in Figures 6.10(a) and 6.10(b).



(a) (b)

Figure 6.10: The cost function (a) and the normalized cost function (b)

For the cost function two graphs, representing the cost versus T, have been generated. The first graph, shown in Figure 6.10(a) displays the cost function for the measured values for $S(t)$, $BI(t)$ and P_{bl} with the weight set to $\kappa_1 = \kappa_2 = \kappa_3 = 33.3\%$. As stated in the previous paragraphs the range of the different functions differ significantly from each other resulting in a cost function that closely resembles the function with the highest range, $BI(t)$ in this case. In order to compensate for these large differences we have created a normalized cost function displayed in Figure 6.10(b).

If we look at the results for our cost function in Figure 6.10(a) we see that indeed it closely resembles the graph of the (normalized) bandwidth inefficiency shown in Figure 6.7. The main reason why this is, is because of the range of the bandwidth inefficiency fluctuates between 0 en 0.72 while the signaling load and blocking probability fluctuate from 0 to 0.04 and 0 to 0.07 respectively. Because of the large difference in ranges the bandwidth inefficiency has the upper hand in the computation of the cost. In order to allow the other parameters to have a larger impact on the outcome of the cost we have attempted to normalized the results. Also the bandwidth inefficiency function is a rather linear one meaning that our cost function is also more or less linear, and thus having no optimum value.

The normalized cost function shown in Figure 6.10(b) is computed as follows:

- The values for the signaling load have been normalized according to the maximum value for the signaling load which can be achieved in our prototype implementation. As stated previously a back-off timer restricts our aggregate update policy to update the aggregate reservation more often than 30 seconds. In other words, the closest we can get to the maximum signaling load is to update the aggregate every 30 seconds. In the previous paragraph we have computed the maximum signaling load to be approximately 0.21892816. Note that this value is only valid for our particular traffic

model used for the experiments. If we would generate another traffic model then the maximum signaling load would be different due to the different number of flow arrivals (and departures). The measured results for our signaling load will be normalized by dividing them with our computed maximum.

- The maximum value for the (normalized) bandwidth inefficiency has been computed in one of the previous paragraphs. This value (0.77061805555) will be used to normalize the measurements of $BI'(t)$.
- For the blocking probability the measured results will not be normalized. The reason of this is because the blocking probability is not constrained by any boundaries. Whereas the signaling load is constrained by a back-off timer and the (normalized) bandwidth inefficiency is constrained by the maximum link capacity, the blocking probability knows no boundaries. The measured values are indeed low and will not have a massive impact on the cost but this was expected.

The normalized cost function shown in Figure 6.10(b) is quite similar to the original cost function shown in Figure 6.10(a). Only for the small values of T the difference between the two is apparent; the normalized cost function is slightly curved upward while the original cost function remains more or less linear. The curve in the normalized cost function is caused by the increased influence of the signaling load which, for small values for T is rather high compared to the large values for T. Because of this curve the normalized cost function has an optimum value, for $T = 120$ sec the normalized cost is minimal. Although this result is not very convincing it illustrates the purpose of our cost function. The idea of the cost function was to express the tradeoff between the three parameters as a single parameter and based on it curve compute an optimal value for T and ϵ. Sadly enough we feel that our cost function does not achieve this goal. However we do note that the results of measured $S(t)$, $BI'(t)$ and P_{bl} show promising results. For example, the fact that $S(t)$ and P_{bl} drop exponentially for increasing T, while the $BI'(t)$ increases linearly tells us that an optimum value should be found for some small value of T. Maybe using a more complex cost function, this optimum value can be computed more accurately.

Varying ϵ for $T = 300$ seconds

The results above have been measured for our aggregate update policy with the ϵ parameter set to a fixed value. In the next set of experiments we will set the value for T to be fixed at 300 seconds. The values for ϵ will be set to the following values: 0.25, 0.1, 0.05, 0.025, 0.01, 0.005 and 0.001. Again, due to time constraints the experiment is run only once for every ϵ. This implies that we can not draw detailed conclusion based on the results but only spot trends in the graphs.

The parameter ϵ is mainly used for the calculation of the *cushion* (and the high threshold Δ_{high}). The size of the *cushion* determines the offset for the high threshold Δ_{high} from the aggregate reservation size C_a. This threshold in turn determines when the update policy should trigger an increase for the aggregate reservation. Setting ϵ to a high value will result in a the small value for the *cushion* resulting in the update policy to increase the aggregate in an earlier stage. For large values of ϵ the aggregate update policy is more relaxed and will trigger an increase at a later time. As stated before we don't think that changing the ϵ will have a big effect on the performance of our update policy. The parameter ϵ has an impact on when the aggregate is update but when it comes to the computation of the size of the OR this parameter plays a small role. Here T is most likely to have the upper hand.

Signaling load. The signaling load for different values of ϵ is shown in Figure 6.11.

For Figure 6.11 we can conclude that the signaling load in not affected by increasing ϵ. Initially the signaling load remains the same for increasing ϵ however for $\epsilon = 0.1$ we see an increase in the signaling load. In order to analyze why the signaling load increases for increasing ϵ we have generated the following graphs.

