Faculty of Electrical Engineering, Mathematics and Computer Science
Network Architectures and Services

Implementing Link-state Update Policies for Quality of Service Routing

M. Noordermeer
(1178318)

Committee members:

Chair:	prof. dr. ir. P.F.A. Van Mieghem
Supervisor:	dr. C. Doerr
Member:	dr. ir. G.N. Gaydadjiev

November 2, 2011
M.Sc. Thesis No: PVM 2011–72

Implementing Link-state Update Policies for Quality of Service Routing

Master's Thesis in Computer Engineering

Network Architectures and Services group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Mike Noordermeer

November 2, 2011

Author
Mike Noordermeer

Title
Implementing Link-state Update Policies for Quality of Service Routing

MSc presentation
November 10, 2011

Graduation Committee

prof. dr. ir. P.F.A. Van Mieghem (chair)	Delft University of Technology
dr. C. Doerr	Delft University of Technology
dr. ir. G.N. Gaydadjiev	Delft University of Technology

Abstract

This thesis describes the implementation of available bandwidth link-state update policies for Quality of Service routing. Periodic, threshold-based and class-based policies are described and implemented in a QoS router testbed, using Quagga and its OSPF-API for low-level communications. The implementation is thoroughly tested using unit tests and simulated traffic. A performance comparison of the link-state update policies is done, using two different scenarios. The results of this comparison show that determining the right parameters for the policy is more important than the kind of policy used. The performance of the policies is generally worse than in previous work, due to the absence of explicit flow admission control, which makes it impossible to account for traffic until after it has been sent. The results also indicate that the use of a moving average instead of a hold-down timer leads to less link-state updates, while not impacting performance.

Preface

Before you is my Master of Science thesis, which describes the work I conducted over the last years at the Network Architectures and Services group of the Faculty EEMCS, Delft University of Technology. Computer networks, the Internet in particular, have always intrigued me. It is fascinating to see how modern businesses cannot exist without these networks, and how people from all over the world connect through a variety of applications. For this reason, I chose to "switch" from Computer Science to Computer Engineering after my Bachelor's degree, so I could perform my Master's project at the Telecommunications department. Being a software minded person, my thesis assignment, which for a large part consisted of implementing a network protocol in a software router, was a perfect fit. My only doubts were with the "QoS part" of the assignment. I was never convinced of the future of Quality of Service in networks. The need for QoS is clear, but the implementation is so complex I doubted QoS would ever be implemented in a network as large as the Internet. Looking back at my thesis, this skepticism is still present, and I am interested in what the future will bring to the Internet.

I would like to thank some people who have been of great help during my project. First of all, I would like to thank my mentor Tom Kleiberg for his help, insightful comments and support. Unfortunately, he left the faculty to pursue his career elsewhere before I could finish my thesis, and I want to wish him the best of luck with that. Christian Doerr took over the supervision, and I want to thank him for that. Also, I would like to thank the graduation committee members for their comments and remarks. My last thanks go out to my family and friends, for their support and patience, even when it took somewhat longer than planned to finish this thesis.

Mike Noordermeer

Delft, The Netherlands
November 2, 2011

Contents

Preface **v**

1 Introduction **1**
1.1 Problem definition 3
1.2 Outline 4

2 QoS measures and link-state update policies **5**
2.1 QoS measures 5
2.1.1 Bandwidth 6
2.1.2 Delay 6
2.1.3 Packet delay variation 7
2.1.4 Packet loss 8
2.2 Link-state update policies 8
2.2.1 Periodic policy 9
2.2.2 Threshold-based policies 9
2.2.3 Class-based policies 11
2.2.4 Hold-down timer 12
2.2.5 Moving average 12

3 Implementation of a link-state update protocol **15**
3.1 Link-state update protocols 16
3.1.1 OSPF 17
3.1.2 QOSPF 17
3.1.3 OSPF-TE 18
3.1.4 OSPF-xTE 20
3.1.5 Other protocols 20
3.2 Our own link-state update protocol 21
3.3 Implementation 22
3.3.1 Link-state update library 23
3.3.2 Monitoring and update daemon 29
3.3.3 QoS router testbed integration 29

4 Verification of the software **35**
4.1 Requirements and performance criteria 35
4.2 Unit testing 38
4.3 Test scenarios 39
4.4 Results 41

5 Comparison of link-state update policies **45**
5.1 Performance metrics 45
5.2 Test setup 47
5.3 Test scenarios 48
5.4 Results 51
5.4.1 10 Mbit/s scenario 51
5.4.2 100 Mbit/s scenario 56

6 Discussion **61**
6.1 Summary of performance comparison 61
6.2 Discussion of the results 62
6.2.1 High update error 62
6.2.2 Lower check interval increases update error and rate 63
6.2.3 Sending more updates does not lead to better performance 64
6.2.4 No performance difference between policies 65
6.2.5 Lower average relative error for 100 Mbit/s scenario 65
6.2.6 Use of a moving average decreases update rate with same performance 66

7 Conclusions and Future Work **67**
7.1 Conclusions 67
7.2 Future Work 68

Bibliography **71**

1 Introduction

Quality of Service (QoS) is becoming increasingly important in current networks. While the Internet was originally built for the exchange of simple text messages with no explicit performance requirements, the current use of multimedia and real-time applications place strict requirements on network performance. Each application has its own set of requirements; for example, voice over IP applications expect low latency, whereas multiplayer games are especially sensitive to packet loss. The QoS forum, which was actively researching QoS solutions for the Internet, used the following definition of QoS: *"Quality of Service is the ability of a network element to have some level of assurance that its traffic and service requirements can be satisfied."* In the current Internet, explicit QoS management is almost non-existent. Providers try to keep their service on a high level, mostly through over-provisioning, but there is no solution that can offer hard guarantees for a connection or let applications specify their QoS demands. Critics even claim that QoS in the Internet is an utopian idea, and will never be realized [27].

However, over the last decades, several standards have been proposed and implemented to provide QoS in a network. Differentiated Services, or DiffServ, described in RFC 2474 [30], is one of the most common standards for providing QoS over IP networks. It is a coarse-grained solution, with several classes to classify traffic. An application can choose, from low to high priority, for the default behavior, assured forwarding or expedited forwarding. The priority is communicated in the DS field of the IP header. This way, realtime traffic can be

given priority over bulk traffic, without a lot of administrative overhead. Usually ingress routers set the DS field. The information in the DS field is then used by backbone routers to determine the priority of a packet. Because DiffServ is based on per packet, per hop behavior, routers do not need to keep track of traffic flows and no path setup protocol is necessary. However, DiffServ is unable to provide any strict QoS guarantees: if too much traffic arrives, it will be dropped anyway, even if it has a high priority. The current trend of network convergence, where voice and data networks are combined into one IP network, requires networks to be able to prioritize voice traffic. For this reason, many core networks deploy QoS by using DiffServ (or MPLS priorities, which are similar [17]). Also, company networks provide DiffServ support in order to give priority to voice traffic.

Integrated Services, or IntServ, described in RFC 1633 [12], is the fine-grained counterpart of DiffServ. Each flow is described using a traffic specification and a request specification. The traffic specification describes the bandwidth requirements of the traffic. The request specification describes whether the traffic should be handled on a best effort, controlled load or guaranteed basis. Each flow has to be setup using a signaling protocol; the Resource Reservation Protocol (RSVP) [13] is normally used for this purpose. After setup of a flow every router has to keep track of the flow and handle it in the way specified in the traffic and request specification. The advantage of IntServ over DiffServ is that once the flow has been setup, the application can be sure the bandwidth will be available. This is because each router accepted the flow during the signalling phase and has reserved resources for it. But because every router has to keep track of each flow, IntServ does not scale well, and is seldom used in practice.

Multiprotocol Label Switching (MPLS) [33] is a label switching protocol, where traffic labeled with a certain label is sent over a predefined label-switched path (LSP). MPLS itself implements eight priority levels, which can be used to provide QoS in a manner similar to DiffServ (but over predefined paths). RSVP - Traffic Engineering (RSVP-TE) [8], an extension to RSVP, and the now deprecated CR-LDP [21], both provide a mechanism to dynamically setup LSPs, adhering to certain QoS constraints. This setup has the same disadvantage as IntServ: each router has to keep track of every flow. MPLS priority levels are often used by providers to offer customers prioritized treatment over other customers in exchange for a higher fee. The problem with this approach is twofold: providers can only guarantee the QoS within their own autonomous system, and different applications of the same customer are unable to specify a different QoS.

Strict QoS provisioning imposes several requirements on the path between the end-nodes. First, it is necessary that ample capacity can be reserved in order to meet the QoS requirement. Second, a path selection algorithm is necessary.

IntServ and RSVP generally still use the shortest path available, which might not provide the desired QoS. RSVP-TE and CR-LDP provide the opportunity to setup explicit paths, but do not specify the routing algorithm to use.

To provide full QoS capabilities, routing algorithms have been developed that take into account both the layout and the current state of the links in the network, like SAMCRA [38]. Research over the last years has been focussed on developing and testing these algorithms. In order to test the algorithms, testbeds have been developed that make it possible to simulate traffic loads and test the behavior of the QoS routing algorithms [7].

As said, most QoS routing algorithms need the current layout of the network as well as the current state of the links in the network to operate. Distributing the current network layout throughout the network is a problem that has been solved a long time ago by link-state routing algorithms (e.g., OSPF or IS-IS). Since this information is relatively static, all updates on network layout can be distributed without performance issues. Distributing the exact link-state information, like the current link delay, available bandwidth and amount of packet loss, has proven to be a bigger problem. Because this information is so dynamic by nature, distributing every available update would lead to excessive updates and bad network performance. In the literature, various policies to distribute link-state updates have been proposed. Most of these policies have been examined theoretically and in network simulators, but none of these policies have been implemented in real-life networks, which brings us to our problem definition.

1.1 Problem definition

The Delft University of Technology employs a QoS routing testbed based on the work of Avallone [7]. This testbed supports the use of various QoS routing algorithms and has the ability to simulate traffics flows over a network. At this moment, the testbed uses a centralized architecture, where a central component (the Network Controller) has full knowledge of network state. For this reason, no link-state update policies have been implemented into the network. In order to support a more distributed routing setup, and to be able to compare the performance of available link-state update policies, it is desirable to implement support for link-state update policies in this network.

In this thesis project we will implement link-state update policies in the QoS routing testbed. Also we will do a performance comparison between the available link-state updates policies to see which policy performs best. We will achieve these goals by completing the following subgoals:

- Study existing literature on update policies and identify existing link-state update policies.
- Study what network protocols are available to exchange link-state information. Based on this information choose (or adapt) one of these protocols to form the basis for our own implementation.
- Implement the network protocol to exchange link-state information in the QoS router testbed.
- Implement the link-state update policies found in literature on top of the developed network protocol.
- Verify the correctness of the network protocol and link-state update policy implementations.
- Do performance measurements on the various policies (e.g., which policy causes most update messages, which provides best accuracy).
- Integrate our implementation into the Network Controller of the testbed, to make the link-state information available to the QoS routing algorithms that are already available in the testbed.

1.2 Outline

The structure of this document is as follows. Chapter 2 discusses QoS measures and link-state update policies in more detail. In Chapter 3, network protocols available to exchange link-state information are discussed, and a software solution is implemented that makes it possible to exchange available bandwidth information through an OSPF network according to various policies. Verification of the correctness of the software is done in Chapter 4. Chapter 5 compares the performance of link-state update policies in several scenarios after which we discuss these results in Chapter 6. We then conclude this thesis and discuss some possibilities for future work in Chapter 7.

2

QoS measures and link-state update policies

As explained in the previous chapter, most QoS routing algorithms depend on accurate link-state information to provide good routes. The link-state information consists of one or more QoS measures and has to be broadcasted through the network to provide each QoS routing protocol instance with the necessary information. In this chapter we discuss the available QoS measures in more detail. We also discuss the link-state update policies available, which determine when link-state is exchanged between nodes.

2.1 QoS measures

The Quality of Service of a certain connection can be classified using various measures, e.g., bandwidth, delay, packet delay variation or packet loss. Depending on the needs of the application one or more of these measures must meet some requirements. The QoS of a certain path is determined by combining the measures of the individual links of the path. Some of the measures are additive (e.g., delay), while others are multiplicative (e.g., packet loss) or min-max measures (e.g., available or used bandwidth) [7]. The most important measures will be discussed here. Others exist (e.g., hop count or error rate), but those are either not important to the end-user application (hop count) or can be incorporated into the other measures (errors can be seen as packet loss, since the receiving

host will discard packets with errors).

2.1.1 Bandwidth

The available bandwidth on a path is one of the most important QoS measures available. Insufficient bandwidth for a certain application will result in increased delay, due to temporary queueing in intermediate hops, or packet loss, due to queues overflowing.

The bandwidth necessary for a connection is normally specified by the rate of the traffic (in bytes per second). Some standards, like IntServ, choose to add a bucket depth in order to specify the burstiness of the traffic. This is because some applications have a low average bandwidth requirement, but need to send traffic bursts from time to time. The typical example is telephony: during silences the application can save on bandwidth by not sending any packets, resulting in less but burstier traffic.

Bandwidth is a minimum QoS measure: the link with the lowest available bandwidth will be the bottleneck for the complete path.

2.1.2 Delay

Another important measure is end-to-end one-way delay, usually measured in milliseconds. Realtime applications like video conferencing, telephony or online gaming require a low delay in order to provide a good user experience. The ITU-T Recommendation G.114 [19] recommends an end-to-end one-way delay of less than 150 ms for interactive applications.

Delay consists of several components: propagation, serialization, queueing, and processing delay. The propagation delay is the time it takes for the signal to travel through the communication medium (e.g., fiber optic cable), and accounts for the largest part of the delay in inter-network communication. Depending on the distance between sender and receiver, and the medium used, this delay can be anywhere from 1 to 200 ms. A typical value for the propagation delay in fiber optic cable is 1 ms per 160 km [34].

The serialization delay is the time it takes to signal a packet to the medium. Today's fast communication channels have made this delay almost obsolete; a 1500 bytes packet is signalled in 12 µs when using a 1 Gbit/s link [34]. On slower links the serialization delay can be of more influence in the total delay. For instance, on a 1.544 Mbit/s DS-1 line it takes 7.7 ms to serialize a 1500 bytes packet.

Queueing delay is incurred when a router receives multiple packets and has to queue some of them while processing the other packets. When the buffer

is empty (i.e., the link is lightly loaded) or for high priority packets (which get queued in front of the other packets), the queueing delay is at most the serialization delay of one packet (the one that the router might be already transmitting). On heavily loaded links, queueing delay can become a great part of the total delay, depending on buffer size, link speed, packet priority and router configuration.

Processing delay is the time it takes for all hosts on the path to process the contents of the packet (e.g., checksum verification, routing decisions, packet creation and interpretation). The typical router only needs a fraction of a microsecond for this processing [34]. Depending on the application and protocols in use, the delay incurred for packet creation and interpretation on the sending and receiving host can vary wildly. Since the application can determine the packet creation and interpretation delay itself (by using different protocols or algorithms) we will not further consider this delay.

Looking at the various delays, we see processing and serialization delays are generally small and cannot be influenced by the network routing protocols. Queueing and propagation delay are the important delay components to consider when talking about QoS. On heavily loaded networks, where queueing delays are high, certain packets should be given priority to provide the right QoS level. On long-distance, international connections, propagation delays may be high. Also, multiple paths may be available, each with their own propagation delay. A QoS routing algorithm can choose between these paths, to provide each flow with a delay that is within their QoS requirements.

Delay is an additive QoS measure: the total delay of a path is the delay of the individual propagation, serialization, queueing and processing delay components combined.

2.1.3 Packet delay variation

Packet delay variation (PDV), sometimes also called jitter, is another QoS measure of importance to certain application. Specified in more detail in RFC 3393 [15], PDV concerns the difference in delay between arriving packets of the same flow. This difference can be measured in several ways, like averaging the absolute value of all differences together, or by determining the variance. A high PDV may also cause packets to be received out of order.

Most delay sensitive applications also depend on a low packet delay variation. Other applications, like video streaming, are able to cope with PDV by using their buffers.

Depending on the unit and measurement method chosen, the packet delay variation can be an additive QoS measure (ie., when a proper random variable

describing the PDV can be constructed). But sometimes it is harder to determine the PDV of a path from the PDV of the individual links. For instance, when the absolute value of the variations is used, it is non-trivial to determine the PDV of the path from its individual links.

2.1.4 Packet loss

Packet loss considers packets that get lost during transmission. This can occur for various reasons, like congestion, packet corruption or signal degradation. Packet loss is measured as the percentage of packets that get lost.

When packet loss is experienced, application performance suffers. TCP connections will have to retransmit packets, which causes a delay because the retransmission timer has to expire before a packet is retransmitted. UDP-based applications, like video, audio and gaming applications, will miss some of their data. The user will experience this as an interruption in the signal.

Packet loss is a multiplicative QoS measure: multiplying the percentage of packets that arrive on individual links gives the percentage of packets that arrive on the complete path. The measure can be converted to an additive QoS measure by using the logarithm of the values.

2.2 Link-state update policies

Link-state update policies (LSUPs) consider the distribution of link-state information (e.g., available bandwidth or delay) through the network. Most QoS routing algorithms base their routing decisions on the latest link-state information available. Up-to-date information is therefore important in order to make sound decisions. However, distributing link-state information through the network places a load on the network and routers. A good tradeoff has to be found between the frequency of the updates (staleness of the information) and the load on the network.

In literature, many link-state update policies have been proposed. Most of these policies concern themselves with the available bandwidth measure, because it is the most fluctuating metric. Other QoS measures are generally more stable over time, which makes the link-state update policy used less important. For this reason, we will focus ourselves on the available bandwidth measure. We will now discuss the policies for distributing available bandwidth information in more detail.

