UNIVERSITY OF OSLO
Department of Informatics

QoS related admission control for Web services

Master thesis

Øyvind Kolbu

February 8, 2011

Abstract

In a military tactical network bandwidth is often scarce. Web services lack a standardized approach to provide Quality of Service. In this thesis a broker which performs access control on bandwidth is created to provide a role based admission control. The goal is to achieve high client satisfaction where the client's role is considered important.

Acknowledgments

First and foremost, a huge thanks to my supervisors at the Norwegian Defence Research Establishment (FFI), Frank Trethan Johnsen and Trude Hafsøe, for their continuous support and encouragement. Secondly, I would like to thank Pål Halvorsen at the Simula Research Laboratory for his guidance. Finally, I would like thank my family and friends, for their motivation while I wrote this thesis.

Øyvind Kolbu
Kjeller, February 8, 2011

Contents

1. Introduction

> War is a product of its age. The tools and tactics of how we fight have always evolved along with technology. We are posed to continue this trend.
> – Dr. David S. Alberts on *Network Centric Warfare*

In 1999, the United States envisioned a military where units can interconnect to share information and resources, a concept they named «Network Centric Warfare» [1]. Since then, this concept has been further researched in Norway and Sweden, as «Network Based Defence», in the United Kingdom as «Network Enabled Capabilities» and later in the North Atlantic Treaty Organization (NATO), as NATO Network Enabled Capabilities (NNEC).

A huge challenge is to find the technology to bind all new and existing military equipment together allowing them to interconnect. In this respect the military has looked for possible existing civilian solutions. Web service technology has become popular in civilian networks, both for internal use and for public services on the Internet. The technology is based on open standards which makes it easy to build and deploy services. The success in civilian use has lead military researchers into attempting to employ Web services in military networks. These networks are often complex, slow and consist of heterogeneous technologies. Therefore, using Web services can be a complex endeavor. This thesis will focus on using Web services in military networks in order to access resources.

1.1. NATO Network Enabled Capabilities

As seen in Figure 1.1, military networks can be complex and consist of different operational levels, contain both mobile and stationary nodes, and use a heterogeneous set of communication technologies. It is a goal to be able to buy commercial off-the-shelf (COTS) equipment, and the NATO policy is to use civilian standards where possible. If these standards do not meet the specialized military requirements, NATO can develop own standards, known as NATO standardized agreements (STANAGs). In the «NNEC Feasibility Study» (NNEC FS) [65, and ref. therein], NATO has found that a Service Oriented Architecture (SOA) implemented with Web Services is a viable way to standardize information sharing and services within the alliance, using IP as the common network protocol. The study does not specify how each nation should build their information infrastructure, but instead focuses on the need for future interoperability between nations and thus creating a *federation of systems*, which is an assembly of systems, where each sub-system is under autonomous control.

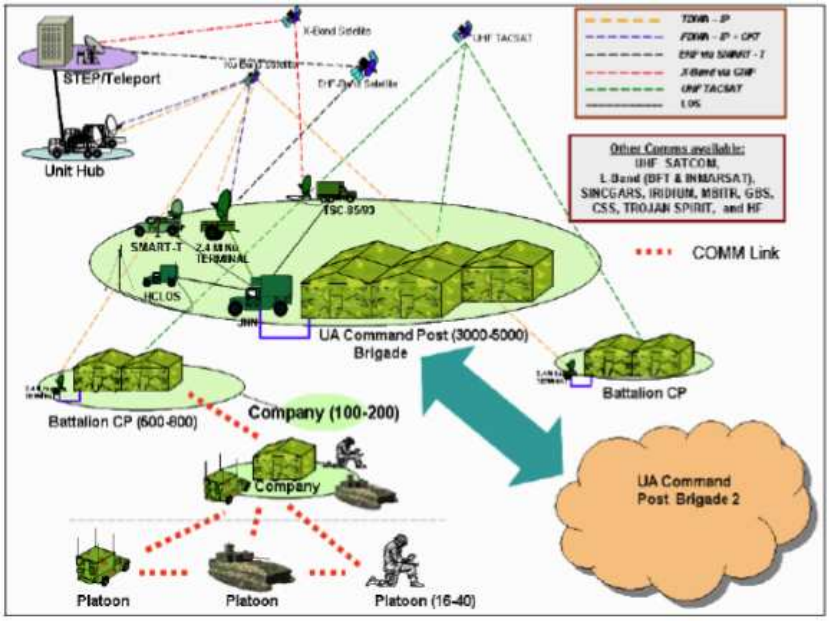

Figure 1.1.: Operational network, from [17].

1.2. Service Oriented Architecture

In [46] SOA is defined as:

> Service Oriented Architecture is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.

A capability in a network is in a SOA known as a possible *service*, e.g., a web camera, a GPS or a software service, and [46] also defines a *service* as:

> A service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description. A service is provided by an entity – the service provider – for use by others, but the eventual consumers of the service may not be known to the service provider and may demonstrate uses of the service beyond the scope originally conceived by the provider.