Looking at the two figures we can conclude the following :

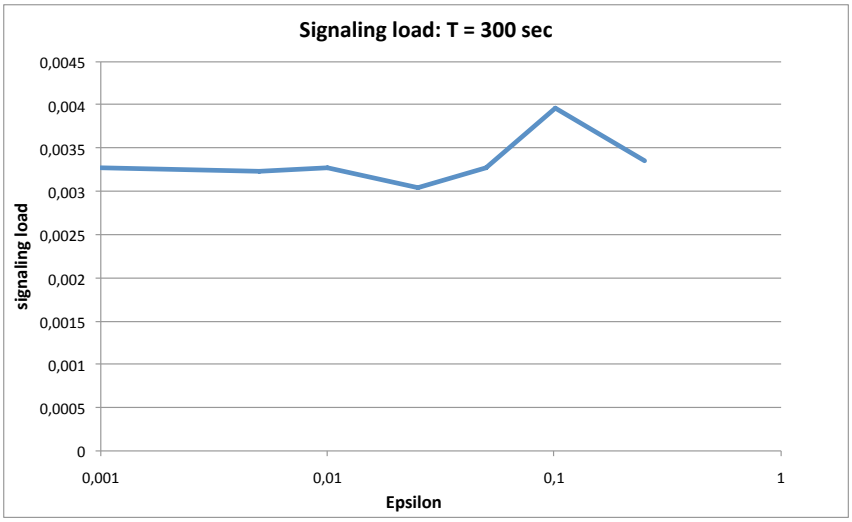
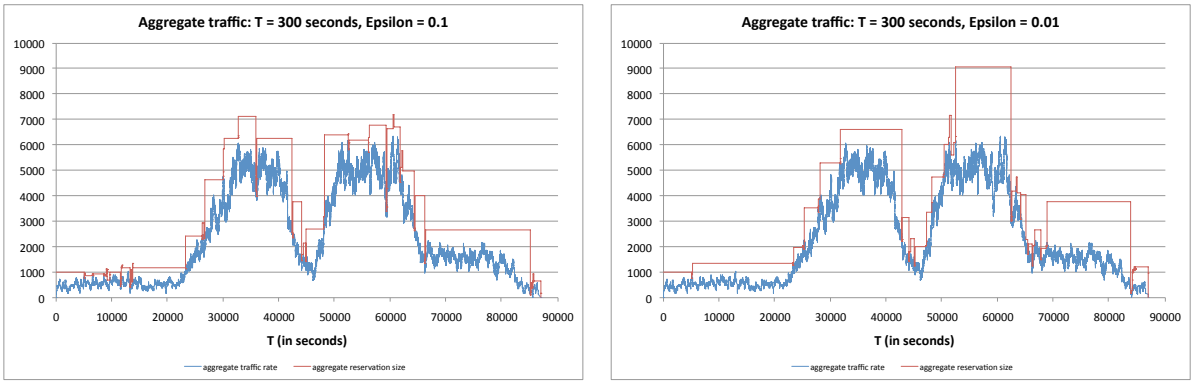


Figure 6.11: The signaling load.



(a) (b)

Figure 6.12: Aggregate update for $\epsilon = 0.1$ (a) and $\epsilon = 0.01$ (b)

- For large value for ϵ the update policy has the tendency to follow the aggregate traffic demand. This can be seen in Figure 6.12(b) during the increase and decrease of the second peak. Here the aggregate reservation size becomes rather bursty and somehow closely follows the aggregate traffic rate.

  We question whether the results shown here are consistent for all increasing values of ϵ or whether the computed aggregate reservation is simply updated at the wrong time. In a simple case of bad luck our update policy would have to increase the aggregate reservation, and shortly after have to decrease it again due to a large drop in the aggregate traffic demand. If we look more closely at aggregate traffic demand we see that the at the start of the seconds peak (around 53000 seconds), the fluctuation in the aggregate traffic demand are rather high. As a result the aggregate reservation is increase but shortly afterwards increased again. The following increases occur in a short period of time resulting in the aggregate reservation going through the roof. The aggregate reservation has been set to about 9000 kbps and after quite some time the aggregate update policy is triggered to decrease the reservation. During this decrease this new aggregate size is set to a value in a dip of the aggregate traffic demand (at about 61000 seconds). Immediately after the aggregate reservation has been decreased it needs to be increased again due to a small peak in the aggregate traffic demand. It is certainly possible that the sum of these unfortunate updates have resulted in a (small) increase in the signaling load.

- Figure 6.12(b) shows promising results. During most of updates the aggregate reservation remains stable for a rather large period of time. We do however conclude that the average size of the inter update times seem to be rather large. Our aggregate update policy is set to target an inter update time of 300 seconds and looking at Figure 6.12(b) many large update intervals can be seen.

Aside from this anomaly described above the graph for $S(t)$ seems to meet our expectations.

Bandwidth inefficiency. The bandwidth inefficiency for different values of ϵ is shown in Figure 6.13.