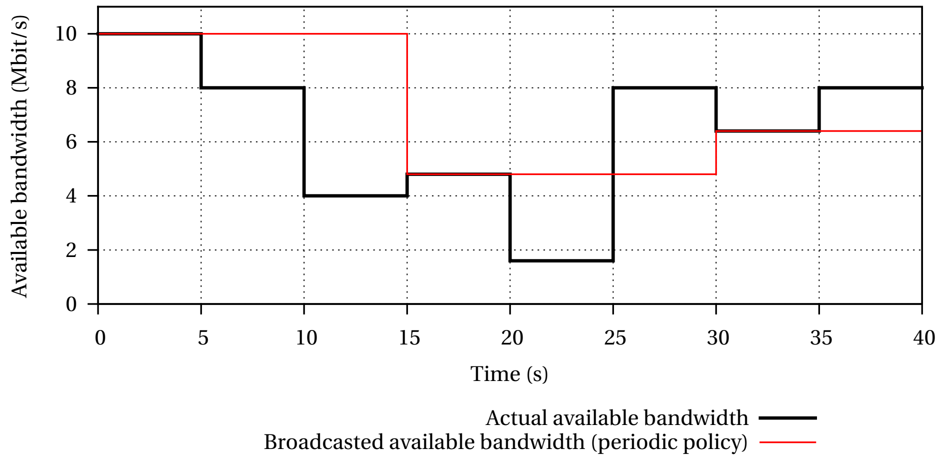


Figure 2.1 – Example of a periodic update policy with an interval of 15 seconds.

2.2.1 Periodic policy

When using a periodic policy, the link-state will be updated on a set interval. With this policy, the exact overhead caused by the updates is known beforehand. An important limitation of this policy is that network conditions might change in a much smaller interval than the update interval, which will cause stale information because a periodic policy is not adaptive to the traffic situation. Shaikh et al. have shown that a periodic policy can cause excessive route flapping and can perform worse than best effort routing [36]. Figure 2.1 gives an example of a periodic policy with an interval of 15 seconds. The black line indicates the actual available bandwidth, while the red line indicates the broadcasted available bandwidth value.

2.2.2 Threshold-based policies

Threshold-based policies update link-state when the available bandwidth changes by a predefined amount. This change can be either an absolute value, or a percentage of the last announced bandwidth. Figure 2.2 gives an example of threshold-based policies.

Absolute change

With an absolute change policy, the available bandwidth information is updated when the last announced value, B', differs from the current value, B, by a certain threshold, t_a. This leads to the following equation:

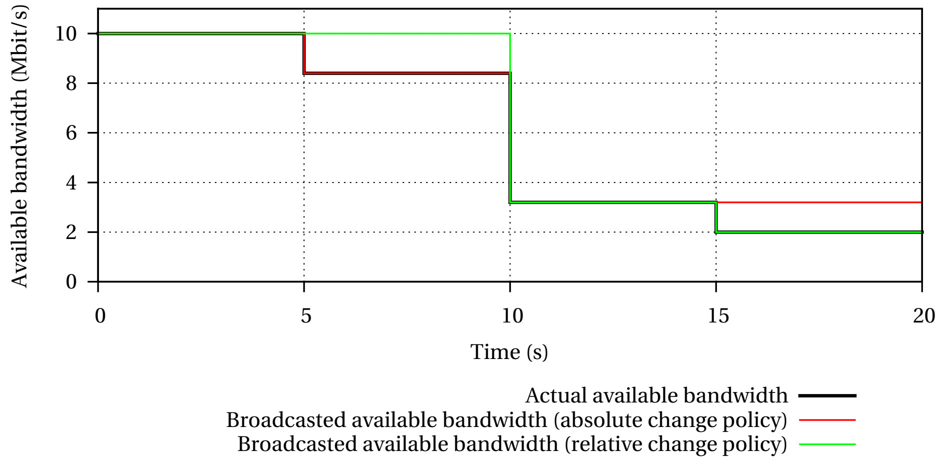


Figure 2.2 – Example of an absolute change policy with a required change of 1.5 Mbit/s and an relative change policy with a required change of 20%. Note that the relative change policy does not update on the initial change in available bandwidth (which is less than 20%), but does update on the later, smaller change in bandwidth (which is more than 20%).

$$t_a \leq |B - B'| \tag{2.1}$$

It is also possible to use a percentage, t_p, of the capacity, C, instead of an absolute value, yielding:

$$t_p \leq \frac{|B - B'|}{C} \tag{2.2}$$

Ariza et al. have shown that using an absolute change policy causes excessive updates and worse routing performance than other update policies [6]. Basu and Riecke concluded an absolute change policy leads to more rejects with the same processor utilization than more advanced link-state update algorithms [9].

Relative change

Instead of using the absolute change in available bandwidth, it is also possible to look at the relative change, t_r, compared with the previous update:

$$t_r \leq \frac{|B - B'|}{B'} \tag{2.3}$$

This has the advantage that the updates are send out more frequently when available bandwidth on a link gets low. Ariza et al., Basu and Riecke, and Shaik et

al. all have shown a relative change policy leads to less updates than a periodic policy and less blocking than periodic and absolute change policies [6, 9, 36].

2.2.3 Class-based policies

Class-based policies divide the bandwidth in classes and send out an update when the available bandwidth passes a class boundary. In order to limit the amount of updates when the available bandwidth fluctuates around a class boundary, Apostolopoulos et al. have proposed a *hysteresis* mechanism where an update is only send out when available bandwidth falls below the middle value of the new class [4]. They also introduced the idea of sending a *quantized* value in order to give better routing performance [3]. Another solution for the excessive update problem is to only send an update when the available bandwidth has passed more than one class boundary or to use a hold-down timer (see Section 2.2.4).

Equal-sized classes

An equal-sized classes policy divides the capacity, C, of a link in N_{eq} equal-sized classes:

$$\left[0, \frac{C}{N_{eq}}\right), \left[\frac{C}{N_{eq}}, \frac{2C}{N_{eq}}\right), \ldots, \left[\frac{(N_{eq}-1)C}{N_{eq}}, C\right) \tag{2.4}$$

Figure 2.3 gives an example of an equal-sized class-based update policy. Since all classes have an equal size, this policy looks a lot like an absolute change policy, and also has some of the same disadvantages. Yuan et al. have shown that an equal-sized classes policy nevertheless can be a useful policy since algorithms like safety-based routing [4] can take the fixed class size into account [39].

Exponential-sized classes

An exponential-sized class-based LSUP uses geometrically increasing class sizes. First, the size of the base class is determined using a base factor b ($b < 1$). Next, the class boundaries can be determined using a growth factor f ($f > 1$):

$$\left[0, Cb\right), \left[Cb, (1+f)Cb\right), \left[(1+f)Cb, (1+f+f^2)Cb\right), \ldots \tag{2.5}$$

Figure 2.4 gives an example of an exponential-sized class-based update policy. Due to the increasing class size the link-state information will be less precise when more bandwidth is available. This is comparable with the relative change policy. Apostolopoulous et al. have shown that an exponential-sized

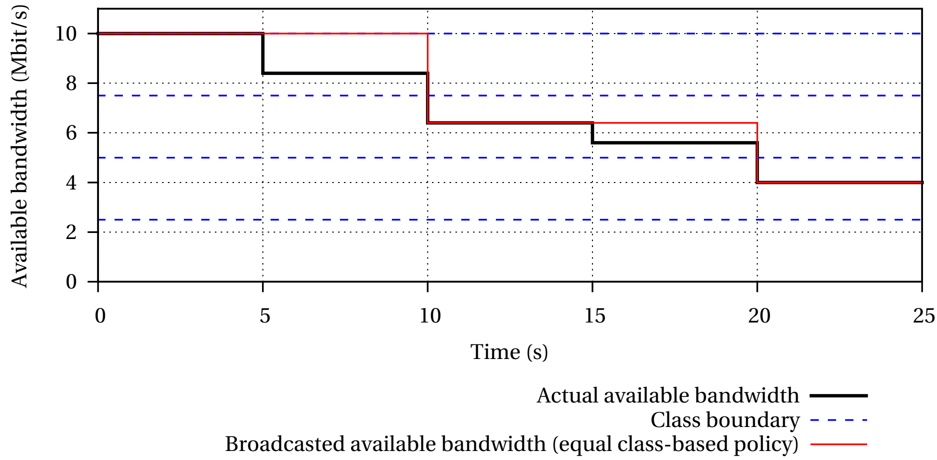


Figure 2.3 – Example of an equal-sized class-based update policy with $N_{eq} = 4$ on a 10 Mbit/s link.

class-based policy achieves roughly similar routing performance as a relative change policy [4].

2.2.4 Hold-down timer

In order to limit the number of updates and prevent excessive updates many people have suggested implementing a hold-down timer on link-state updates. Most routing protocols (e.g. BGP, OSPF, RIP) already implement a hold-down timer to prevent route flapping and network instability. The same approach can be used for LSUPs. When using a hold-down timer, after an update has been sent, no updates will be send out for a certain amount of time. Yuan and Zheng have shown that a hold-down timer introduces random imprecision in the data [39].

2.2.5 Moving average

Instead of a hold-down timer, Lekovic and Van Mieghem suggest to use a moving average [24]. The bandwidth utilization is smoothened by computing the mean of several successive values, and using that mean value in the various update policies. They show that using a moving average instead of a hold-down timer leads to less updates, smaller update error and less blocking.

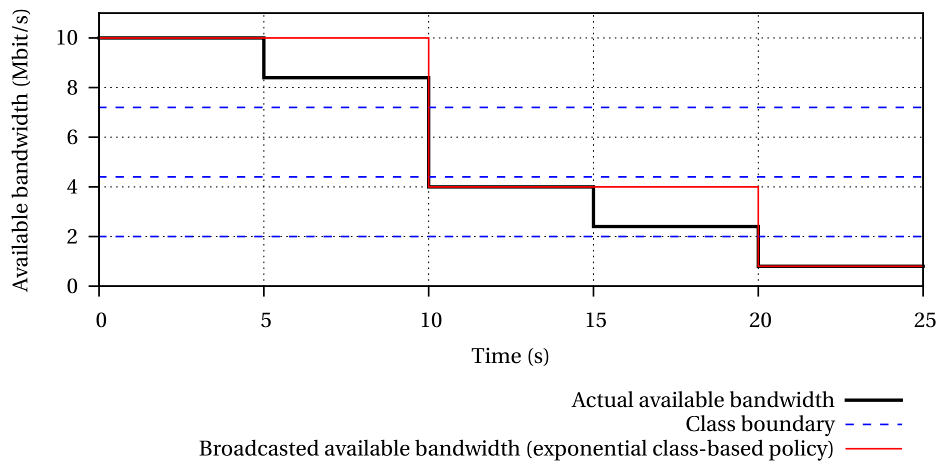


Figure 2.4 – Example of an exponential-sized class-based update policy with $b = 0.2$ and $f = 1.2$.

3

Implementation of a link-state update protocol

As we have seen in Chapter 2, various link-state update policies (LSUPs) have been suggested in the literature. Most of these policies have either been judged theoretically or using various simulation applications. Although some simulations use extensive setups, emulating complete routers including their links in software [5], real world experience is also necessary in order to compare the performance of different LSUPs.

The current QoS router testbed at Delft University of Technology does not have support for link-state updates. The testbed runs on Fedora Linux[1] and uses the Quagga[2] OSPF implementation to provide the network with interior routing. Also SNMP [14] services are available throughout the network. The testbed uses a central node, the Network Controller (NC), that possesses the complete state information of the network. In larger setups, this centralized approach might not scale well enough. Also, traffic that is not explicitly managed by the NC is currently invisible to the QoS routing process. For these reasons, and because one of our research goals is to compare link-state update policies, we will extend the current QoS router testbed with link-state updates. This implementation should give the user the option to select between LSUPs and choose the various parameters for the policies.

1 http://fedoraproject.org
2 http://www.quagga.net

For most QoS routing algorithms, complete link-state information on all the links in the network is necessary. But without communicating with other hosts, a host only knows the state of its own links, since measurements are done locally. Because of this, a communication protocol is necessary to distribute the link-state information throughout the network. To develop this protocol, we will compare the features, advantages and disadvantages of existing protocols. We will then implement one of these protocols in the testbed, or create a new protocol if the current protocols do not suffice.

This chapter will first discuss the various protocols available to exchange link-state information. Next, the protocol used for our implementation will be determined and documented. We then discuss the link-state update library that has been developed and the accompanying client application. We conclude by explaining the integration of the library with the existing QoS router testbed.

3.1 Link-state update protocols

In Section 2.2 different link-state update policies have been discussed. However, these policies alone are not enough to implement the exchange of link-state information in a network. In order to exchange and distribute the information throughout the network, a communication protocol has to be developed.

When it comes to monitoring link-state, both pull-based and push-based protocols are available. With pull-based protocols, like the Simple Network Management Protocol (SNMP) [14], clients can retrieve interface state information on demand. With push-based protocols, new information is pushed (i.e., flooded) to the clients when it is available.

An advantage of the pull-based approach is that no flooding or distribution protocol has to be developed, which saves development time. However, there are some disadvantages to pull-based protocols. First of all, pull-based protocols do not scale well in a distributed routing environment. The current QoS router testbed implementation has a central Network Controller that handles the QoS routing process. As long as this is the case, scalability will be fine because only this Network Controller needs to know the network state. However, for various reasons (e.g. availability, performance) a distributed routing setup can be desirable. In this case, the routers need to make QoS routing decisions themselves, and thus need state information. To retrieve the state information N nodes will have to poll $N-1$ nodes, leading to excessive network load.

An even more important disadvantage is that clients can only know the link-state after explicitly asking for an update. This conflicts with link-state update policies that are threshold or class-based. When using a pull-based protocol,

we can therefore only implement a periodic update policy. Since the main goal of our research is to implement and compare several different link-state update policies, pull-based monitoring is not a viable option for our approach.

For that reason, we will now focus ourselves on existing push-based protocols. The possibilities and limitations of these protocols will be discussed in more detail. In order to limit our search scope and guarantee a good integration in the existing testbed, we will keep the current QoS router testbed implementation and its capabilities in mind during our search.

3.1.1 OSPF

Open Shortest Path First (OSPF) [29] is an interior, link-state routing protocol for IPv4 networks. Since OSPF is already concerned with communicating the link-state updates to the routers in the network, it seems like a logical choice to add the QoS measures to these updates.

OSPF router link-state advertisements (LSAs) contain an entry for each OSPF-enabled interface of a router. Every interface defines a cost metric, specifying the administrative cost of the link. One of the possibilities for a link-state update protocol is to use the cost metric to communicate a QoS measure. However, this approach has some important disadvantages. First of all, only one QoS measure can be communicated since there is only one metric field. This means no multi-constraint QoS routing can be done. Also, when reusing the cost metric this way, we lose the existing (administrative) meaning of the cost metric. Non-QoS aware routers use the same metric, which might lead to strange routing behavior when QoS aware routers are combined with non-QoS aware routers in the same network. Because each OSPF router sends only one router LSA that contains all of its links in a certain area, a change in one of the link states will send out an update for all links. This causes unnecessary network traffic. For these reasons using the administrative cost metric of the OSPF protocol is not a good solution.

3.1.2 QOSPF

In the original OSPF proposal [28], each link also defined several Type of Service (TOS) metrics. The idea of these metrics was to be able to provide a different link cost for traffic carrying a certain value in the TOS field of the IP header. Each OSPF router would then run a separate shortest path calculation and keep a different routing table for each TOS value. Since RFC2474 [30] defined a Differentiated Services (DS) field in the Internet Protocol, replacing the TOS field, the TOS metrics were deprecated in the final OSPF standard, but the fields

are still present. Also, the TOS-bit, which indicates if a router supports TOS metrics, is still part of the OSPF Options field.

An option for a link-state update protocol would be to reuse the TOS fields in the OSPF protocol, since these are currently not used. The OSPF extension for QoS routing mechanisms (QOSPF), defined in RFC2676 [5], is an experimental proposal for reusing the TOS-bit as a QoS-bit. Also, the TOS metrics in the OSPF router LSAs are reused to communicate available bandwidth and link delay information.

QOSPF tries to take care of backwards compatibility with routers supporting TOS. This is done by encoding the QoS metrics in a such a way, that routers that still support the old OSPF TOS specification will route minimum delay or maximize throughput traffic over links with the lowest delay or highest available bandwidth. Incompatibilities may still arise though, since the TOS metrics are normally not used in practice because they are deprecated. Quagga for instance, handles TOS metrics incorrectly in the user interface code, which might lead to administrative issues, because it is unable to display LSAs with TOS metrics correctly in the administrative interface (*vtysh*).

Since QOSPF information is sent in the router LSAs, a change in one of the metrics of one of the links will cause an update of all metrics of all links of a certain router. This may cause unnecessary traffic or route recalculations, leading to a higher overall load on the network.

In the RFC only the available bandwidth and delay metrics were specified, but since the TOS field supports 32 different types, more QoS measures (e.g., jitter or reliability) can be specified in the future.

3.1.3 OSPF-TE

Traffic Engineering extensions to OSPF (OSPF-TE), defined in RFC3630 [23], provide exchange of QoS information over an OSPF network. OSPF-TE uses area-local opaque LSAs [10] to distribute the information through the network. For this reason, OSPF-TE only functions within an OSPF area and autonomous systems spanning multiple areas are not supported.

OSPF-TE defines an area-local opaque LSA type, the Traffic Engineering LSA (TE LSA), with identifier 1, which carries the TE information through the network. See Figure 3.1 for an example of an LSA header. The information is carried in Type-Length-Value (TLV) triplets for extensibility. Each TLV starts with a two octets type field, followed by a two octets length field. The length field specifies the length of the value part of the TLV, without padding, in octets. After the type and length the value portion follows, padded to four-octet alignment. An example of a TLV can be seen in Figure 3.2.