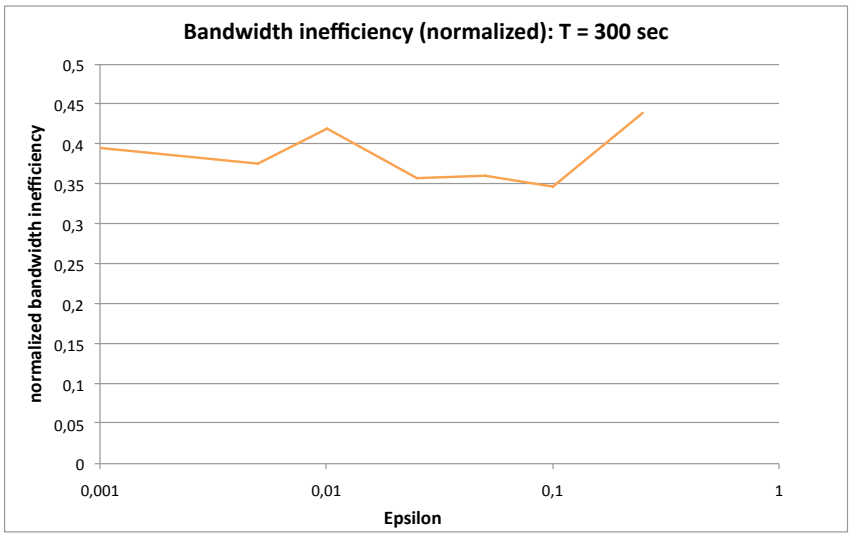


Figure 6.13: The bandwidth inefficiency.

Looking at the (normalized) bandwidth inefficiency graph we see a rather stable horizontal trend. Again here a peak is seen for $\epsilon = 0.01$. If we look at Figure 6.12(b) we can see that this is not that strange. The update policy seems to have made some crucial mistakes at certain point which have resulted in a rapid increase in the aggregate reservation. Because the computed values for the OR are also most likely to be very large the aggregate reservation remains high for quite some time. The reasoning here is that the low threshold is set very low during a rapid increase and thus a decrease is postponed for quite some time. In the mean time the excessive bandwidth is not used and thus a large bandwidth inefficiency is measured. The slight increase in the (normalized) bandwidth inefficiency for $\epsilon = 0.025$ can also be explained as some specific bad judgement calls on behalf of the aggregate. In our generated graph (not shown here) a large increase at the seconds peaks is shown and after the seconds peak the aggregate reservation remains high for quite some times. A similar pattern as the one in Figure 6.12(b) can be seen.

Blocking probability. The blocking probability for different values of ϵ is shown in Figure 6.14.

The results for the blocking probability seem so be inconclusive. From Figure 6.14 we can not draw many conclusions. It seems that the blocking probability shows an increasing trend but due to the large fluctuations in the graph we cannot conclusively state whether this is true. As for the measured blocking probability it seems that the targeted probability ϵ is never reached. In our experiments we have measured the average number of aggregate updates during one experiment to be 87. During these updates periods the average number of flow arrival within a reservation latency periods is about 3000 flows. The probability is the fraction of these 3000 flows that have been rejected. If we set our ϵ to 0.001 for example this would imply that only 3 flows are allowed to be rejected. From Figure 6.14 we can conclude that this target is never almost reached.

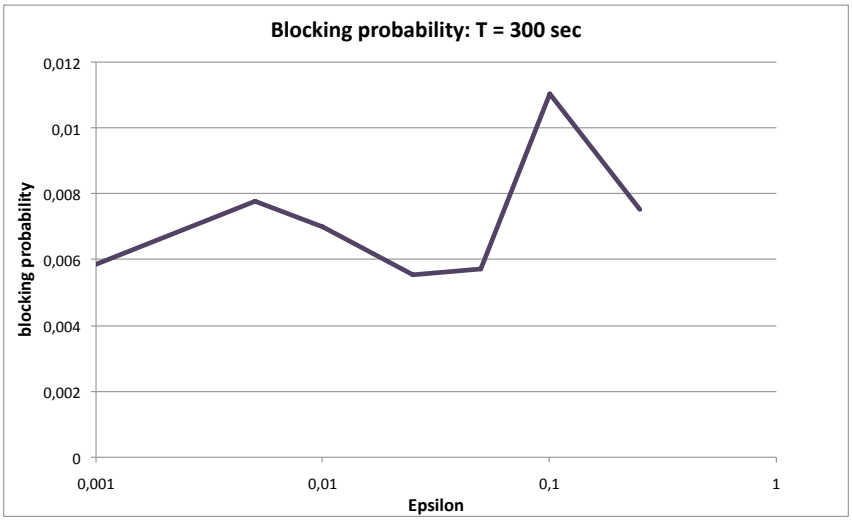


Figure 6.14: The (measured) blocking probability.

Cost The cost function for different values of ϵ is shown in Figure 6.15.

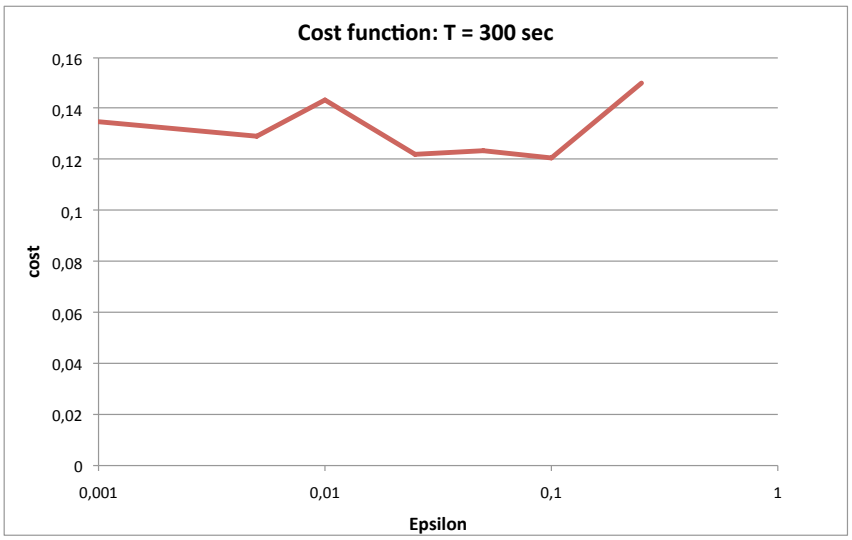


Figure 6.15: The cost function.

For this set of experiments we have chosen not to show the graph for the normalized cost function. The reason why is because the results are similar to the cost function shown in Figure 6.15. The form is curve is the more or less the same but only the values are different. If we look at the cost function we see that it remains horizontal for all the experiments. Because the curve shows some fluctuations we can state that it has a minimum value at $\epsilon = 0.1$. However, this value is not very reliable. The reason why the graph is horizontal is because the graphs of both the signaling load and the normalized bandwidth inefficiency is horizontal. The graph do not show an increasing nor decreasing trend rendering our cost function pretty much useless. The only thing that can be concluded here is that the impact of the ϵ parameter on the performance of our aggregate update policy is very small.

Chapter 7

Conclusion

The main goal of this assignment was to design and implement an aggregate update policy which solves the following optimization problem:

> *Can we design an aggregate update policy so that the signaling load, the (normalized) bandwidth inefficiency and the blocking probability are as low as possible? Or in other words, when do we need to update the aggregate reservation and what should the new size be so that the signaling load, the (normalized) bandwidth inefficiency and the blocking probability are as low as possible?*

In this thesis a solution for the aggregation of end-to-end flows in a backbone network, using such a policy, has been presented. In Chapter 1 the aggregation problem has been defined, followed by a discussion of the relevant algorithms used in flow aggregation and our proposed solution in Chapter 2. A brief overview of the NSIS framework has been given in Chapter 3, followed by the RMD QoSM specification used for the aggregation of end-to-end flows in Chapter 4. Finally the design and implementation of our prototype has been discussed in Chapter 5 followed by a discussion on the experimental evaluation of our prototype in Chapter 6. In this final chapter we will conclude whether the predefined goals have been reached or not. In addition, we will give some recommendations for future work.

7.1 Conclusion

Much research has been conducted on how to address the optimization problem before designing and implementing the update policy. After the aggregate update policy has been defined, it was incorporated in the RMD-QoSM specification. This QoSM specification was implemented as an extension on an existing prototype implementation. The existing design and implementation of this prototype have been examined and has led to redesign of certain parts of the prototype, the NSLP layer being completely redesigned. Because of the extensive changes made to the prototype implementation, exhaustive testing was required. After a set of functional experiments, we can conclude that the specified QoSM has been successfully implemented. The implementation of the aggregation based RMD-QoSM has passed all the functional experiments, meaning that the implementation adheres the defined specifications.

The aggregate update policy defined in our proposed solution has two parameters (T and ϵ) which can be configured to tune the performance of the update policy. The parameter T can be used to tune the signaling load $S(t)$ and the (normalized) bandwidth inefficiency $BI'(t)$, while the parameter ϵ can be used to tune the blocking probability P_{bl} and the (normalized) bandwidth inefficiency $BI'(t)$. A cost function has been designed to compute the optimum value for T and ϵ. Performance experiments have been used to measure the performance of our aggregate update policy and to verify the relationship between the parameters (T and ϵ) and the constraints ($S(t)$, $BI'(t)$ and P_{bl}). The following can be concluded:

- The parameter T can be used to balance $S(t)$ and $BI'(t)$. From our results we can conclude that it is a good parameter for the tuning of these two constraints. Setting T to a low value results in a performance similar to using the IntServ architecture, in which $S(t)$ is high and $BI'(t)$ is low.

High values of T generate results similar to those found in networks using the DiffServ architecture, where $S(t)$ is low but $BI'(t)$ is high.

The parameter ϵ, was used to balance $BI'(t)$ and P_{bl}. Our experiments show that it is difficult to use this parameter to tune these constraints effectively.

- The cost function which computes the optimum value for T (and ϵ) does not give satisfying results with the weights $\kappa 1, \kappa 2$ and $\kappa 3$ set to their default values. In this case no optimum value for T could be computed. The measured results for $S(t)$ and $BI'(t)$ do indicate that there should be some optimum value for T. For increasing T, $S(t)$ drops exponentially while $BI'(t)$ increases linearly. In other words, sacrificing a little bandwidth results in a large gain for the signaling load. The main reason why no optimum value could be found, is because the cost function is largely influenced by $BI'(t)$. If we want to compute a proper cost then the weights $\kappa 1, \kappa 2$ and $\kappa 3$ should be set to different values, in which the weight for the $BI'(t)$ is decreased.