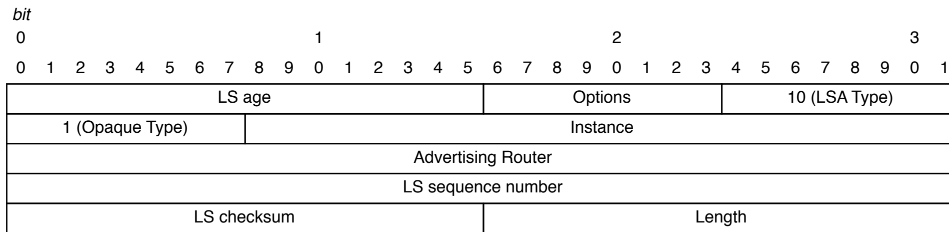


Figure 3.1 – LSA header layout for an OSPF-TE LSA.

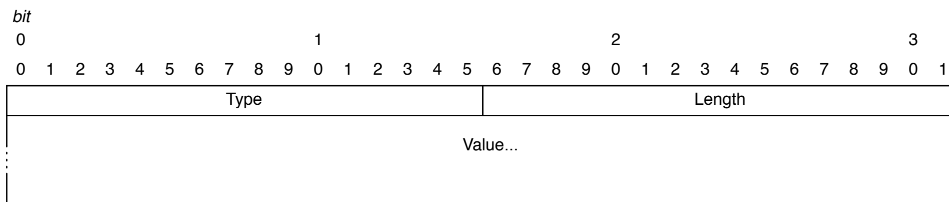


Figure 3.2 – Packet layout of a Type-Length-Value triplet.

Two TLV types have been defined, along with several sub-TLV types. The first TLV type is the Router Address TLV, which carries a stable IP address of the advertising router (e.g., a local loopback address). A Router Address TLV should be present in exactly one TE LSA originated by a router[3].

The second TLV type is the Link TLV. A Link TLV consists of various sub-TLVs and describes a link of a router. Each TE LSA should only carry one Link TLV, in order to make it possible to update the link-state of an individual link without updating the state of all other links.

Each Link TLV has two mandatory Link sub-TLVs: Link Type and Link ID. All other sub-TLVs are optional. The Link Type sub-TLV specifies if the link is a point-to-point or multi-access link. The Link ID sub-TLV specifies the Router ID of the neighbor (for point-to-point links) or the interface address of the designated router (for multi-access links)[4].

The other sub-TLVs describe local and remote interface addresses, an administrative TE metric, the maximum bandwidth, maximum reservable bandwidth and unreserved bandwidth of a link and the administrative group.

[3] Note that Quagga implements this incorrectly and sends the Router Address TLV in every TE LSA. Since this only generates some unnecessary data, it does not cause problems in practice.

[4] Note that Quagga implements this incorrectly and always sends the network address of a link as Link ID. Since we do not need the designated router address in our implementation, this causes no problems.

RFC5786 [1] defines an additional TLV type, the Node Attribute TLV, which contains all local addresses of a node not available through the Router Address TLV or Link TLVs.

3.1.4 OSPF-xTE

The Experimental Extension to OSPF for Traffic Engineering (OSPF-xTE), defined in RFC4973 [37], has been developed as a successor of OSPF-TE. Currently, the IETF OSPF working group is at the position that, although the OSPF-xTE proposal has some useful properties, OSPF-TE is sufficient for the traffic engineering needs of the community. We will nevertheless describe OSPF-xTE here, to give a complete overview on the OSPF Traffic Engineering extensions.

Unlike the Opaque LSA approach used by OSPF-TE, OSPF-xTE uses a new set of OSPF LSAs to distribute TE information. A separate TE link-state database (LSDB) is used, whereas OSPF-TE uses the same LSDB for TE and non-TE traffic. OSPF-xTE also supports multiple areas and can thus be used in larger autonomous systems. The capability advertisements specified in RFC4970 [26] are used to announce the OSPF-xTE support of a router.

When using OSPF-TE every node receives all link-state update traffic because area-local Opaque LSAs are used. OSPF-xTE only sends update traffic to TE-capable nodes, limiting useless traffic to non-TE capable nodes. Incremental updates of the LSDB are supported through separate incremental LSA types. It is also possible to specify a link as carrying only TE or non-TE traffic.

The above reasons make clear that OSPF-xTE certainly has some advantages over OSPF-TE. However, in the end, the exchanged TE information is the same. OSPF-xTE is more cumbersome to implement, because of the new LSA types and the separate LSDB.

3.1.5 Other protocols

Some other protocols that are relevant in the context of traffic engineering have been defined. RFC5329 [20] defines IPv6 support for OSPF-TE. The RFC expands OSPF-TE with some new sub-TLVs and a new TLV that define IPv6 versions of their IPv4 equivalents. Since our QoS router testbed is operating on IPv4, these additions are not necessary at this time.

RFC5305 [25] specifies TE extensions for IS-IS [31], another common interior routing protocol. The features of these extensions mirror those of OSPF-TE. Since the QoS router testbed already uses OSPF and the IS-IS extensions do not provide any new features compared to OSPF-TE, we have not further considered the IS-IS TE extensions.

3.2 Our own link-state update protocol

The goal of our project is to extend the QoS router testbed with the exchange of QoS measure information using various link-state update policies. The current testbed already uses OSPF for interior routing. We can either choose to extend this OSPF implementation, or create a protocol stack from scratch. If we choose to create an own protocol stack, we will not be compatible with any existing implementations. Also, we will have to do all the work ourselves (e.g., low-level packet creation and communication, flooding of updates, link monitoring, determining when to send an update). On the other hand, using the Quagga OSPF-API, it is possible to extend the current OSPF implementation. We can then reuse the existing OSPF flooding mechanisms, and only have to cater for the exact packet contents. This will save us a lot of work. Since OSPF opaque LSAs give us the opportunity to fully define the data we want to transmit, there are no real disadvantages to this approach. For these reasons, extending the current OSPF implementation seems like the best solution. Next, we will have to define what will be the exact contents of the packets we exchange.

Looking at the metrics that have to be exchanged, QOSPF is the only solution capable of distributing available bandwidth and delay information. Unfortunately QOSPF requires changes in the core OSPF protocol and implementation. This might introduce incompatibilities with existing routers and requires major changes in the complex core Quagga OSPF code.

Implementing OSPF-xTE would require some extension of the protocol with QoS metric information. Next to this, various new LSA types would have to be implemented in the Quagga code. These changes would be both complex and extensive due to the amount of new LSA types and the complexity of the Quagga code. Also, OSPF-xTE is not an industry standard, and probably never will be since the IETF OSPF WG chose not to enter the standards track.

OSPF-TE on the other hand is an IETF standard and implemented in most router software, including Quagga. It is easily extendible with new Link sub-TLVs to include available bandwidth or delay information. Due to the use of Opaque LSAs, the implementations of the flooding mechanism and the packet creation and interpretation are uncoupled.

In the end, we chose to implement a new protocol, which is an extension of the current OSPF-TE protocol. This way, we can develop our implementation separately from the Quagga core code and do not require any changes in the Quagga code. Communication with the Quagga core can be done through the OSPF-API. We could also have chosen to extend the Quagga OSPF-TE implementation, but then we would have had to change the Quagga codebase directly, which ties our implementation to a specific Quagga version. Experi-

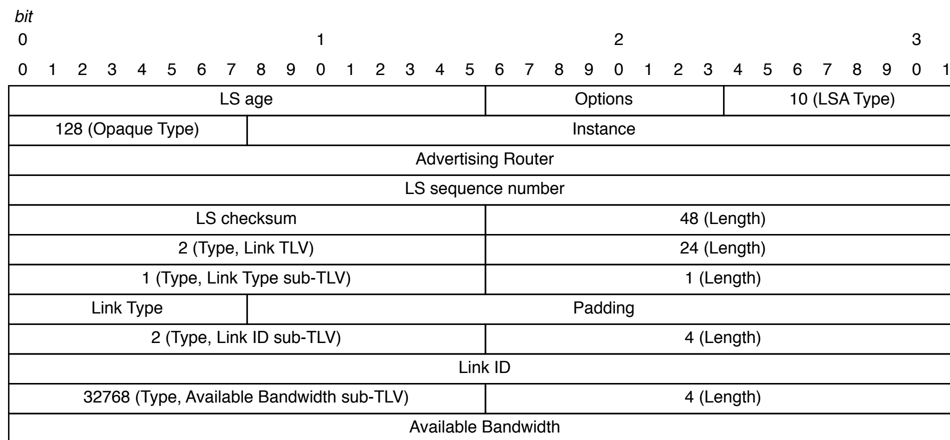


Figure 3.3 – Link-state protocol packet.

mental opaque type 128 was chosen as identifier for the new protocol and experimental Link sub-TLV type 32768 was chosen for exchange of available bandwidth information. Other new sub-TLVs can be added in the future, for exchange of other QoS measures like delay or jitter information.

Only Link TLVs are exchanged and each Link TLV has the same required sub-TLVs, Link Type and Link ID, as in the original OSPF-TE protocol. The new Available Bandwidth sub-TLV contains a 4 octets IEEE single precision float [18] as the value part, which indicates the available bandwidth on a link in bytes per second. The exact packet layout can be seen in Figure 3.3.

3.3 Implementation

In Chapter 2, we described the available link-state update policies. In the first part of this chapter, we created a network protocol to distribute link-state updates. We will now combine this research to implement link-state updates in the QoS router testbed. Several choices were made during the implementation. We will first describe the global choices we made (e.g., chosen programming language), after which the various parts of the implementation will be described in more detail. Together with the source code documentation, it should be possible to understand and improve the code further in the future.

This solution was developed in the C programming language [2]. C was chosen because Quagga is written in C, the OSPF-API is available in C, and the QoS router testbed is written in C++. Using C, it is little effort to integrate with the existing infrastructure of the QoS router testbed. Also, the low-level

memory access possibilities of C make it easy to compose network packets and implement a network protocol.

All code has been documented using Doxygen[5]-style comments. Using the Doxygen program it is possible to generate documentation pages in several formats that describe all functions and files of the application. This is similar to the more well-known Javadoc application for the Java platform.

Multiple programming paradigms exist, like procedural, object-oriented and functional programming. Each paradigm has its own advantages and drawbacks, which we will not fully describe here, but it is well known that the use of procedural languages, like C, can easily lead to *spaghetti code*, which leads to applications that are hard to maintain and improve. Although C is used as the language for our applications, a more object-oriented approach is possible. In that case each file should be handled as a separate unit, and any object instance related data (e.g., interface information) should be passed with a pointer to a struct. Private methods are possible by defining the corresponding procedures as being static, which makes them inaccessible from outside the file scope. We chose this approach to keep the code clear and maintainable.

The application can be split into several parts. First, a library was developed that can be used by other programs to provide link-state updates in a Quagga OSPF powered network. This library is the largest part of the work, and takes care of both the implementation of the link-state update policies as well as the implementation of the network protocol and the communication with Quagga.

To use the developed library, a command-line application and an extension to the Network Controller of the QoS router testbed were developed. Using the application or the extension, the library can be initialized and activated to send and receive link-state updates. Figure 3.4 gives an overview of the applications and their components. The developed library, application and extension will now be discussed in more detail, describing their possibilities, architecture and components.

3.3.1 Link-state update library

The link-state update library takes care of everything necessary to distribute link-state updates through the network. This includes the link-state update policy algorithms, the network protocol used to distribute the updates, measuring the bandwidth currently available and communication with Quagga. The library consists of five parts: the library interface, communication with the OSPF-API, interface management, packet creation, and logging. A global overview of the

[5] http://www.doxygen.org

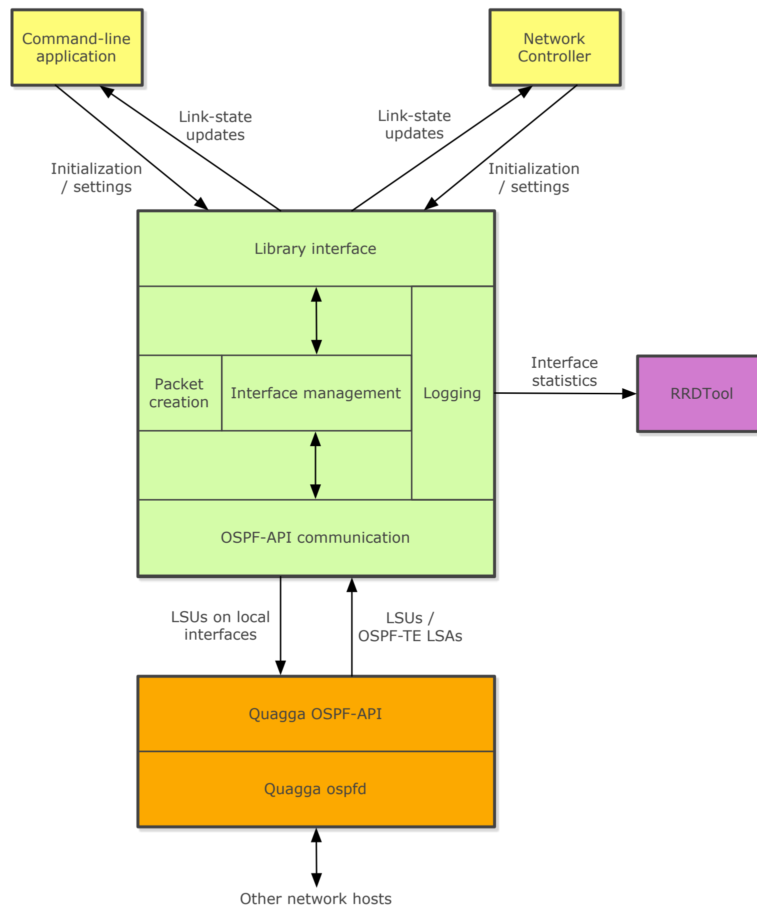


Figure 3.4 – Library and application architecture. The library is shown in green, the Quagga components are shown in orange and the user applications are shown in yellow.

different parts of the library is given in Figure 3.4. Every part of the library will now be discussed separately.

Library interface

The interface of the library, which is used by other applications to communicate with the library, is defined in `libospfteclient.h`. Some additional methods are defined in `interface.h`. Applications can include these header files and link the library to be able to read and transmit link-state information. All non-static method names are prefixed with `ospftecl_` to prevent name clashes.

An application begins by calling various setter functions in the library, to define the link-state update policy parameters or configure logging. It will then

call `ospftecl_init()` which sets up internal data structures and connects to the OSPF-API. After connecting the application can register various callbacks. These callbacks will be called by the library when a certain event happens. The full list of callbacks available is:

- Callback on receipt of a new OSPF-TE LSA.
- Callback on deletion of an OSPF-TE LSA from the link-state database.
- Callback on change in available bandwidth of an interface (by OSPF Link ID).
- Callback on change in available bandwidth of an interface (by Link Address).

The interfaces for which link-state updates should be transmitted should be defined by calling `ospftecl_add_interface()`. Finally, the application calls `ospftecl_run()` to start the processing of packets and monitoring of link-state. This method enters a loop, reading and writing network packets, and never returns. Figure 3.5 contains a sequence diagram describing how applications can use the library.

Communication with the OSPF-API

The Quagga OSPF-API[6] provides the possibility to communicate with Quagga through a socket to read the OSPF link-state database and send Opaque LSAs. The API is included in Quagga when the `--enabled-opaque-lsa` flag is given during the Quagga compilation process, which is true for most binary Quagga distributions. The API can then be enabled by giving the `-a` flag as a startup parameter to the *ospfd* process.

Unfortunately, there is not a lot of documentation on the API available. In the end we resorted to reading the OSPF-API implementation source code to understand the API. `ospf_apiclient.h` contains the various API functions. The basic idea is to connect to the API, register some callbacks for asynchronous messages, sync the link-state database, register an opaque type, and start a read loop. These function calls can be seen in `ospftecl_init()` in `libospfteclient.c`. All communication with the OSPF-API is done in this file. Figure 3.6 shows the exact sequence of function calls made to the OSPF-API.

[6] http://wiki.quagga.net/index.php/Main/OspfApi

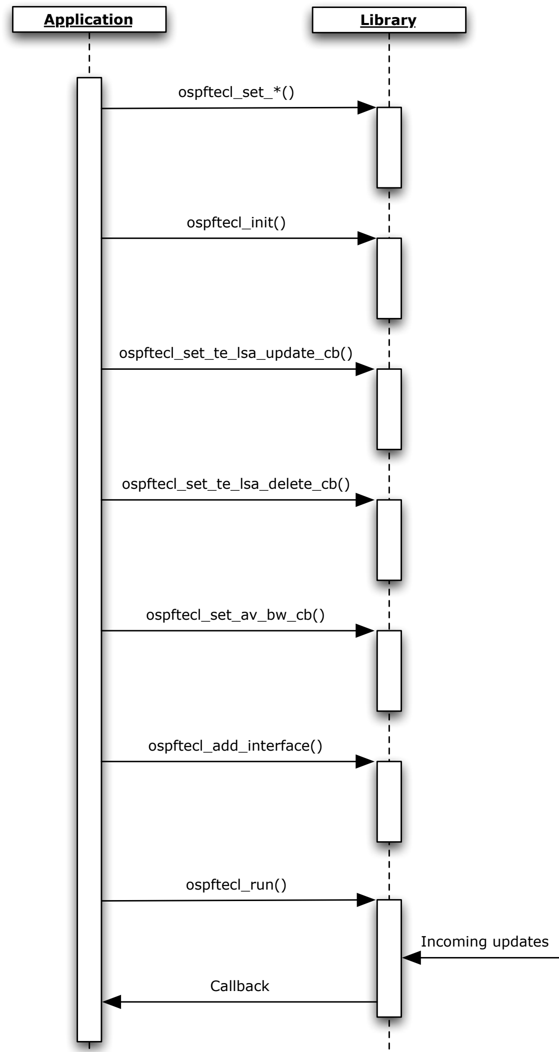


Figure 3.5 – Sequence diagram of communication between applications and the developed library. First, various setters are called that configure the library. Next, the library is initialized, callbacks are set and interfaces are added. Finally, the run loop is called, which calls the callbacks when new packets arrive.