7.2 Future Work

There is still much work to be done on the prototype implementation, therefore the following we will recommended:

- The design of the prototype implementation was based on specifications which are still in development stage. As a result many specifications used in this thesis, namely the QoS NSLP draft [18] and the RMD-QoSM specification [1] have been superseded by newer versions. Updating the prototype implementation to comply with these new specifications is highly recommended.
- The NSLP statemachine designed in Chapter 5 does not include all the functionality provided by the NSIS framework. More specifically the support for receiver initiated reservation has no been added to our design. The prototype will be greatly enhanced if this feature is added.
- The RMD-QoSM specification implemented in this assignment only supports the aggregate of end-to-end flows. The current RMD-QoSM specification [1] also supports the tunneling of end-to-end reservations, receiver initiated reservations and the aggregation of flow using a measurement based scheme. Some of these features have been implemented in Martijn Swanink [35] a separate implementation. We advise this code be ported to the current prototype implementation.

With respect to our aggregate update policy we recommend conducting research on the following issues:

- For the optimization problem defined in Section 1.4.1 we have defined a cost function which can be used to compute the optimum values for T (and ϵ). The results from our cost function are not very satisfying. Some work still remains to be done on the calibration of the weights $\kappa 1, \kappa 2$ and $\kappa 3$. As an alternative one could define an entirely different cost function.
- In our performance experiments we have only examined the effects of T and ϵ on our aggregate update policy for constant bit rate flows with fixed bandwidth. It should be interesting to see how our aggregate update policy performs when heterogenous flows are used.

Bibliography

[1] G. Karagiannis C. Kappler A. Bader, L. Westberg and T. Phelan. RMD-QOSM - The Resource Management in Diffserv QOS Model, February 2007.

[2] F. Baker, C. Iturralde, F. Le Faucheur, and B. Davie. Aggregation of RSVP for IPv4 and IPv6 Reservations. RFC 3175 (Proposed Standard), September 2001.

[3] Y. Bernet, P. Ford, R. Yavatkar, F. Baker, L. Zhang, M. Speer, R. Braden, B. Davie, J. Wroclawski, and E. Felstaine. A Framework for Integrated Services Operation over Diffserv Networks. RFC 2998 (Informational), November 2000.

[4] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Service. RFC 2475 (Informational), December 1998. Updated by RFC 3260.

[5] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. RFC 1633 (Informational), June 1994.

[6] 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. Updated by RFCs 2750, 3936, 4495.

[7] K. Carlberg, I. Brown, and C. Beard. Framework for Supporting Emergency Telecommunications Service (ETS) in IP Telephony. RFC 4190 (Informational), November 2005.

[8] B. Davie, A. Charny, J.C.R. Bennet, K. Benson, J.Y. Le Boudec, W. Courtney, S. Davari, V. Firoiu, and D. Stiliadis. An Expedited Forwarding PHB (Per-Hop Behavior). RFC 3246 (Proposed Standard), March 2002.

[9] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999. Obsoleted by RFC 4346, updated by RFC 3546.

[10] R. Elz and R. Bush. Serial Number Arithmetic. RFC 1982 (Proposed Standard), August 1996.

[11] F. Le Faucheur. Protocol Extensions for Support of Diffserv-aware MPLS Traffic Engineering. RFC 4124 (Proposed Standard), June 2005.

[12] F. Le Faucheur and W. Lai. Requirements for Support of Differentiated Services-aware MPLS Traffic Engineering. RFC 3564 (Informational), July 2003.

[13] A. Bader G. Ash and C. Kappler. QoS NSLP QSPEC Template.

[14] M. Dolly P. Tarapore C. Dvorak G. Ash, A. Morton and Y. El Mghazli. Y.1541-QOSM – Y.1541 QoS Model for Networks Using Y.1541 QoS Classes.

[15] I. Okumus H. Mantar, J. Hwang and S. Chapin. Edge-to-Edge Resource Provisioning and Admission Control in Diffserv Networks. *IEEE 2001 International Conference on Software, Telecommunications, and Computer Networks*, 2001.

[16] R. Hancock, G. Karagiannis, J. Loughney, and S. Van den Bosch. Next Steps in Signaling (NSIS): Framework. RFC 4080 (Informational), June 2005.

[17] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski. Assured Forwarding PHB Group. RFC 2597 (Proposed Standard), June 1999. Updated by RFC 3260.

[18] A. McDonald J. Manner, G. Karagiannis. NSLP for Quality-of-Service Signaling, 2007.

[19] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM '88*, pages 314–329, Stanford, CA, August 1988.

[20] D. Katz. IP Router Alert Option. RFC 2113 (Proposed Standard), February 1997.

[21] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), March 2006.

[22] H. Mantar, J. Hwang, S. Chapin, and I. Okumus. A Scalable Model for Inter-Bandwidth Broker Resource Reservation and Provisioning, December 2004.

[23] H. Mantar, I. Okumus, J. Hwang, and S. Chapin. A scalable intra-domain resource management architecture for DiffServ networks.

[24] M. Menth. A Scalable Protocol Architecture for End-to-End Signaling and Resource Reservation in IP Networks. Technical Report 278, University of Wurzburg, Departmenet of Computer Science, juli 2001.

[25] K. Nichols, V. Jacobson, and L. Zhang. A Two-bit Differentiated Services Architecture for the Internet. RFC 2638 (Informational), July 1999.