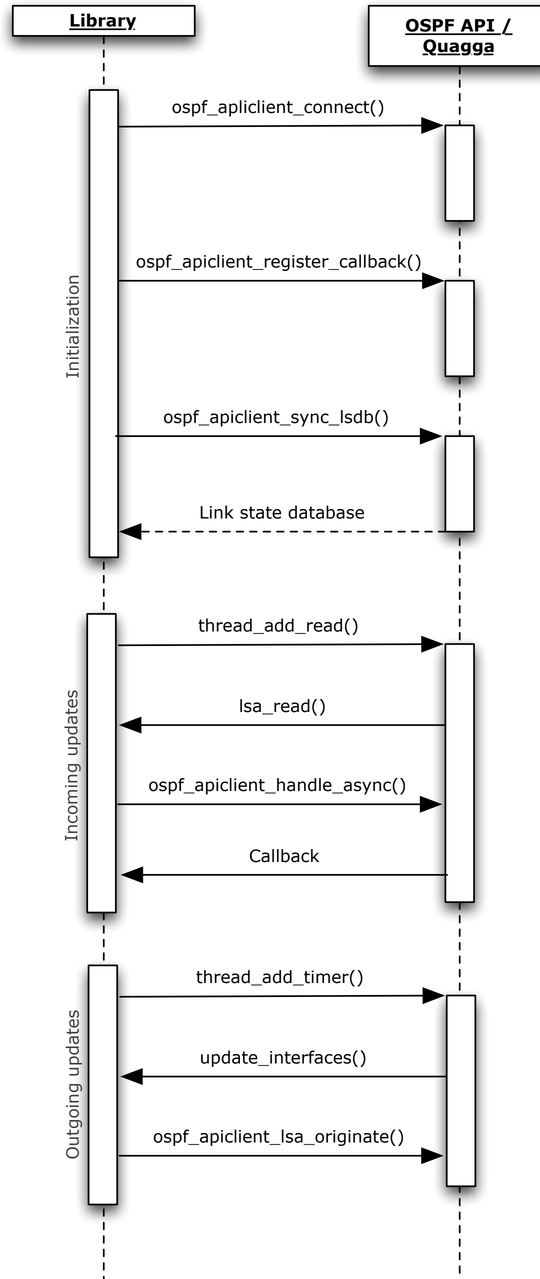


Figure 3.6 – Sequence diagram of communication with the Quagga OSPF-API. Both the initialization, the sequence for incoming updates (event/select-based) and the sequence for outgoing updates (timer-based) are shown.

Interface management

All interface information is stored in a linked list, see `struct ospftecl_interface_list` in `interface.h`. This list contains information like the name and IP of the interfaces, their maximum bandwidth, their interface state history and the last flooded value. Management of this list is done by the methods available in `interface.c`.

The available bandwidth of an interface is determined by reading `/proc/net/dev`. This file, provided by the Linux kernel, provides a counter for incoming and outgoing traffic on interfaces. By comparing the current and previous value of the counter, and the elapsed time, the current outgoing traffic rate can be calculated. Together with the configured maximum bandwidth, the currently available bandwidth can then be calculated by the method `ospftecl_get_available_bandwidth()`.

Since the update protocol requires specification of the Link ID, the IDs from the LSDB are retrieved and cached by the interface management code. This makes it possible to lookup a Link ID by local address and thus send out the correct ID in updates.

Packet creation

Packet creation is handled in `avbwtlv.c`. Only Link TLVs containing a Link ID, Link Type and Available Bandwidth sub-TLV have to be created. The LSA headers are created by the OSPF-API. The method `ospftecl_build_link_tlv()` first clears the given memory. Next it sets the TLV header and then creates the sub-TLVs. In this process all values are converted to network byte order, which is big-endian in IP networks [32].

Logging

The library provides the option to log the available bandwidth history for further analysis. This logging is done to a round-robin database, using RRDTool[7], or to a comma-separated values (CSV) file. Both the last flooded value as well as the actual value are logged to make it possible to compare the performance of various link-state update policies. RRDTool provides options to graph and analyze the data.

Integration with RRDTool is provided through wrapper methods in `rrdwrap.c`. The methods in this file take care of invoking the RRDTool library with the arguments in the right format.

[7] http://www.mrtg.org/rrdtool/

3.3.2 Monitoring and update daemon

To actually use the library, some application needs to include it, set the options and call the execute loop. A small application was developed that exposes all possibilities of the library as a simple command-line program. This application makes it possible to debug and test the library outside of the QoS router testbed (only a Quagga ospfd installation is required). Also, the application is used in the actual testbed integration, more information on that integration is given in the next section. This application, of which the sources are available in `ospfteclient.c`, exposes all options of the library through a command-line interface. All possible options can be retrieved by starting the program with the `-h` parameter.

OSPF-TE integration

The library needs to know what interfaces it should monitor, and for this reason interfaces have to be defined by specifying their local IP address and maximum bandwidth on the command-line. Since this would be a tedious task in larger networks, it is also possible to configure the interfaces dynamically using the OSPF-TE configuration of the local interfaces.

To be able to use OSPF-TE to configure our library, we need the local IP address and maximum bandwidth of an interface. In the Quagga implementation of OSPF-TE, the local address sub-TLV has not been implemented. We wrote a small patch for Quagga that adds local address support to their OSPF-TE implementation. Since the maximum bandwidth sub-TLV was already implemented in Quagga, we can now use the OSPF-TE LSAs to configure the network interfaces in our library.

3.3.3 QoS router testbed integration

To test the QoS routing protocols available, a QoS router testbed is available. This testbed has been extended to make it possible for QoS routing algorithms to take advantage of the link-state information distributed by our library. Due to our extension, the actual available bandwidth information is now available for QoS routing algorithms in a list inside the Network Controller, which determines the routing in the QoS router testbed. Actually using this information in the QoS routing algorithms was left for future research.

Before describing how we integrated the link-state update library in the testbed, we will first describe the layout and components of the QoS router testbed. A full description of the testbed is available in the work of Avallone [7].

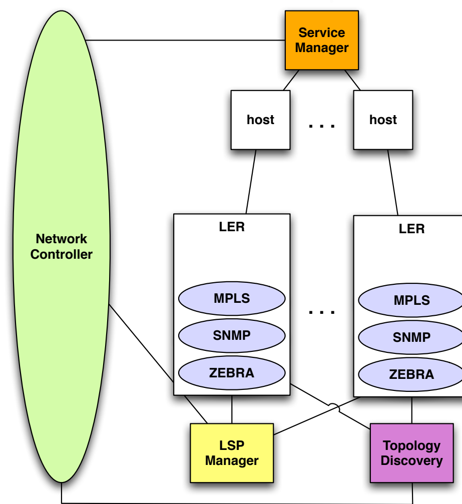


Figure 3.7 – QoS router testbed architecture.

Testbed architecture

Figure 3.7 gives an overview of the QoS router testbed architecture. The testbed uses a centralized approach, where the QoS router components like the Network Controller or Service Manager only have one instance in the network. All communication with the hosts and routers is done using `rexec`, a remote execution client. The various parts that can be distinguished in the architecture are:

Network Controller Makes routing decisions and has full knowledge of the network. Provides flow admission services and coordinates between the Service and LSP Manager.

Service Manager Receives requests for QoS flows from the hosts and communicates the requests to the Network Controller. When a flow is admitted by the Network Controller it will send this acknowledgement to the requesting host.

LSP Manager Sets up Label Switched Paths (LSPs) on request of the Network Controller. Communicates with the (Linux based) routers and hosts to setup and tear down LSPs and provide the traffic flows with their respective MPLS labels.

Topology Discovery Discovers the topology of the network using SNMP and communicates that topology to the Network Controller.

LER Label Edge Routers which provide the routing of QoS flows over the network after a path has been setup by the LSP Manager.

Host End-user clients that request QoS flows at the Service Manager and send and receive application traffic.

The Network Controller (NC) and Service Manager (SM) usually run on the same host in the network, which is one of the LERs. To setup a flow, a host contacts the SM, which will send this request to the NC. The NC then runs its configured routing algorithm to determine if a path is available for the requested flow, and will inform the LSP Manager if the flow is available. The LSP Manager will setup the path, after which the NC will acknowledge the flow to the SM, which will communicate this acknowledgement to the host.

Changes made for testbed integration

For our integration, the NC needs the available bandwidth information from the LERs. This is because the NC runs the routing algorithms and the LERs have the interfaces that need to be monitored. The LSP Manager takes care of the communication with the LERs, so in order to provide link-state updates in the network, both the NC and LSP Manager will have to be extended. Figure 3.8 shows how the various components work together to provide link-state updates in the network. We will now describe this figure in more detail.

First, the NC connects directly to its local Quagga installation using our library and sets the requested LSUP parameters. The local Quagga installation is available since the NC runs on one of the LERs. It will also register a callback that saves the available bandwidth information in a list. Access to the list is protected by a mutex to provide thread-safety, since the library callbacks will be done in a separate thread.

Next the NC will loop through the list of interfaces available in the network. For local interfaces it will add the interface using the library call. For interfaces on other hosts it will send an `INITCAP` message to the LSP Manager, which starts the command-line client on the host to monitor the interfaces and send out updates. The format of this `INITCAP` message is as follows:

```
INITCAP <interface ip> <interface speed in bytes/s> ...
  <interface ip> <interface speed in bytes/s>
  <other arguments for command-line program>
```

After this setup phase, the NC will create a new thread using pthreads and call `ospftecl_run()`. This new thread will then receive available bandwidth

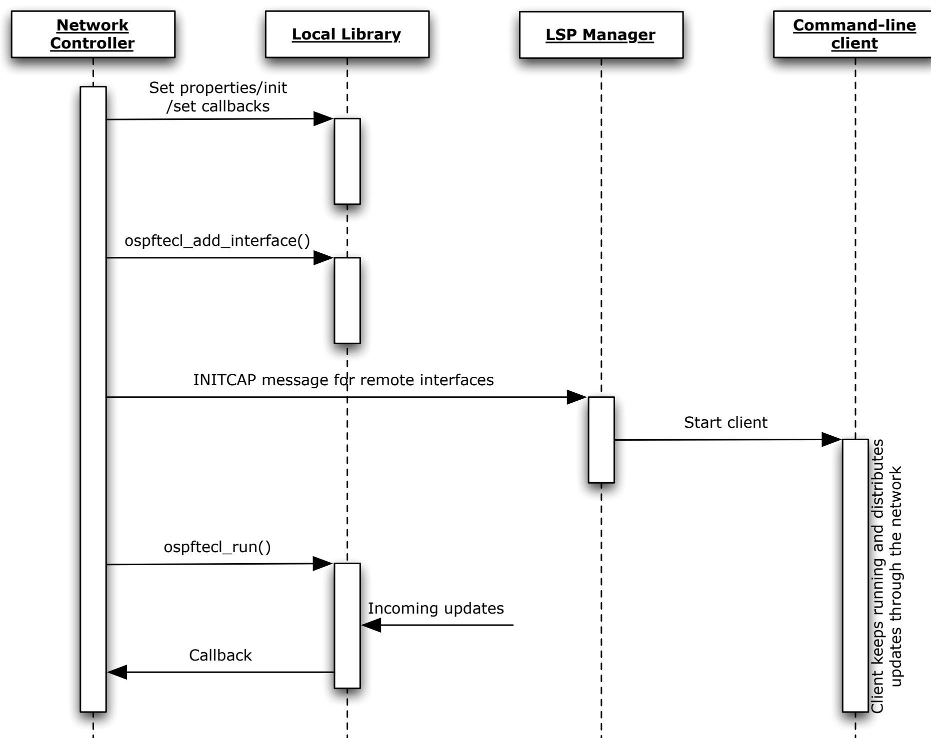


Figure 3.8 – Interaction between the Network Controller, local copy of the link-state update library, the LSP Manager and the command-line client on the Label Edge Routers.

updates and invoke the callback on each update, which stores the available bandwidth in the list.

Integration of the available bandwidth information in the QoS routing algorithms is left for future work. We have focussed on implementing the LSUPs and making the information available to the NC.

4
Verification of the software

Verification of the software is an important step. Without verification it would not be possible to trust the measurement results of the developed tools. For this reason, unit tests have been implemented. Also, an extensive suite of real-life tests was run. In this chapter we first discuss the performance criteria for our software. We then proceed to our unit testing setup, after which we define various test scenarios for real-life tests. In the end we present the results of our verification process.

4.1 Requirements and performance criteria

Before we are able to test our software, the exact requirements and performance criteria for the software have to be determined. Several software development methodologies exist. With classic methodologies, like the waterfall model, requirements of the software are determined upfront. In our case, an agile model was used, where requirements (or *user stories*) and new features were added during multiple development cycles of the software. For instance, the background daemon code and the option to specify IP and bandwidth pairs on the command-line were not added until they were necessary for the integration with the Network Controller. Also, the various update policies were implemented and extended as new policies were 'discovered' in literature. For this reason, we will now first give an overview of all the user stories that were implemented, and the exact requirements and performance criteria the software must meet in order to

be able to do reliable tests comparing link-state update policies.

The software implements five link-state update policies: timer-based updates, updates on an absolute change in available bandwidth, updates on an relative change in available bandwidth, equal sized class-based updates, and exponential sized class-based updates. Each of these policies has to be tested for correct functionality. For a description of the functionality we refer to Chapter 3. Every policy has its own set of possible parameters. The list of parameters is given in Table 4.1 and the specific parameters of the policies are detailed in Table 4.2.

	Parameter description
C	Check interval in seconds
H	Holddown timer in seconds
M	Moving average over last n measurements
A	Absolute amount of change required before broadcast (in bits/s)
R	Relative amount of change required before broadcast (in %)
D	Number of class boundaries to cross before broadcast
N	Number of classes
B	Base factor to determine class boundaries
G	Growth factor to determine class boundaries

Table 4.1 – Parameters available for the link-state update policies.

Link-state update policy	C	H	M	A	R	D	N	B	G
Periodic	•	•	•						
Absolute change	•	•	•	•					
Relative change	•	•	•		•				
Equal classes	•	•	•			•	•		
Exponential classes	•	•	•			•		•	•

Table 4.2 – Parameters used by the link-state update policies. The letter coding can be found in Table 4.1.

Apart from update policies there are also some other features in the software that have to be verified, including logging, interface management and general application behavior. This concerns the following features:

- Daemonize after startup.
- Automatic interface management using OSPF-TE.

- Manual interface management by specifying IP/bandwidth pairs on the command-line.
- Log data to a round robin database.
- Log data to CSV files.
- Show help text.

Next to the features of the program we also have some performance criteria. These criteria do not reflect actual functional requirements, but merely define the technical specifications the application should adhere to. We can define the following criteria:

- *Stability*: the application should not crash, dump core or have any memory leaks.
- *Reliability*: link-state updates should be transmitted reliably and should not be lost. The link-state should always be the same on all hosts (after the flooding time), and no updates may be lost. Since OSPF is used as data carrier, and the OSPF specification specifies a reliable flooding algorithm, we can assume this criteria is met (by assuming the correctness of Quagga).
- *Speed*: update flooding should not take a significant longer time than normal routing protocol (OSPF) updates. This criteria is met since OSPF LSAs are used as the flooding mechanism. Shaikh and Greenberg have shown LSA processing and distribution takes about 30 to 40 milliseconds on a common router [35], so updates should generally be available within a second even on large OSPF networks.
- *Accuracy*: the flooded information should be sufficiently accurate to form the basis for routing decisions. In practice this means that flooded available bandwidth numbers should not differ significantly (> 1%) from their actual values. This way, small rounding and floating point errors are permitted.

Now the requirements of the application have been defined, we can define the way the application is tested. First, the application will be unit tested, to confirm the correct working of the basic functionality of the program. Afterwards, we will define the test scenarios that we will run in our lab.

4.2 Unit testing

Unit testing tests the functionality of the smallest building blocks of programs: units. When using the programming language C [2], these units generally refer to the various procedures of the program. By applying unit testing, we can verify the correctness of separate parts of the program, and locate errors faster and easier. Since unit testing only tests small parts of the program at a time, more testing is necessary to ensure correctness of the complete program. For that reason we will also run the program in various real-life scenarios later on.

In order to assist unit testing, several frameworks have been developed. The most well known framework is JUnit for Java[1]. Also for the C programming language unit testing frameworks have been developed. We compared the available frameworks on their features and impact on the overall project source. Some frameworks, like Check[2], have extensive features, such as segmentation fault detection, various output methods and fixtures. Unfortunately these features come with a downside. Check is quite a large package, needs a separate library and is tightly integrated with autotools, a build system that is not used by our application. Since our application is quite small, the number of test cases is limited, and a full-fledged framework like Check seems overkill. The same seemed true for CUnit[3].

For this reason we started looking at smaller frameworks. CuTest[4] proved to be a simple and concise framework. It does not provide features like segmentation fault detection, but those are not really necessary: it is also possible to test the return code of the test suite to detect such an error. CuTest consists of one source file and one header file. Tests can be created together with the source of the application, or in a separate source file. With a small shell script it is possible to automatically find the test cases in the source code and write a wrapper application for them. The application shows the progress of the test suite and reports any faults.

An other option would have been to not use a framework at all, but just define some preprocessor macros. This would have been possible, but then we would have probably ended up reproducing something like CuTest, which is essentially the same: some small assertion procedures and preprocessor macros.

In the end 15 unit tests were implemented. Not all functionality of the application could be unit tested. For instance, parts of the application concerned

[1] http://junit.sourceforce.net
[2] http://check.sourceforge.net
[3] http://cunit.sourceforge.net
[4] http://cutest.sourceforge.net

with communication with Quagga and the rest of the network cannot be unit tested since we would then be testing the complete functionality of Quagga instead of just a small unit of our application. In fully object-oriented languages, mockups would be used to overcome this problem, but the C programming language does not offer such features in an easy way.

The tests that were implemented test the following features:

- Interface management (adding and removing of interfaces).
- Available bandwidth calculations.
- Determining bandwidth in use.
- Calculating the current bandwidth class in case of class-based update mode.
- Mapping of interface addresses to interface names.
- The various update policies (when to update or not to update).
- Memory allocation and reallocation.

To check for memory leaks and other programming errors all tests were run using the tool Valgrind[5]. Valgrind detects memory leaks and incorrect use of pointers and prints a report after the program completes. No memory leaks or other problems were found.

As said, not all parts of the application can be tested using unit testing. To test the complete functionality of the application, some real-life tests are necessary. In the next sections these tests will be defined and executed.

4.3 Test scenarios

In order to test the functionality of the application, several test scenarios were defined. Every link-state update policy is tested at least twice, with different parameters (except for the periodic, timer based update policy, which does not have any parameters). That way an incorrect implementation of the parameter processing or the algoritm should come to light. Also various check intervals and holddown timers are tested, and some experimenting is done with the moving average parameter. The exact parameters used are shown in Table 4.3. When choosing the parameters, equivalence partitioning was used. This is a standard software testing technique, where the input is divided into equal classes to limit

[5] http://valgrind.org/

#	LSUP	Check	Holddown	Parameters
1	Periodic	1	5	-
2	Periodic	1	5	Average: 10
3	Periodic	5	20	-
4	Absolute	1	5	Threshold: 1,000,000
5	Absolute	1	5	Threshold: 3,000,000
6	Relative	1	5	Threshold: 10%
7	Relative	1	5	Threshold: 30%
8	Equal class	1	5	Nr. of classes: 5
9	Equal class	1	5	Nr. of classes: 5; Change required: 3
10	Equal class	1	5	Nr. of classes: 15
11	Exp. class	1	5	Base fact.: 0.1; Growth fact.: 1.2
12	Exp. class	1	5	Base fact.: 0.2; Growth fact.: 1.4

Table 4.3 – Link-state update policy parameters for software verification. *Check* indicates the traffic measurement interval in seconds. *Holddown* indicates the holddown timer in seconds.

the number of test cases. For instance, by testing a check timer of 1 second, and a check timer of 5 seconds for the periodic policy, we cover all check timers values for all policies, since the check timer behavior is not policy specific.

The network used for the tests is a simple hub-and-spoke network, shown in Figure 4.1. The network has two groups of two end nodes, and two core nodes. The capacity of the links on the path between nodes 1 and 2 is 100 Mbit/s, while the links on the path between nodes 5 and 6 have 10 Mbit/s capacity. This difference is to test if the application can handle multiple link speeds correctly. During the tests data is generated using the D-ITG traffic generator [11]. Some adjustments to the D-ITG code were necessary to solve bugs in the multithreading code. Data flows from node 1 over 3 to 2, and from 5 over 4 to 6. The amount of data changes every 15 seconds. For the first flow traffic rates will be 0, 5, 15, 30, 80, 90, 80, 30, 15, 5 and 0 Mbit/s. This way, the behavior of the application on narrowing and widening links is tested. For the second flow, which is on a 10 Mbit/s link, we will use the following traffic rates: 0, 0.5, 1.5, 3, 8, 9, 8, 3, 1.5, 0.5 and 0 Mbit/s. Because both small and large steps in available bandwidth are made, all update policies will be triggered to send an update. D-ITG parameters for both flows can be found in Tables 4.4 and 4.5. A constant packet size of 1 Kbyte (958 bytes of data, 8 bytes of UDP headers, 20 bytes of TCP headers and 14 bytes of Ethernet headers) is used, combined with a changing packet rate. The flows are modelled in D-ITG by sending multiple flows after each other, where

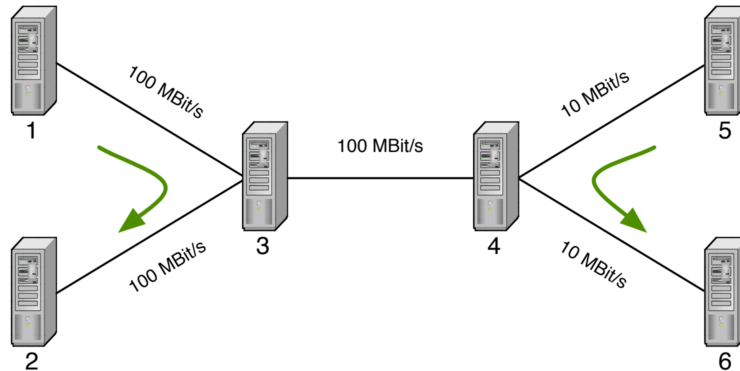


Figure 4.1 – Verification network setup. The green arrows indicate the traffic flow of the generated traffic.

Traffic (Mbit/s)	Packets/s	Packet size	Duration (ms)
5	625	958	15,000
15	1,875	958	15,000
30	3,750	958	15,000
80	10,000	958	15,000
90	11,250	958	15,000
80	10,000	958	15,000
30	3,750	958	15,000
15	1,875	958	15,000
5	625	958	15,000

Table 4.4 – D-ITG parameters for 100 Mbit/s links.

each flow has a duration of 15 seconds. The first flow starts immediately, the second flow starts after 15 seconds delay, et cetera.

4.4 Results

The results of the twelve measurement runs can be seen in Figure 4.2. The graphs show the actual and broadcasted available bandwidth (in Mbit/s) on the 100 Mbit/s link from node 3 to 2 and on the 10 Mbit/s link from node 4 to 6. The solid black line indicates the actual available bandwidth on the link, while the red and green lines indicate the broadcasted available bandwidth.

All graphs were manually verified for correctness. This was done by checking the actual available bandwidth history with the algorithm descriptions of Chap-

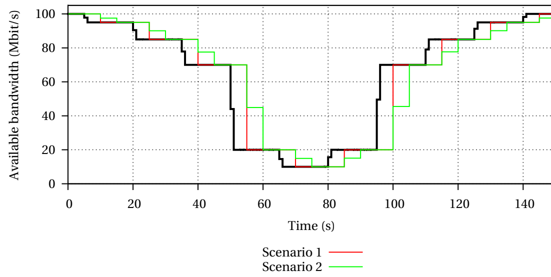
Traffic (Mbit/s)	Packets/s	Packet size	Duration (ms)
0.5	63	958	15,000
1.5	188	958	15,000
3	375	958	15,000
8	100	958	15,000
9	1,125	958	15,000
8	1,000	958	15,000
3	375	958	15,000
1.5	188	958	15,000
0.5	63	958	15,000

Table 4.5 – D-ITG parameters for 10 Mbit/s links.

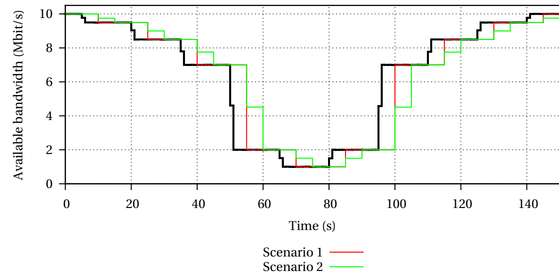
ter 2. For instance, in scenario 6 (relative policy, threshold 10%), we know the broadcasted value should not change after 5 seconds (change from 100 Mbit/s to 95 Mbit/s available bandwidth, which is 5%), but should change after 20 seconds (when the available bandwidth changes to 85 Mbit/s, a difference of 15 %). Looking at Figure 4.2e we can see this indeed happens. We can do this verification for every algorithm and all expected broadcasts of (changes in) available bandwidth.

Some of the policies required inspection of the logfiles to see if updates were sent at the right moments. For instance, scenario 1 and 2 update every 5 seconds, which is hard to see in the graphs because the available bandwidth displayed in the graph does not change every 5 seconds. After manual verification we were able to conclude that all policies are correctly implemented and updates are sent out at the right moments.

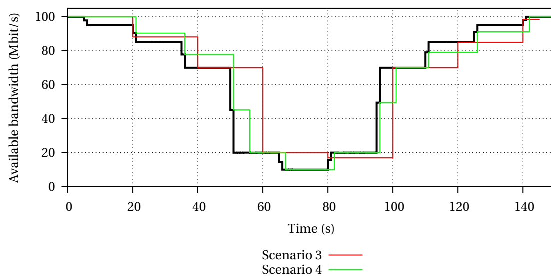
Other features of the program, like daemonizing, logging to RRD files and interface configuration from the command-line, were tested manually by running the command-line application with the corresponding parameters, and were found to be functioning correctly.



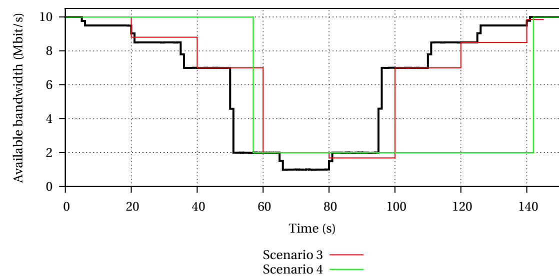
(a) Scenario 1 & 2: 100 Mbit/s link



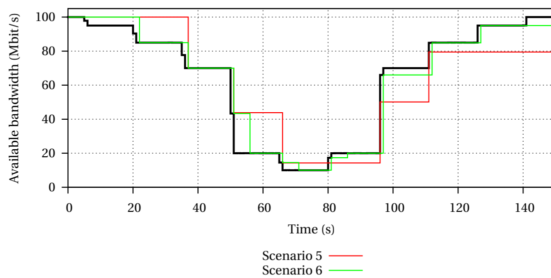
(b) Scenario 1 & 2: 10 Mbit/s link



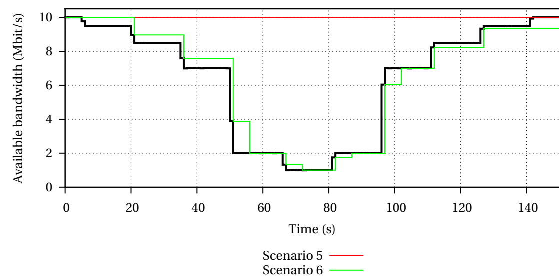
(c) Scenario 3 & 4: 100 Mbit/s link



(d) Scenario 3 & 4: 10 Mbit/s link

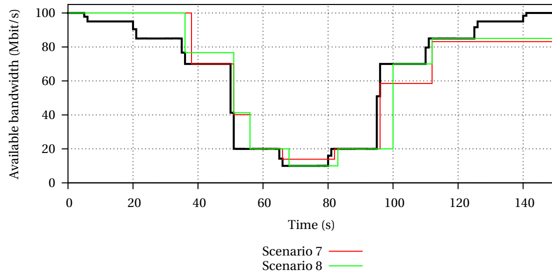


(e) Scenario 5 & 6: 100 Mbit/s link

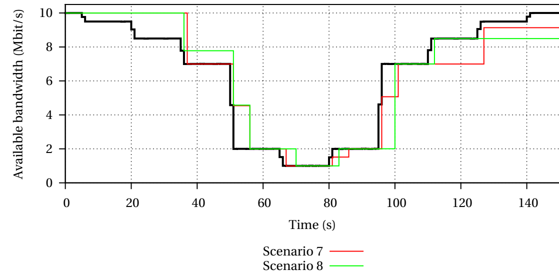


(f) Scenario 5 & 6: 10 Mbit/s link

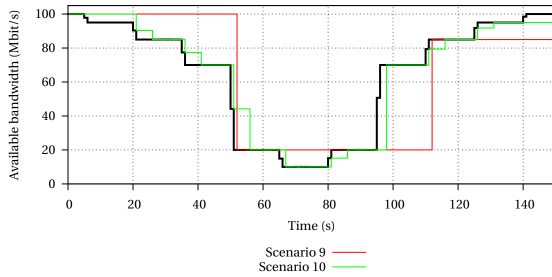
Figure 4.2 – Results of verification measurements. The solid black line shows the actual available bandwidth, while the red and green lines show the available bandwidth as broadcasted through the network. *Figure continues on the next page.*



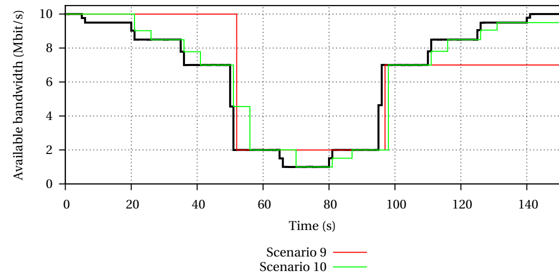
(g) Scenario 7 & 8: 100 Mbit/s link



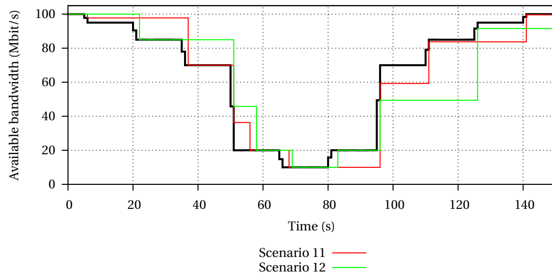
(h) Scenario 7 & 8: 10 Mbit/s link



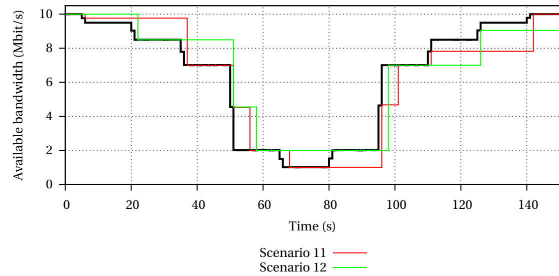
(i) Scenario 9 & 10: 100 Mbit/s link



(j) Scenario 9 & 10: 10 Mbit/s link



(k) Scenario 11 & 12: 100 Mbit/s link



(l) Scenario 11 & 12: 10 Mbit/s link

Figure 4.2 – *—continued—*

5
Comparison of link-state update policies

In the preceding chapters, we have built a solution to distribute link-state updates through a network according to several policies. This gives us the opportunity to compare the performance of the link-state update policies and to try to find an optimal policy. In this chapter we will first present a test setup and discuss the performance metrics we use to compare link-state update policies. Multiple scenarios will be defined, using real-life traffic, to measure the performance of the policies. Finally, the results of the comparison will be presented.

5.1 Performance metrics

To compare the available link-state update policies, some performance metrics have to be defined. These metrics have to measure the quality and the quantity of the updates. A higher update quality will lead to more accurate link-state information in the network, which in turn might lead to more accurate routing. A higher update quantity will lead to more updates being flooded through the network, which causes a higher processing load on the routers.

Past research has been mainly focussing on the simulation of complete QoS networks. This made it possible to measure the quality of the link-state updates by looking at call (or flow) acceptance and setup rates. These numbers give a complete picture of the performance of a certain network, which can certainly be an advantage. In our case, only link-state updates were implemented, and a full QoS network that uses these updates is not (yet) available. Also, we

specifically want to compare the link-state update policies, and leave out any bias caused by the use of a certain QoS routing algorithm. For these reasons, only the direct measurement data is used: actual available bandwidth at time interval t to $t+1$, B_t, and the available bandwidth last flooded through the network at time t, B'_t. Using this data we can define the following variables:

$$\text{absolute link-state update error} = e_a = |B'_t - B_t|$$

$$\text{relative link-state update error} = e_r = |\frac{B'_t - B_t}{B_t}|$$

The relative link-state update error tries to take into account that flows normally only get rejected when a link is full. An error in the link-state update value is therefore less important when a link has ample bandwidth left. Both the mean and standard deviation of these values will be taken into account when we discuss the results. Because the absolute value is used, no difference is made between a negative and positive error. The reason for this is twofold. First of all, because we are not testing a full QoS routing setup, but only the link-state update policies, it is hard to predict the exact influence of the link-state update error on the network. If the broadcasted available bandwidth value is too low (compared to the actual available bandwidth), this will lead to routing failures. A too high value for the available bandwidth will lead to setup failures. The exact influence of these routing and setup failures on the efficiency of the network depends on the network layout and routing algorithms in use [16]. Also, previous work has used the same absolute error performance metrics [24]. Reusing these metrics makes it possible to compare the results.

Note that there will always be an offset between the currently available bandwidth and the last flooded value. This is because the value flooded in time interval t to $t+1$ is based on the actual available bandwidth measured in time interval $t-1$ to t. With dynamic traffic patterns, this offset might lead to link-state update policies performing differently from previous experiments by other researchers. Most, if not all, previous work was done under the assumption of flow-based measurements, where new link-state updates are sent on admission of a new flow. Since our test setup does not use explicit flow admission control, we are unable to determine the bandwidth a flow uses until after the traffic has been sent.

To measure the quantity of the updates, we simply use the average time between updates. In our measurements we will try to find the parameters for the link-state update policies that result in the same link-state update error. We can then compare the average time between updates to see which policy performs best.

5.2 Test setup

For the comparison we will reuse the network from the previous chapter, as shown in Figure 4.1. This network provides a simple 100 Mbit/s and 10 Mbit/s path. We will stream traffic from nodes 1 and 5 to, respectively, nodes 2 and 6 again. The measurements on the link-state update policies will be performed on nodes 3 and 4. The reason we use this relatively simple network instead of a complex one is twofold. First of all, a larger network would not add much value, since part of our measurements consists of replaying and simulating traffic from backbone links and all network dynamics are already captured in that traffic pattern. Also, we want to focus on the link-state update policies, and not on any QoS routing algorithms, which are necessary to route traffic over a more complex network.

To generate traffic on the network, several options were considered. The first option considered was using a traffic generator, such as D-ITG [11], to generate the traffic on the links. A problem with this approach is that the burstiness of the traffic is somewhat limited. Even with a large variance in inter-arrival times, on a five second interval the traffic patterns still appear as a constant, nonvarying load. This means no link-state updates will be generated, which makes traffic generators unsuitable for comparing link-state update policies. To generate more dynamic traffic using traffic generators, multiple instances of the generator would have to be started, each representing its own flow. The starting and stopping of these instances then determines the dynamics of the network traffic. Problem with this approach is that is it hard to find a model that captures a real-life situation. Often Poisson processes are used to model the flow start, size and duration. However, for most Internet traffic it is still unclear if the traffic can be modeled accurately this way [22]. For these reasons we looked at other ways to generate realistic network traffic.