[26] The Internet2 [Online]. http://www.internet2.edu/.

[27] The Internet2 QoS Working Group [Online]. http://qos.internet2.edu/, 2007.

[28] E. Hahne P. Pan and H. Schulzrinne. BGRP: A Tree-Based Aggregation Protocol for Inter-domain Reservations. *Journal of Communications and Networks*, 2000.

[29] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[30] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168.

[31] R.Klaver. Using NSIS (Next Steps in Signaling) for support of QoS aware multimedia services, February 2007.

[32] J. Schmitt, O. Heckmann, M. Karsten, and R. Steinmetz. Decoupling different time scales of network qos systems, July 2002.

[33] H. Schulzrinne and R. Hancock. GIST: General Internet Signalling Transport.

[34] 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. Updated by RFC 3309.

[35] M. Swanink. RMD (Resource Management in DiffServ) within NSIS (Next Steps in Signaling): protocol implementation, 2006.

[36] F. Wang and P. Mohapatra et al. An Efficient Bandwidth Management Scheme for Real-Time Internet Applications, 2002.

[37] H. Tschofenig T. Tsenov X. Fu, B. Schloer. QoS NSLP State Machine, 2007.

Appendix

A GIST API Service Primitives

The GIST API has the following service primitives, which are discussed in section 3.3.1:

- *SendMessage*
- *SetStateLifeTime*
- *InvalidRoutingState*
- *MessageStatus*
- *NetworkNotification*

The detailed description of the primitives below was taken from [33].

A.1 SendMessage

This primitive is passed from a signaling application to GIST. It is used whenever the signaling application wants to initiate sending a message.

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

The following arguments are mandatory:

NSLP-Data: The NSLP message itself.

NSLP-Data-Size: The length of NSLP-Data.

NSLP-Message-Handle: A handle for this message, that can be used by GIST as a reference in subsequent MessageStatus notifications. Notifications could be about error conditions or about the security attributes that will be used for the message. A NULL handle may be supplied if the NSLP is not interested in such notifications.

NSLPID: An identifier indicating which NSLP this is.

Session-ID: The NSIS session identifier. Note that it is assumed that the signaling application provides this to GIST rather than GIST providing a value itself.

MRI: Message routing information for use by GIST in determining the correct next GIST hop for this message. The MRI implies the message routing method to be used and the message direction.

The following arguments are optional:

SII-Handle: A handle, previously supplied by GIST, to a data structure that should be used to route the message explicitly to a particular GIST next hop.

Transfer-Attributes: Attributes defining how the message should be handled. The following attributes can be considered:

- *Reliability*: Values 'unreliable' or 'reliable'.
- *Security*: This attribute allows the NSLP to specify what level of security protection is requested for the message (such as 'integrity' or 'confidentiality'), and can also be used to specify what authenticated signaling source and destination identities should be used to send the message. The possibilities can be learned by the signaling application from prior MessageStatus or RecvMessage notifications. If an NSLP- Message-Handle is provided, GIST will inform the signaling application of what values it has actually chosen for this attribute via a MessageStatus callback. This might take place either synchronously (where GIST is selecting from available messaging associations), or asynchronously (when a new messaging association needs to be created).
- *Local Processing*: This attribute contains hints from the signaling application about what local policy should be applied to the message; in particular, its transmission priority relative to other messages, or whether GIST should attempt to set up or maintain forward routing state.

Timeout: Length of time GIST should attempt to send this message before indicating an error.

IP-TTL: The value of the IP layer TTL that should be used when sending this message (may be overridden by GIST for particular messages).

GIST-Hop-Count: The value for the hop count when sending the message.

A.2 RecvMessage

This primitive is passed from GIST to a signaling application. It is used whenever GIST receives a message from the network, including the case of null messages (zero length NSLP payload), typically initial Query messages. For Queries, the results of invoking this primitive are used by GIST to check whether message routing state should be created (see the discussion of the 'Routing-State-Check' argument below).

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)

The arguments are defined as follows:

NSLP-Data: The NSLP message itself (may be empty).

NSLP-Data-Size: The length of NSLP-Data (may be zero).

NSLPID: An identifier indicating which NSLP this message is for.

Session-ID: The NSIS session identifier.

MRI: Message routing information that was used by GIST in forwarding this message. Implicitly defines the message routing method that was used and the direction of the message relative to the MRI.

Routing-State-Check: This boolean is True if GIST is checking with the signaling application to see if routing state should be created with the peer or the message should be forwarded further. If True, the signaling application should return the following values via the RecvMessage call:

- A boolean indicating whether to set up the state.
- Optionally, an NSLP-Payload to carry in the generated Response or forwarded Query respectively.

This mechanism could be extended to enable the signaling application to indicate to GIST whether state installation should be immediate or deferred.

SII-Handle: A handle to a data structure, identifying a peer address and interface. Can be used to identify route changes and for explicit routing to a particular GIST next hop.

Transfer-Attributes: The reliability and security attributes that were associated with the reception of this particular message. As well as the attributes associated with SendMessage, GIST may indicate the level of verification of the addresses in the MRI. Three attributes can be indicated:

- Whether the signaling source address is one of the flow endpoints (i.e. whether this is the first or last GIST hop);
- Whether the signaling source address has been validated by a return routability check.
- Whether the message was explicitly routed (and so has not been validated by GIST as delivered consistently with local routing state).