Another option to generate the traffic is the open source video player VideoLAN (VLC)[1]. This player has the ability to stream a video over the network. Combined with the right codecs, using a variable bit rate, this leads to dynamic traffic patterns that can make a good comparison between link-state update policies possible. Depending on the image size and codec type, the bit rate can be varied between a few hundred kilobit per second and several megabits per second.

A last option to generate realistic network traffic is replaying an already captured traffic trace from an existing network link. This is possible using the Tcpreplay suite of utilities[2]. First the existing traffic trace is modified using

[1] http://www.videolan.org

[2] http://tcpreplay.synfin.net

tcprewrite. This tool makes it possible to inject the destination MAC address of the next-hop router, and alter the source and destination IP addresses of all flows in the capture. Also the packets are expanded to their original size, since usually traffic traces only contain the headers of the packets. Next, the altered traffic trace is replayed on the network using *tcpreplay*. Because existing trace files are used, it is possible to generate network traffic similar to traffic on backbone network links, by using only one link connecting two nodes in a test setup.

5.3 Test scenarios

For the test scenarios we have to determine the actual link-state update policy parameters we will use during our tests. Also we have to decide on the actual traffic pattern we will play back over the links during the test.

The parameters chosen for the link-state update policies can be found in Table 5.1. The values of the parameters are based on existing literature, like the research of Lekovic and Van Mieghem [24]. For the check interval, the interval on which our tool measures the available bandwidth and checks if an update is necessary, we chose two fixed values of 1 and 5 seconds. Also, we always enable a minimum hold-down timer of 5 seconds, since the OSPF specs do not allow more frequent updates and Quagga employs a built-in hold-down timer of 5 seconds. A smaller check interval should give a more accurate view of the available bandwidth, but might also lead to less accurate updates, since updates can only be distributed every 5 seconds. Check intervals higher than 5 seconds will average the traffic rate so much that the link-state update policy used does not make any difference anymore.

Some parameters, like the check interval, will be used for every link-state update policy. Other parameters, like the number of classes in case of an equal-sized classes policy, are specific to a policy. Also, some parameters exclude each other: since the moving average parameter is meant as a replacement for the hold-down timer, we will not use these parameters together. These parameters amount to 228 test scenarios in total.

The traffic generated on the 100 Mbit/s link between node 1 and 2 consists of a 15 minute trace on a backbone link of the WIDE project[3]. A graph of the traffic on the link can be seen in Figure 5.1. This capture was taken from a 100 Mbit/s connection in a Japanese research network on the 14th of October, 2006. The average traffic rate is 43.26 Mbit/s, with a standard deviation of 7.49 Mbit/s (when sampling every second). This traffic trace has been chosen because it

[3] http://tracer.csl.sony.co.jp/mawi/

Parameter	Min	Max	Step
Check interval (s)	1	5	4
Hold-down timer (s)	5	45	20
Moving average (# measurements)	0	15	5
Absolute change (100 Mb/s link) (B/s)	125,000	1,625,000	500,000
Absolute change (10 Mb/s link) (B/s)	12,500	162,500	50,000
Relative change (%)	10	50	10
Number of classes	5	20	5
Base factor	0.01	0.05	0.04
Growth factor	1.1	1.9	0.2
Classes change before update	1	2	1

Table 5.1 – Link-state update policy parameters used for the comparison. For each parameter a minimum and maximum value is given, together with the step value by which the parameter is increased.

is a very dynamic, bursty trace. This should make it possible to highlight the differences between the link-state update policies well.

The traffic on the 10 Mbit/s link is generated by replaying a H.264 video from the Blender Institute: Big Buck Bunny[4]. The movie has a resolution of 854x480 pixels, and a runtime of 9 minutes and 57 seconds. A graph of the network traffic generated while playing the movie can be seen in Figure 5.2. This movie has been chosen because it is freely available, and because the H.264 codec results in a dynamic traffic pattern. The other available codecs resulted in a constant bitrate stream, which is not very useful for link-state update policy measurements.

In the graph can be seen that the complete link is filled at the end of the movie, between 530 and 560 seconds. This seems to be because the H.264 codec cannot handle the credits very well and uses a high bit rate to encode them. Because a full link will only cause packet loss and interfere with our measurements, we decided to use only the first 500 seconds of the movie for traffic generation. The average traffic rate is 3.26 Mbit/s, with a standard deviation of 1.34 Mbit/s.

Of each measurement result, the first 75 and last 10 seconds are ignored. The first 75 seconds are skipped because, when using a moving average, this is the time necessary to fill the interface history. The last 10 seconds are skipped to account for the fact that our test scripts cannot stop the link-state updates client and the traffic generation at exactly the same moment.

[4] http://www.bigbuckbunny.org

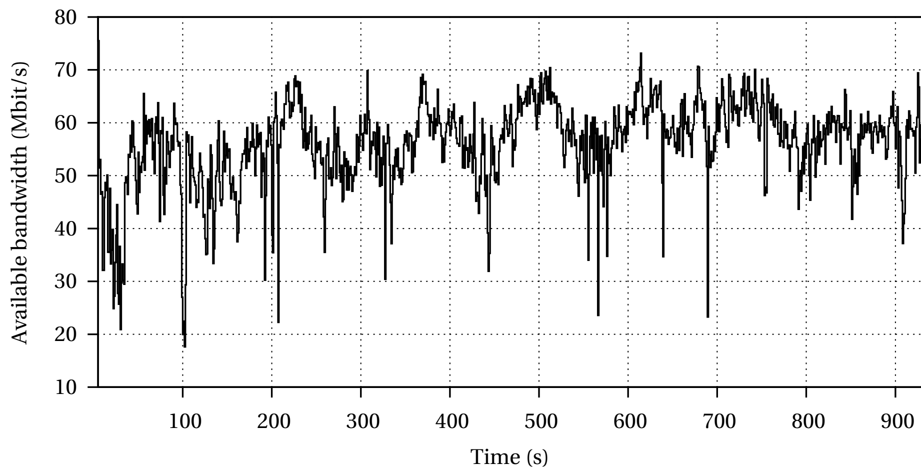


Figure 5.1 – Traffic pattern generated while playing back the dump from the MAWI Working Group of the WIDE project, measured on a 1 second interval.

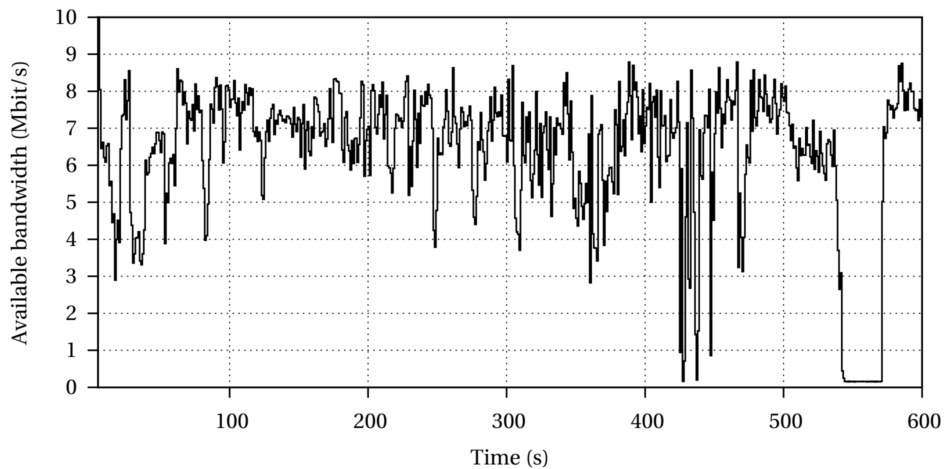


Figure 5.2 – Traffic pattern generated while playing back the movie *Big Buck Bunny* in 480p quality, measured on a 1 second interval.

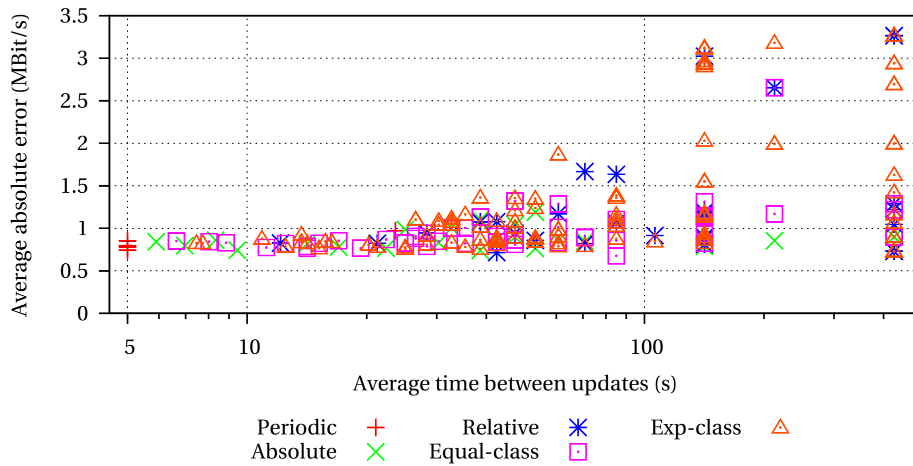


Figure 5.3 – Absolute error versus update rate for 10 Mbit/s, 5 second check interval scenario.

5.4 Results

In this section we will present the results of our measurements. Discussing these results and relating them to previous work is left for the next chapter, although some short explanations for the observed behavior will be given. We will first present the results of the 10 Mbit/s scenario, where a video stream was played back over the network. Next, the results of the 100 Mbit/s scenario, where a traffic dump was replayed, will be presented. For each scenario, graphs will be given that show the average absolute and relative error versus the update rate for each test case, which results in 228 points per graph. Also, the influence of the hold-down timer and moving average settings on the error is presented in a table. Because of the large amount of measurements, only the highlights are discussed. This includes the best and average results. The full result set can be obtained from the author on request.

5.4.1 10 Mbit/s scenario

Figure 5.3 graphs the average absolute error versus the update rate for the 10 Mbit/s scenario. This scenario used a 5 second check interval. The graph is divided according to the link-state update policy used. Figure 5.4, which includes the average relative error instead of the absolute error, shows a pattern that is comparable to the previous graph.

The lowest average absolute error found is 0.68 Mbit/s, using an equal-sized classes policy. This policy generates an update, on average, every 85

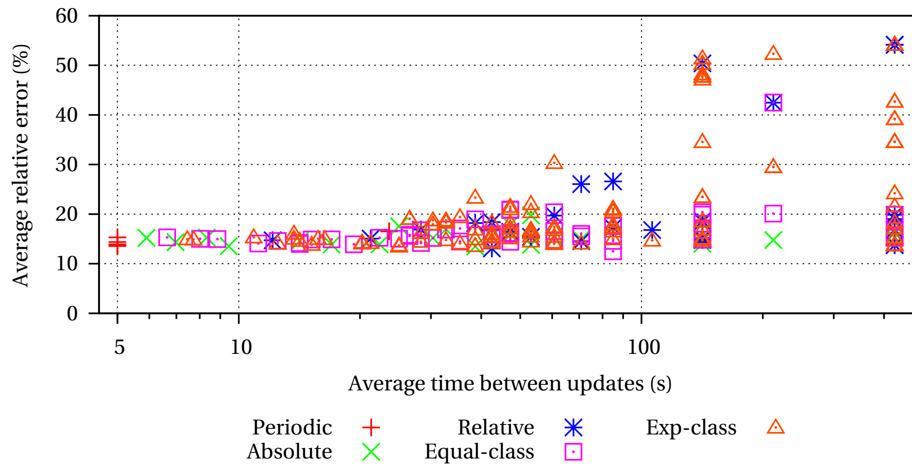


Figure 5.4 – Relative error versus update rate for 10 Mbit/s, 5 second check interval scenario.

seconds, and has an average relative error of 12.5%. There are, however, a lot of policies that do not generate any intermediate updates during the test. Some of these policies perform on a level comparable to the best performing policy. For instance, 9 different policies that do not generate any updates have an average absolute error of 0.73 Mbit/s, which is only slightly above the best performing policy. Figure 5.5 shows the detail graph for one of those policies. From the graph can be seen that no updates are generated. Nevertheless, the average absolute error is only 0.73 Mbit/s. If we compare this to a policy that generated more updates, like the one in Figure 5.6, we might think the latter provides a smaller average error. In reality, the average absolute error of the second policy is 0.83 Mbit/s. Figures 5.7 and 5.8 show the error between the actual and broadcasted value, and clearly show that the second policy does not perform better than the first.

Because the rather large average error was suspected to be caused by the offset between the measurement of the available bandwidth and the distribution of the link-state update (as already discussed in Section 5.1), it was decided to repeat the measurements with a 1 second check interval. The results of these measurements can be found in Figures 5.9 and 5.10.

Comparing Figure 5.3 to 5.9, we can see that the average absolute error increases in the 1 second check interval scenario. The average absolute error increases from 1.27 Mbit/s to 1.32 Mbit/s (+4.1%). Also, the minimum average error increases to 0.91 Mbit/s. The best performing policy is an absolute change policy that sends an update every 5 seconds. The average time between updates

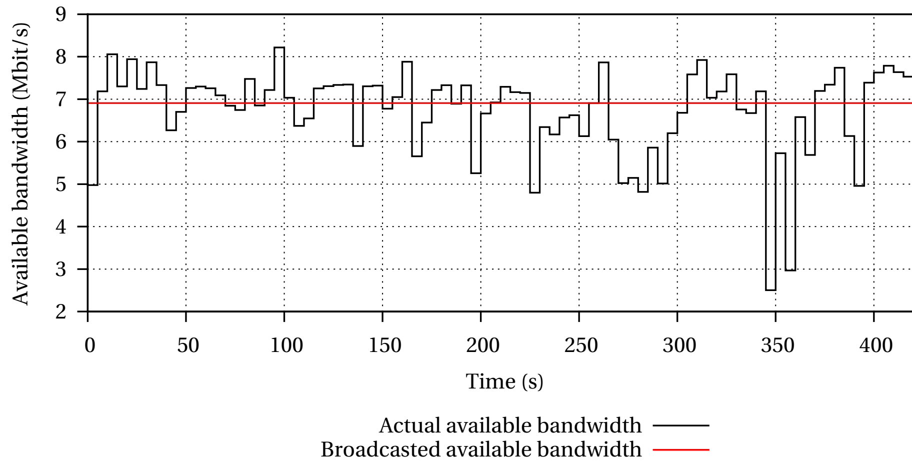


Figure 5.5 – Link-state updates in the 10 Mbit/s scenario, 5 seconds check interval. Uses an exponential sized classes policy ($b = 0.01$, $f = 1.5$) and a moving average over 10 measurements.

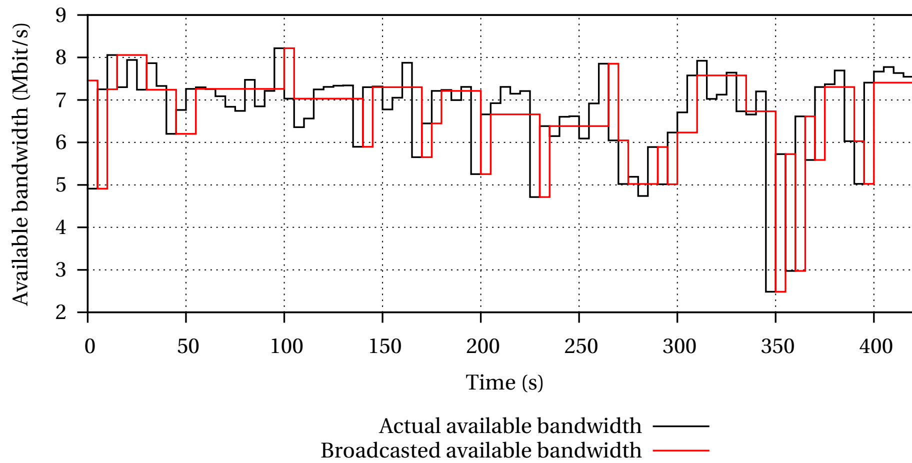


Figure 5.6 – Link-state updates in the 10 Mbit/s scenario, 5 seconds check interval. Uses an relative change policy ($t_r = 10\%$) and a hold-down timer of 5 seconds.

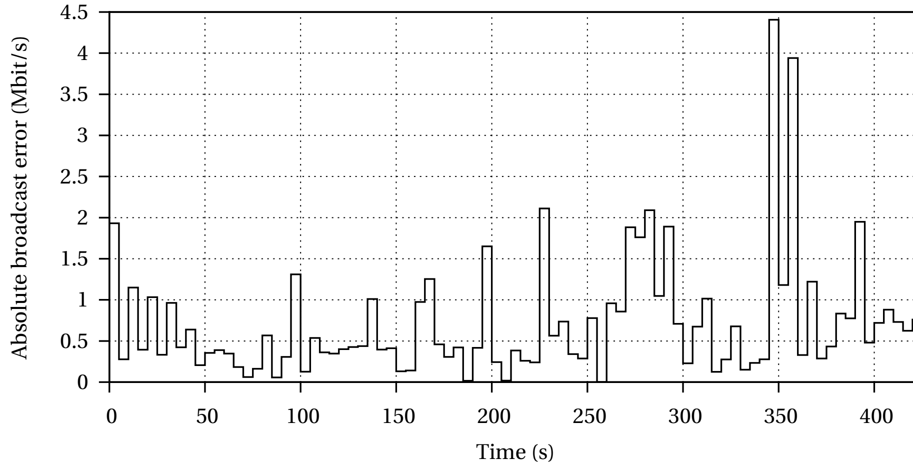


Figure 5.7 – Absolute error (difference between actual and broadcasted bandwidth) for Figure 5.5.

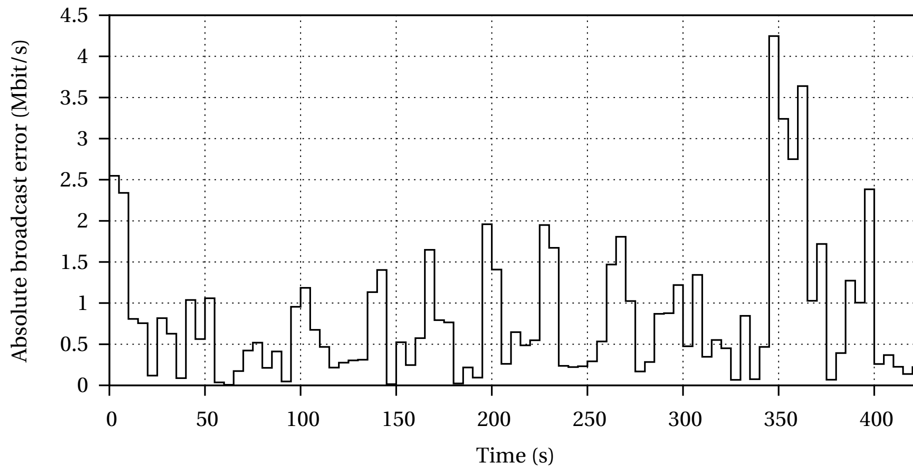


Figure 5.8 – Absolute error (difference between actual and broadcasted bandwidth) for Figure 5.6.

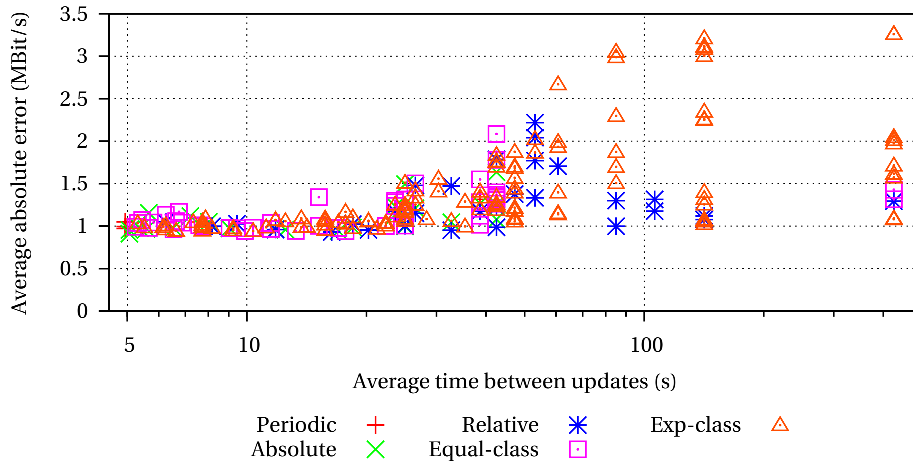


Figure 5.9 – Absolute error versus update rate for 10 Mbit/s, 1 second check interval scenario.

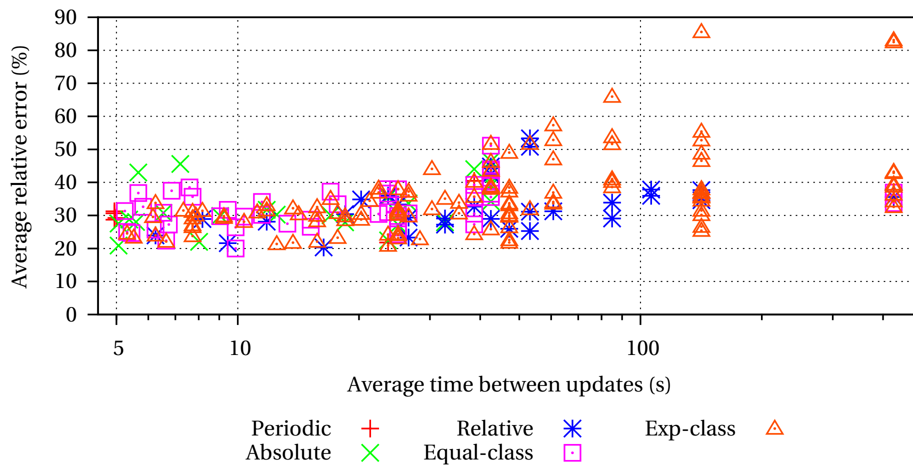


Figure 5.10 – Relative error versus update rate for 10 Mbit/s, 1 second check interval scenario.

decreases, from 142 seconds to 61 seconds (-57%). The average relative error increases a lot more than the average absolute error, from 21.4% to 34.1% (+59%). The reason that the update rate and the error increase, is twofold. First of all, the traffic will be more dynamic, because measurements are done on a one second instead of a five second scale. This makes it harder for the link-state update policies to accurately broadcast the available bandwidth. At the same time, the number of broadcasts is limited by the minimum hold-down timer of the OSPF protocol, which is 5 seconds. This means that the traffic will be more dynamic, but the algorithms will be unable to send more updates when necessary. We will further explore this problem in the next chapter.

If we look at the performance of the difference policies, it is hard to find a clear trend. All policies can perform well, if used with the right parameters. This can also be seen in the graphs: all policies are able to get close to the lowest average error, and can do so without generating excessive updates. This underlines the fact that it is more important to find the right parameters for the specific network traffic, than the right policy. Unfortunately, finding the right parameters is still a manual process. To automate this, we would have to test with more traffic patterns, model these traffic patterns in a statistical model, and try to find a correlation between the traffic patterns and the link-state update policy parameters. Even then, it it unsure if a correlation can be found. We will further discuss these future research opportunities in Section 7.2.

Six different timing policies were used. In three cases a hold-down timer was used, and in three cases a moving average was used. A comparison of the policies can be found in Table 5.2. We either used a hold-down timer, or a moving average, no combinations between those were made. Each data point had an equal weight in the moving average. This corresponds to the way a moving average was used in previous work [24]. Looking at the results, using a moving average instead of a hold-down timer leads to a large decline in the number of updates. The error does not seem to increase significantly. This is because a moving average damps sudden, temporary changes in available bandwidth, that would cause unnecessary updates to be sent when a hold-down timer is used.

5.4.2 100 Mbit/s scenario

The 100 Mbit/s scenario is a longer, more stable scenario. Figures 5.11 and 5.12 show the average absolute and relative error of the generated link-state updates in the 5 second check interval scenario. Compared with the previous measurements, the error is relatively lower. While the average absolute error is higher (7.5 Mbit/s versus 1.3 Mbit/s), the link speed also increased tenfold. The relative

Parameter	Hold-down timer 5	Hold-down timer 25	Hold-down timer 45	Moving average 5	Moving average 10	Moving average 15
Avg. absolute error (Mbit/s)	1.13	1.33	1.51	1.18	1.24	1.35
Avg. relative error (%)	23.0	25.7	32.4	27.1	27.6	30.7
Avg. time between updates (s)	39.1	60.1	66.4	91.5	170.1	183.0

Table 5.2 – Comparison of different timer policies in the 10 Mbit/s scenario. Both the 1 and 5 second check interval measurements are combined.

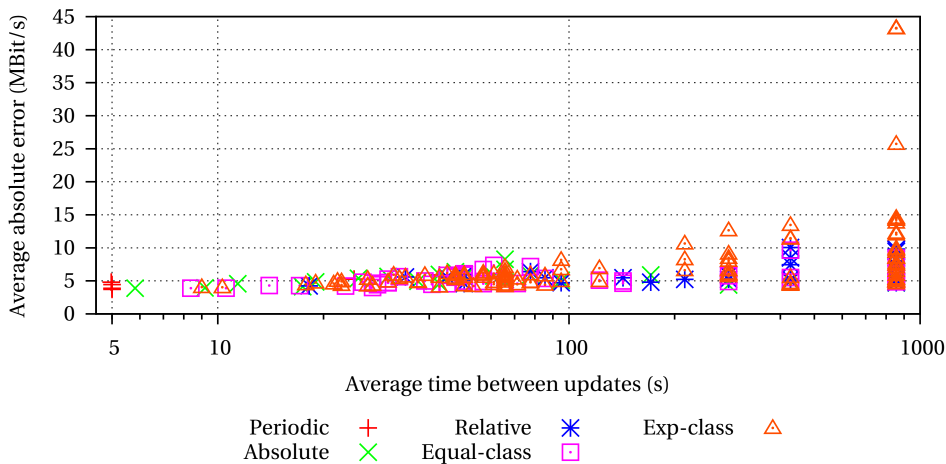


Figure 5.11 – Absolute error versus update rate for 100 Mbit/s, 5 second check interval scenario.

error is therefore lower (13.5% versus 21.4%). The lower error is due to the lower deviation in the traffic (the traffic is less bursty), as described in Section 5.3.

The distribution of the policies over the graph is comparable to the distribution in the 10 Mbit/s scenario. Most policies generate an update every 10 to 20 seconds. Some policies generate only one update during the entire test, and some policies update at the maximum rate (every 5 seconds).

The lowest average absolute error is found for the periodic policy that updates every 5 seconds, and is 3.7 Mbit/s. However, just as in the other tests, other policies come close. For instance, one of the policies manages an average absolute error of 4.2 Mbit/s, with an average time between the updates of 65.8 seconds.

Also in this case, we repeated the measurements with an 1 second check interval instead of a 5 second check interval. The results of these measurements can be found in Figures 5.13 and 5.14.

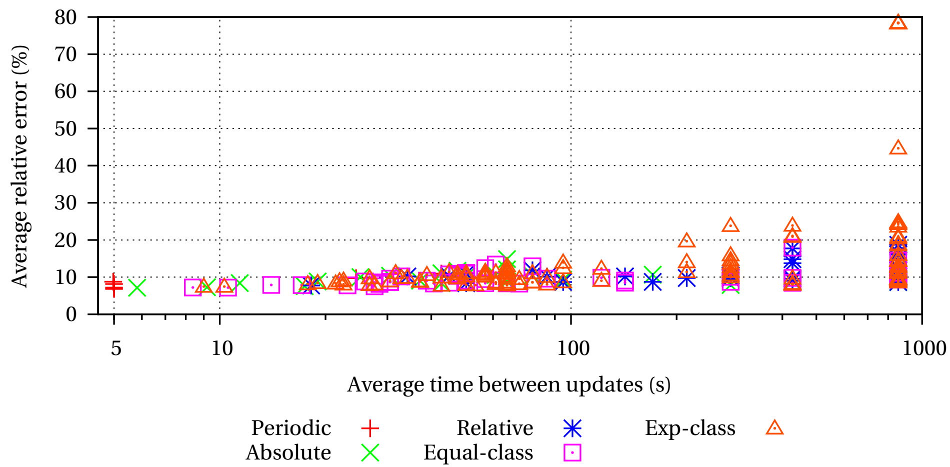


Figure 5.12 – Relative error versus update rate for 100 Mbit/s, 5 second check interval scenario.

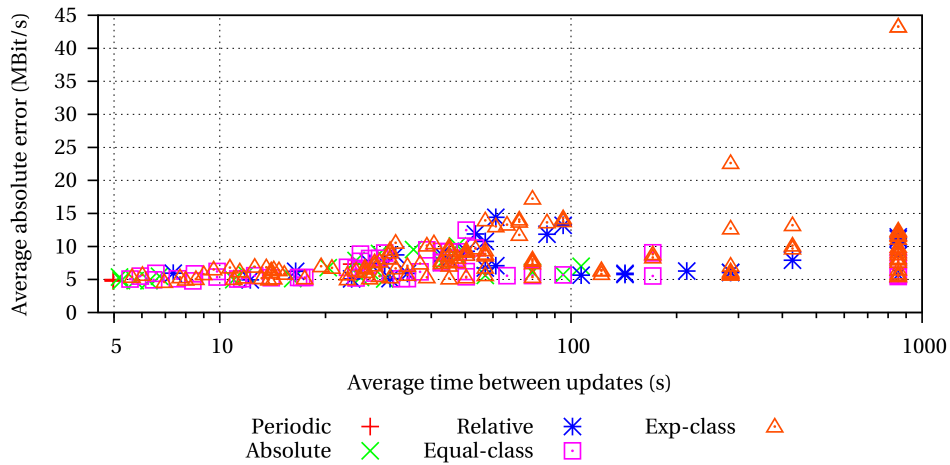


Figure 5.13 – Absolute error versus update rate for 100 Mbit/s, 1 second check interval scenario.

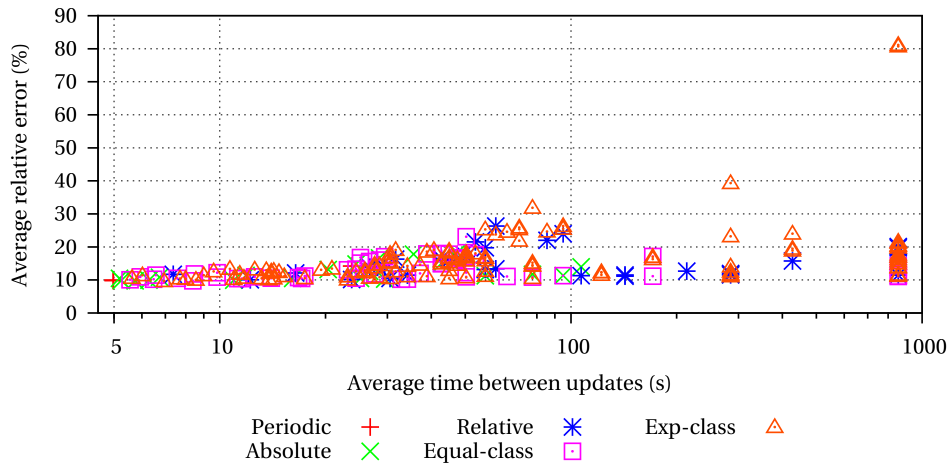


Figure 5.14 – Relative error versus update rate for 100 Mbit/s, 1 second check interval scenario.

	Hold-down timer			Moving average		
Parameter	5	25	45	5	10	15
Avg. absolute error (Mbit/s)	6.67	8.56	8.86	6.64	7.49	7.84
Avg. relative error (%)	12.4	15.8	16.4	12.5	14.1	14.6
Avg. time between updates (s)	157.9	132.5	166.8	251.6	325.5	398.9

Table 5.3 – Comparison of different timer policies in the 100 Mbit/s scenario. Both the 1 and 5 second check interval measurements are combined.

When we compare these results to the 5 second check interval results, we see the same differences as in the 10 Mbit/s scenario. The average absolute error increases by 5.5% to 7.9 Mbit/s. The lowest absolute average error increases to 4.6 Mbit/s. Also, the average time between updates decreases, from 325 to 153 seconds (-53%) and the average relative error increases from 13.5% to 15.1% (+12%).

Just as with the 10 Mbit/s scenario, it is hard to find a “best” policy. All policies can perform well, and the policies are spread across the graph. The actual policy performance seems to be determined by the parameters, instead of by the policy.

A comparison between the different timing parameters can be found in Table 5.3. Again, the number of updates is lowered significantly by using a moving average instead of a hold-down timer, while the average absolute error does not seem to be affected.

6
Discussion

In the previous chapter, the results of our performance comparison were presented. Some of these results ask for a more thorough explanation. Also, in previous work, similar research has been done, but in a different environment or with different parameters. We begin this chapter by drawing some conclusions from our presented results. Then we try to explain our results and conclusions, and we will compare and connect our results to related work where possible.

6.1 Summary of performance comparison

Looking back at the results, the following observations can be made:

- *High update error*: if we look at the best case average relative error, which uses a periodic policy in the 100 Mbit/s scenario, the error is still 6.8%, with a high standard deviation of 6.6 percentage points. Other scenarios show an even higher update error. With such an high update error, the information may be unusable for the routing process.

- *Lower check interval increases update error and rate*: comparing the 5 second check interval measurements with the 1 second check interval measurements, we clearly see that the lower check interval causes more update and an higher update error.

- *Sending more updates does not lead to better performance*: looking at the performance graphs (see Figures 5.3, 5.9, 5.11 and 5.13), it is immediately clear that for none of the scenarios more updates means a better performance. All scenarios have one or more update policies and parameter configurations that use only one update during the whole testrun, while their update error does not differ significantly from the best performing policy.

- *No performance difference between policies*: as stated in the last chapter, we were unable to find a “best” policy. Every policy tends to be tunable to the right update and error rate, by changing the parameters. In our case, this tuning was done by running each policy multiple times, using 5 to 8 different parameter configurations (see Table 5.1).

- *Lower average relative error for 100 Mbit/s scenario*: compared to the 10 Mbit/s, the average relative error is lower in the 100 Mbit/s scenario. The average absolute error increased.

- *Use of a moving average decreases update rate with same performance*: when a moving average is used instead of the usual hold-down timer, the number of updates goes down significantly. At the same time the update error does not seem to be influenced a lot.

6.2 Discussion of the results

The observations that we just described will now be further discussed. We will try to explain their root causes and will connect the observations to existing previous work.

6.2.1 High update error

The update error is relatively high when compared to previous studies. For instance, the 10 Mbit/s, 1 second check interval scenario shows a best case average relative error of 20.0%. The exact impact of this large error on routing quality depends on the QoS routing algorithm used, but it is clear that a lower error will yield better routing results.