IP-TTL: The value of the IP layer TTL this message was received with (if available).

IP-Distance: The number of IP hops from the peer signaling node which sent this message along the path, or 0 if this information is not available.

GIST-Hop-Count: The value of the hop count the message was received with, after being decremented in the GIST receive-side processing.

Inbound-Interface: Attributes of the interface on which the message was received, such as whether it lies on the internal or external side of a NAT. These attributes have only local significance and are implementation defined.

A.3 MessageStatus

This primitive is passed from GIST to a signaling application. It is used to notify the signaling application that a message that it requested to be sent could not be dispatched, or to inform the signaling application about the transfer attributes that have been selected for the message (specifically, security attributes). The signaling application can respond to this message with a return code to abort the sending of the message if the attributes are not acceptable.

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

The arguments are defined as follows:

NSLP-Message-Handle: A handle for the message provided by the signaling application in SendMessage.

Transfer-Attributes: The reliability and security attributes that will be used to transmit this particular message.

Error-Type: Indicates the type of error that occurred. For example, 'no next node found'.

A.4 NetworkNotification

This primitive is passed from GIST to a signaling application. It indicates that a network event of possible interest to the signaling application occurred.

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

The arguments are defined as follows:

NSLPID: An identifier indicating which NSLP this is message is for.

MRI: Provides the message routing information to which the network notification applies.

Network-Notification-Type: Indicates the type of event that caused the notification and associated additional data. Five events have been identified:

- *Last Node*: GIST has detected that this is the last NSLP-aware node in the path.
- *Routing Status Change*: GIST has installed new routing state, has detected that existing routing state may no longer be valid, or has re-established existing routing state. The new status is reported; if the status is Good, the SII- Handle of the peer is also reported, as for *RecvMessage*.
- *Route Deletion*: GIST has determined that an old route is now definitely invalid, e.g. that flows are definitely not using it. The SII-Handle of the peer is also reported.
- *Node Authorisation Change*: The authorisation status of a peer has changed, meaning that routing state is no longer valid or that a signaling peer is no longer reachable.
- *Communication Failure*: Communication with the peer has failed; messages may have been lost.

A.5 SetStateLifetime

This primitive is passed from a signaling application to GIST. It indicates the duration for which the signaling application would like GIST to retain its routing state. It can also give a hint that the signaling application is no longer interested in the state.

SetStateLifetime (NSLPID, MRI, SID, State-Lifetime)

The arguments are defined as follows:

NSLPID: Provides the NSLPID to which the routing state lifetime applies.

MRI: Provides the message routing information to which the routing state lifetime applies; includes the direction (in the D flag).

SID: The session ID which the signaling application will be using with this routing state. Can be wildcarded.

State-Lifetime: Indicates the lifetime for which the signaling application wishes GIST to retain its routing state (may be zero, indicating that the signaling application has no further interest in the GIST state).

A.6 InvalidateRoutingState

This primitive is passed from a signaling application to GIST. It indicates that the signaling application has knowledge that the next signaling hop known to GIST may no longer be valid, either because of changes in the network routing or the processing capabilities of signaling application nodes.

InvalidateRoutingState (NSLPID, MRI, Status, NSLP-Data, NSLP-Data-Size, Urgent)

The arguments are defined as follows:

NSLPID: The NSLP originating the message. May be null (in which case the invalidation applies to all signaling applications).

MRI: The flow for which routing state should be invalidated; includes the direction of the change (in the D flag).

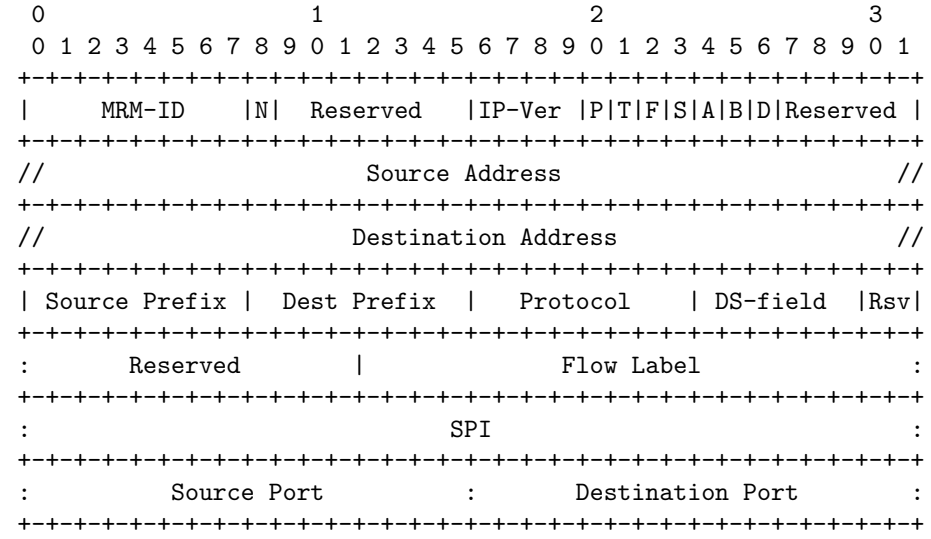
Status: The new status that should be assumed for the routing state, one of Bad or Tentative.

Urgent: A hint as to whether rediscovery should take place immediately, or only with the next signaling message.

The following arguments are optional:

NSLP-Data, NSLP-Data-Size : a payload provided by the NSLP to be used the next GIST handshake. This can be used as part of a conditional peering process. The payload will be transmitted without security protection.

B Path-coupled Message Routing Method



MRM-ID (8 bits): An IANA-assigned identifier for the message routing method.

N flag: If set (N=1), this means that NATs do not need to translate this MRM; if clear (N=0) it means that the method-specific information contains network or transport layer information that a NAT must process.

In the case of basic path-coupled routing, the addressing information takes the following format. The N-flag N=0 for this MRM. *IP-Ver (4 bits)*: The IP version number, 4 or 6.

Source/Destination address (variable): The source and destination addresses are always present and of the same type; their length depends on the value in the IP-Ver field.

Source/Dest Prefix (each 8 bits): The length of the mask to be applied to the source and destination addresses for address wildcarding. In the normal case where the MRI refers only to traffic between specific host addresses, the Source/Dest Prefix values would both be 32/128 for IPv4/6 respectively.

P flag: P=1 means that the Protocol field is significant.

Protocol (8 bits): The IP protocol number. This MUST be ignored if P=0. In the case of IPv6, the Protocol field refers to the true upper layer protocol carried by the packets, i.e. excluding any IP option headers. This is therefore not necessarily the same as the Next Header value from the base IPv6 header.

T flag: T=1 means that the DiffServ field (DS-field) is significant.

DS-field (6 bits): The DiffServ field.

F flag: F=1 means that flow label is present and is significant. F MUST NOT be set if IP-Ver is not 6.

Flow Label (20 bits): The flow label; only present if F=1. If F=0, the entire 32 bit word containing the Flow Label is absent.

S flag: S=1 means that the SPI field is present and is significant. The S flag MUST be 0 if the P flag is 0.

SPI field (32 bits): The SPI field. If S=0, the entire 32 bit word containing the SPI is absent.

A/B flags: These can only be set if P=1. If either is set, the port fields are also present. If P=0, the A/B flags MUST both be zero and the word containing the port numbers is absent.

Source/Destination Port (each 16 bits): If either of A (source), B (destination) is set the word containing the port numbers is included in the object. However, the contents of each field is only significant if the corresponding flag is set; otherwise, the contents of the field is regarded as padding, and the MRI refers to all ports (i.e. acts as a wildcard). If the flag is set and Port=0x0000, the MRI will apply to a specific port, whose value is not yet known. If neither of A or B is set, the word is absent.

D flag: The Direction flag has the following meaning: the value 0 means 'in the same direction as the flow' (i.e. downstream), and the value 1 means 'in the opposite direction to the flow' (i.e. upstream).

C Sample RESERVE message

```
QoS NSLP Reserve Message (complete):
Common Header:
        Message Type: Reserve
        B: 0
        A: 0
        P: 0
        S: 0
        T: 0
        R: 1
TLV header
        Extensibility: Mandatory
        Type: RSN
        Length: 2
RSN Object:
        RSN: 2831123241
        Epoch Identifier: 638999738
TLV header
        Extensibility: Mandatory
        Type: RII
        Length: 1
RII Object:
        RII: 3141525279
```

```
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: 45.000000 bytes/s
                Token Bucket Size: 32768.000000 bytes
                Peak Data Rate: inf bytes/s
                Minimum Policed Unit: 1 bytes
                Maximum Pakcet Size: 1500 bytes
```

D Results Performance Experiments

Due to time constraints the statistical accuracy of the results, e.g.e, computation of the confidence intervals, has not been verified.

D.1 Results for constant epsilon ($\epsilon = 0.005$)

$T =$	86400	43200	21600	10800	7200	3600
$S(t)$	$3,80084 * 10^{-5}$	$7,60167 * 10^{-5}$	0,000114969	0,000152033	0,000342075	0,000494109
$BI'(t)$	0,712883966	0,70118557	0,684638955	0,682125789	0,651755447	0,598235848
P_{bl}	0	0,08451487	0,032817924	0	0	

Table 1: Measured data for an ϵ of 0.005 and $T = [86400, 3600]$.

$T =$	1800	1200	600	300	120	60
$S(t)$	0,000342075	0,000988293	0,001560122	0,003237109	0,020029217	0,040095465
$BI'(t)$	0,593758349	0,481646803	0,472040221	0,376155504	0,267736081	0,168793524
P_{bl}	0	0,000354233	0,004837595	0,007757167	0,034871526	0,083278581

Table 2: Measured data for an ϵ of 0.005 and $T = [1800, 60]$.

D.2 Results for constant inter update target ($T = 300$)

$T =$	0,25	0,1	0,05	0,025	0,01	0,005	0,001
$S(t)$	0,003351104	0,003964926	0,003274195	0,003044835	0,003275442	0,003237109	0,003273945
$BI'(t)$	0,439525952	0,34721359	0,360726916	0,357747298	0,419983805	0,376155504	0,395175377
P_{bl}	0,007505971	0,011035378	0,005693235	0,005541069	0,006976744	0,007757167	0,005840363

Table 3: Measured data for an T of 300.