Unfortunately, most previous work does not mention the absolute or relative update error achieved by the link-state update policies tested. Almost all work concerns the simulation of flow-based QoS router networks, and only takes the update rate and blocking probability of new connections in account. Lekovic and Van Mieghem do mention the relative update error in their work [24]. Their

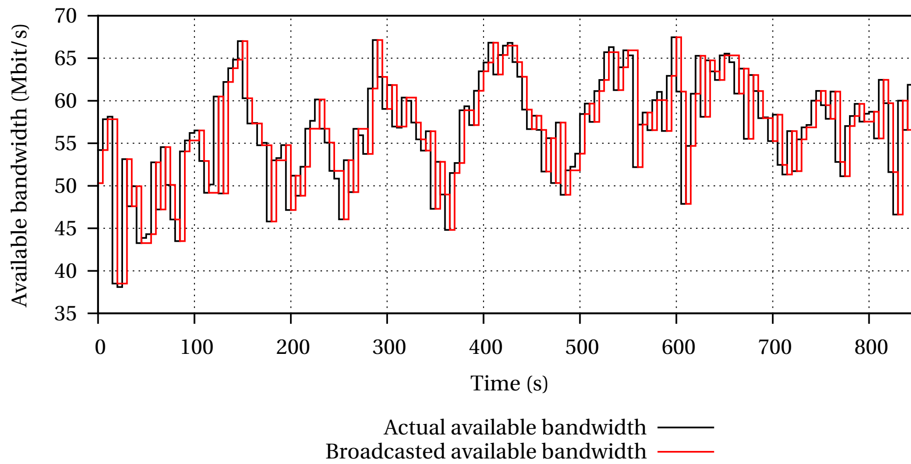


Figure 6.1 – Link-state updates in the 100 Mbit/s scenario, 5 seconds check interval. Uses an absolute change policy ($a = 1{,}000{,}000$ bits/s) and a hold-down timer of 5 seconds.

relative error rates vary from 1 to 3 percent, and are much lower than the update error rates we measured.

To explain the high update error, we take a look at Figure 6.1. This figure shows one of the better performing policies in the 100 Mbit/s, 5 second check interval scenario. From the graph can be seen that there is a delay between the actual available bandwidth change and the link-state update. The broadcasted available bandwidth value might look good at first sight, but Figure 6.2 reveals the large error that is present in the value. This is because our solution can only send a link-state update after the bandwidth has already changed, which introduces a delay the size of the check interval. The broadcasted available bandwidth changes frequently, due to the policy parameters, but it is nevertheless always running behind on the actual available bandwidth. Previous work is almost exclusively based on full QoS networks, where flow signaling and admission is used. Since in that case every flow is signaled before its start, our problem does not exist in those setups.

6.2.2 Lower check interval increases update error and rate

To decrease the discussed delay, we reran the test scenarios, this time with a 1 second check interval. Theoretically, this should decrease the gap from 5 seconds to 1 second and thus improve the performance. In practice, the performance did not improve but only got worse, due to the following two

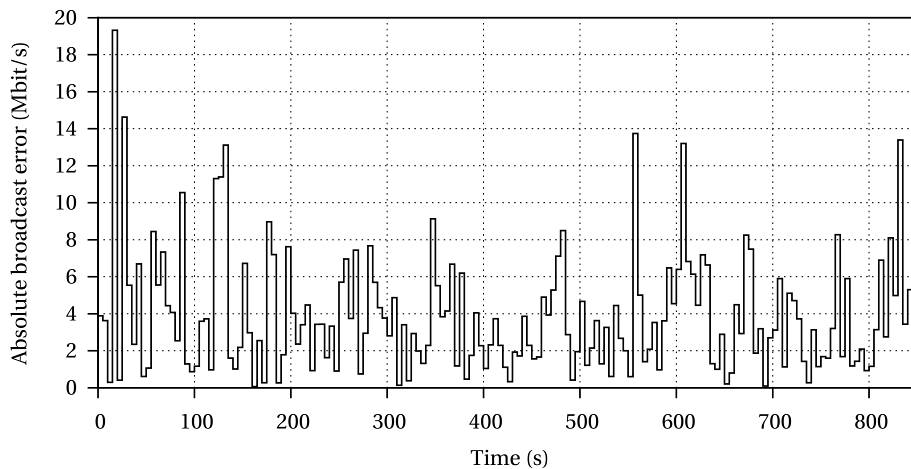


Figure 6.2 – Absolute error (difference between actual and broadcasted bandwidth) for Figure 6.1.

reasons. First of all, a lower check interval means more of the traffic dynamics are captured, so the measured traffic gets burstier. This makes it necessary to send link-state updates more often. Indeed, we saw an increase by roughly a factor of 2 in the number of updates. At the same time, the OSPF hold-down timer of 5 seconds limits the number of updates, and this timer cannot be lowered without violating the OSPF specifications. Because of this hold-down timer, the link-state update policies cannot send all updates necessary to capture the additional traffic dynamics, and performance gets worse. Future research could try to remove the minimum 5 second hold-down timer, but that would also require several changes to the default Quagga code and make the OSPF implementation incompatible with existing routers.

6.2.3 Sending more updates does not lead to better performance

From our results, it is clear that sending more updates does not necessarily improve the performance of the policies. For instance, the policy in Figure 5.5, which does not do any intermediate link-state updates, belongs to the best performing policies in the test. As mentioned earlier, there is a delay in our updates that causes a high update error. Probably, more updates would lead to better performance if this delay could somehow be removed, since the extra updates that get sent by some policies now always lag behind the actual traffic.

Previous work is inconclusive about whether more updates should lead to better performance. Most work uses blocking probability of new flows as a

measure for performance. Shaikh et al. have shown that policies that generate more updates decrease the amount of setup failures, but increase the amount of routing failures and do not necessarily lead to better performance [36]. In the research of Lekovic and Van Mieghem the update error does decrease slightly with more updates, but the hold-down timer and moving average values seem to affect the performance much more [24]. Yuan et al. do see a rather large increase in blocking probability when updates are sent less often [39].

6.2.4 No performance difference between policies

While discussing the results, we already noted that every link-state update policy can perform equally well by tuning the policy parameters. This is to be expected given our scenarios. Normally, the relative change and exponential class-based policies perform better than the absolute change and equal class-based policies, since they send out more updates when the available bandwidth on a link gets low [4, 6, 9]. Our scenarios are based on real-life traffic, where low available bandwidth leads to packet loss and other issues. For this reason, we do not see any low available bandwidth intervals in our tests and all policies can perform equally well. If the testbed would be extended to use the link-state update information in the QoS routing process, it would be possible to generate real-life scenarios with a high network load, to test the behavior of the link-state update policies under low available bandwidth conditions. This is left for future work.

Another reason for all policies performing equally well, is that a lot of different parameter configurations were tested. Most previous work tests only one or two policies, with 5 to 10 different parameter configurations. In our tests, 228 different combinations of link-state update policies and parameters were tested, which increases the chance that one of the combinations gives good results for a certain policy. This does not mean that a certain combination will perform good in all circumstances: more tests with more scenarios would be necessary to confirm that.

We should keep in mind our earlier mentioned large update error. The delay between the actual and broadcasted available bandwidth that causes this error, will probably also have its effect on the performance differences between the link-state update policies.

6.2.5 Lower average relative error for 100 Mbit/s scenario

The relative error in the 100 Mbit/s scenario is much lower than the relative error in the 10 Mbit/s scenario (13.5%-15.1% versus 21.4%-34.1%), while the absolute error is much higher (7.5-7.9 Mbit/s versus 1.3 Mbit/s). This can be explained

by the looking at the traffic pattern. The traffic of the 100 Mbit/s scenario is less bursty, because the chosen traffic trace is less dynamic than the traffic of the chosen video (compare Figure 5.1 and Figure 5.2, and see the explanation in Section 5.3). The standard deviation in available bandwidth is 7.49 Mbit/s for the 100 Mbit/s scenario, and 1.34 Mbit/s for the 10 Mbit/s. Relatively, this means the 10 Mbit/s scenario traffic is more bursty, and will therefore lead to a higher error, since the link-state update policies have 'more work to do'.

6.2.6 Use of a moving average decreases update rate with same performance

From Tables 5.2 and 5.3 can be clearly seen that a moving average decreases the update rate significantly. This is in line with the conclusions of Lekovic and Van Mieghem [24]. A difference is that their work is flow-based, with the moving average taken over a number of arriving flows, instead of over a certain time interval. However, this difference does not seem to lead to different results.

The use of a moving average decreases the update rate by damping sudden changes in available bandwidth. If, for instance, the available bandwidth changes for only one second, this will cause an update when a hold-down timer is used (and thus a larger error for the remainder of the hold-down period). The moving average will damp the sudden change, and suppress the update, which leads to less updates while not increasing the error.

Future work might consider the use of a weighted or exponential moving average, or combine the moving average with a hold-down timer.

7 Conclusions and Future Work

7.1 Conclusions

In this work, we implemented link-state update policies by extending the OSPF protocol and the QoS router testbed. First, an overview of link-state update policies was given. Then, a protocol was designed to broadcast the link-state information over OSPF. This protocol was implemented using the Quagga OSPF-API and a connection between the update daemon and the Network Controller of the QoS router testbed was made. The solution was then extensively tested to guarantee it was functioning correctly. Finally, a comparison between the available link-state update policies was made and the results were discussed.

Most previous work for link-state update policies has been done on policies for distributing available bandwidth information. We have built upon these results and have tried to compare the available policies. Unfortunately, our approach, where traffic is measured on physical link level instead of by accounting for admitted flows, introduces a large update error. Further research should prove if the information is nevertheless usable for QoS routing, or if the error is too large to provide reliable routing. In a certain way, our research underlines one of the big challenges in QoS routing research: most of the current research is focused on flow-based QoS, where all network activity is captured in flows, but not all applications will define their traffic in QoS flows (ie., there will probably always be non-QoS *cross-traffic*). This undermines the fundamentals of the current QoS research.

The “best” link-state update policy could not be found: it all depends on the traffic patterns on the network and the demands of the operator and his QoS routing algorithms. We did confirm the fact that hold-down timers cause excessive updates in the case of link-state update policies, and that the use of a moving average seems to provide better performance.

7.2 Future Work

There is a lot of work left for future research. Both some technical, implementation details still have to be solved, as well as some larger fundamental questions.

Since the actual link-state is now available in the Network Controller, the QoS router testbed can be extended to actually use this information in its routing algorithms. This would give the opportunity to evaluate link-state update policies on the same performance metric as previous work: flow blocking rate. Only available bandwidth link-state updates were implemented, while more QoS measures exist (e.g., delay, jitter). Since the protocol was designed with extensibility in mind, it should not be too difficult to add these measures. There is, however, no previous work on link-state update policies for these measures available.

Determining the right parameter configuration for the link-state update policies is still manual work, and the right configuration depends on the network traffic patterns and wishes of the operators. Future research should try to come up with a solution that, given the traffic pattern on a network, and the desired maximum error and maximum update rate, is able to provide the right link-state update policy and parameters. This is a hard problem, since it is not easy to describe the traffic pattern on a network and the exact influence of the various algorithm parameters is still unclear. By choosing one link-state update policy, testing that LSUP with several parameter configurations in a multitude of scenarios, and doing a factor analysis on the results, it should be possible to gain more insight in the correlation between the performance of an LSUP and its settings. Also it might prove useful to research additional moving average methods (e.g., weighted, exponential) and combining the use of a moving average with a hold-down timer.

On the technical side, our current implementation of link-state updates (and the QoS router testbed) lacks multiple OSPF area support and IPv6 support. These should be relatively easy to implement. Also, it is possible to show the actual link-state in a nice web interface, which can be made using relatively little effort, which would give administrators better real-time feedback on the state of the network.

The current QoS router testbed uses a centralized approach, with a central Network Controller, Service Manager and LSP Manager. To provide fault-tolerant routing as well as scalability, a distributed implementation will be necessary. A distributed solutions will also be necessary to provide inter-AS routing. Without inter-AS QoS routing, QoS on the Internet will always stay a utopian idea. Many concepts and protocols have been proposed in the past, but none have been able to generate widespread acceptance. Different opinions about the definition of QoS, and what a QoS routing protocol should offer the Internet, have slowed down the implementation of QoS on the Internet. Looking at the speed of the current switchover to IPv6, which is “merely” about a simple addressing scheme, and what needs to be done to implement QoS in the Internet, we can only conclude that full QoS support in the Internet is a though problem that might never be solved.

Bibliography

[1] R. Aggarwal and K. Kompella. Advertising a Router's Local Addresses in OSPF Traffic Engineering (TE) Extensions. RFC 5786 (Proposed Standard), Mar. 2010.

[2] ANSI and CBEMA. *American National Standard for information systems: programming language: C: ANSI X3.159-1989*. American National Standards Institute, Dec. 1989.

[3] G. Apostolopoulos, R. Guérin, S. Kamat, and S. Tripathi. Quality of service based routing: A performance perspective. *ACM SIGCOMM Computer Communication Review*, 28(4):17–28, 1998.

[4] G. Apostolopoulos, R. Guerin, S. Kamat, and S. Tripathi. Improving QoS routing performance under inaccurate link state information. *Proceedings of the 16th International Teletraffic Congress*, pages 1351–1362, 1999.

[5] G. Apostolopoulos, S. Kama, D. Williams, R. Guerin, A. Orda, and T. Przygienda. QoS Routing Mechanisms and OSPF Extensions. RFC 2676 (Experimental), Aug. 1999.

[6] A. Ariza, E. Casilari, and F. Sandoval. Strategies for updating link states in QoS routers. *Electronics Letters*, 36(20):1749–1750, 2000.

[7] S. Avallone. *A novel approach to Traffic Engineer QoS-aware networks*. PhD thesis, Università degli studi di Napoli Federico I, Sept. 2004.

[8] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow. RSVP-TE: Extensions to RSVP for LSP Tunnels. RFC 3209 (Proposed Standard), Dec. 2001. Updated by RFCs 3936, 4420, 4874, 5151, 5420, 5711.

[9] A. Basu and J. Riecke. Stability issues in OSPF routing. *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, Aug. 2001.

[10] L. Berger, I. Bryskin, A. Zinin, and R. Coltun. The OSPF Opaque LSA Option. RFC 5250 (Proposed Standard), July 2008.

[11] A. Botta, A. Dainotti, and A. Pescapè. Multi-protocol and multi-platform traffic generation and measurement. In *INFOCOM 2007 DEMO Session*, Anchorage (Alaska, USA), May 2007.

[12] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. RFC 1633 (Informational), June 1994.

[13] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. RFC 2205 (Proposed Standard), Sept. 1997. Updated by RFCs 2750, 3936, 4495.

[14] J. Case, M. Fedor, M. Schoffstall, and J. Davin. Simple Network Management Protocol (SNMP). RFC 1157 (Historic), May 1990.

[15] C. Demichelis and P. Chimento. IP Packet Delay Variation Metric for IP Performance Metrics (IPPM). RFC 3393 (Proposed Standard), Nov. 2002.

[16] B. Fu, F. Kuipers, and P. Van Mieghem. To update network state or not? *The 4th international telecommunication networking workshop on QoS in multiservice IP networks*, Feb. 2008.

[17] I. Gallagher, M. Robinson, A. Smith, S. Semnani, and J. Mackenzie. Multi-protocol label switching as the basis for a converged core network. *BT Technology Journal*, 22(2):95–103, 2004.

[18] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, Aug. 1985.

[19] International Telecommunication Union. ITU-T Recommendation G.114: One-way transmission time. Technical report, International Telecommunication Union, 1993.

[20] K. Ishiguro, V. Manral, A. Davey, and A. Lindem. Traffic Engineering Extensions to OSPF Version 3. RFC 5329 (Proposed Standard), Sept. 2008.

[21] B. Jamoussi, L. Andersson, R. Callon, R. Dantu, L. Wu, P. Doolan, T. Worster, N. Feldman, A. Fredette, M. Girish, E. Gray, J. Heinanen, T. Kilty, and A. Malis. Constraint-Based LSP Setup using LDP. RFC 3212 (Proposed Standard), Jan. 2002. Updated by RFC 3468.

[22] T. Karagiannis, M. Molle, and M. Faloutsos. Long-range dependence: Ten years of internet traffic modeling. *Internet Computing, IEEE*, 8(5):57 – 64, 2004.

[23] D. Katz, K. Kompella, and D. Yeung. Traffic Engineering (TE) Extensions to OSPF Version 2. RFC 3630 (Proposed Standard), Sept. 2003. Updated by RFCs 4203, 5786.

[24] B. Lekovic and P. Van Mieghem. Link state update policies for quality of service routing. *Eighth IEEE Symposium on Communications and Vehicular Technology in the Benelux (SCVT2001)*, 2001.

[25] T. Li and H. Smit. IS-IS Extensions for Traffic Engineering. RFC 5305 (Proposed Standard), Oct. 2008. Updated by RFC 5307.

[26] A. Lindem, N. Shen, J. Vasseur, R. Aggarwal, and S. Shaffer. Extensions to OSPF for Advertising Optional Router Capabilities. RFC 4970 (Proposed Standard), July 2007.

[27] A. Meddeb. Internet QoS: Pieces of the puzzle. *Communications Magazine, IEEE*, 48(1):86–94, 2010.

[28] J. Moy. OSPF Version 2. RFC 1583 (Draft Standard), Mar. 1994. Obsoleted by RFC 2178.

[29] J. Moy. OSPF Version 2. RFC 2328 (Standard), Apr. 1998. Updated by RFC 5709.

[30] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474 (Proposed Standard), Dec. 1998. Updated by RFCs 3168, 3260.

[31] D. Oran. OSI IS-IS Intra-domain Routing Protocol. RFC 1142 (Informational), Feb. 1990.

[32] J. Postel. Internet Protocol. RFC 791 (Standard), Sept. 1981. Updated by RFC 1349.

[33] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. RFC 3031 (Proposed Standard), Jan. 2001.

[34] C. Semeria. Supporting differentiated service classes in large IP networks. Technical report, Juniper Networks, Jan. 2000.

[35] A. Shaikh and A. Greenberg. Experience in black-box OSPF measurement. Sept. 2001.

[36] A. Shaikh, J. Rexford, and K. Shin. Evaluating the impact of stale link state on quality-of-service routing. *IEEE/ACM Transactions on Networking*, 9(2):162–176, Jan. 2001.

[37] P. Srisuresh and P. Joseph. OSPF-xTE: Experimental Extension to OSPF for Traffic Engineering. RFC 4973 (Experimental), July 2007.

[38] P. Van Mieghem, H. De Neve, and F. Kuipers. Hop-by-hop quality of service routing. *Computer Networks*, Jan. 2001.

[39] X. Yuan, W. Zheng, and S. Ding. A comparative study of QoS routing schemes that tolerate imprecise state information. In *In Proc. Computer Communications and Networks*, pages 230–235, 2002.