SOA services can be implemented using a wide variety of technologies, but the most common one is Web services. This is also the technology NATO has chosen to use for interoperability between nations. Web service technology is based on a set of open

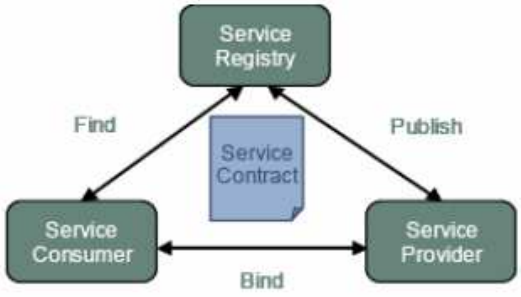

Figure 1.2.: Elements of a Web service architecture [35].

standards, where the Web Services Description Language (WSDL) [10] is used to create a description of a service, using Extensible Markup Language (XML) [8]. Request and responses are sent using SOAP [25] messages transported by HTTP [18] over TCP [57].

A service is of no use without anyone using it, and the counterpart to the service provider is therefore the *consumer*. If a consumer wants to use a service, it must first find it, and that is usually done through a *registry*, in which the creator of the service has registered it. Figure 1.2 shows the relationship between service, consumer and registry, with a WSDL based *service contract*.

There are many ways to define Web Services, but the World Wide Web Consortium (W3C) has the following definition [26] which this thesis will use:

> A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

XML is designed to be both human readable and machine processable, which means that is is rather verbose. Because of this, SOA realized by Web Services generate a significant amount of network traffic. This means that Web Services, in their basic form, are not suited for military networks. In [44, 45, 64], it has been found that this can be vastly improved by using compression, proxies and filtering, making the technology a viable option for military communication.

The registry is not mandatory. Therefore, consumers with prior knowledge of the service can *invoke* it directly. A registry can either be central or distributed, depending of needs and network topology, or it can be a specially tailored mechanism. However, discovery is not the focus of this thesis, and the interested reader should see [36] for a thorough discussion about service discovery, while [33] discusses challenges with service discovery across heterogeneous military networks.

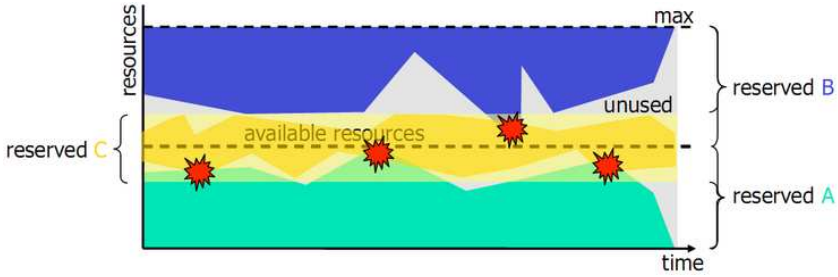

Figure 1.3.: Statistical guaranteed QoS, from [23].

1.3. Quality of Service – QoS

In [69], QoS is defined as "the set of those quantitative and qualitative characteristics of a distributed multimedia system necessary to achieve the required functionality of an application".

In the NNEC FS, QoS is listed as a requirement, but when the study was written no QoS standard existed for Web services, which is also the case today. QoS is in the NNEC FS not only listed as any requirement, but after interoperability as the second most important factor for the NNEC concept to succeed. Because many services in a military network can provide pictures and other multimedia, the above definition is applicable for military use of Web services as well.

Military network resources are scarce, and must therefore be better shared than by using traditional *best effort*, as used in civilian networks. QoS is implemented by giving *priority* to an *entity*, e.g., a user, host or a service, possibly at the expense of another entity.

Priority is usually differentiated in the following three classes [66]:

- *Best effort* is the simplest, as one does nothing and hopefully the protocols are fair. No guarantees can be met which can lead to unreliable services.

- *Deterministic guaranteed* priority reserves the requested resource for future use and can therefore satisfy demands. The downside is that reservation does not imply neither constant use nor that the reservation is just big enough, and therefore it will waste resources.

- *Statistical guaranteed* priority is similar to *deterministic*, but can use statistical methods to achieve a higher work load, though this may violate QoS from time to time, as seen in Figure 1.3.

In this thesis the goal is to provide statistical guaranteed QoS with preemption to ensure the success of high priority client requests.

1.3.1. Client satisfaction

QoS encompasses many aspects, and can be seen from many perspectives. In [38] three perspectives are defined: network QoS, Quality of Availability (QoA) and Quality of Perception (QoP). QoA deals with the availability of a service, while QoP deals with a client's experience of a service. One aspect of QoP is that a client gets an answer within a reasonable time. In this thesis client satisfaction is defined by reaching a given deadline.

1.4. Thesis scope

To provide an integrated QoS platform for NNEC operations, QoS must be present in all aspects of Web Services. A complete solution would require significantly more work than a thesis. Therefore, this thesis will focus on creating an admission control broker for Web Services, which performs admission control of the available bandwidth[1], because bandwidth is the most limited resource in a military network.

1.4.1. Role-based QoS

In a military QoS setting, your job or the *role* you have is more important than who you are. Furthermore, the combination of role and service will yield a priority level, e.g., a low ranking person might get higher priority than a high ranking officer because the service is more *relevant* for the lower ranked person's role. Thus, a user may experience differentiated quality, depending on which service it invokes, as the combination of the user's role and service can yield different priorities.

When using role-based QoS, the first and most obvious challenge is how to assign priorities, and which roles are needed and how to assign roles to users, i.e., who are going to get first rate services while others must patiently wait in line?

In addition to roles, other variables, like the proximity to the service, can further be taken into account when determining priority, but using such variables are beyond the scope of this thesis.

1.4.2. Service descriptions

When creating a QoS-enabled Web service, the standard Web service description must be extended with information about the QoS properties of the service.

Typically, one can provide two services for the same web camera, only with different image resolution, and for instance only allow local and high priority users to get the highest solution.

[1] In the context of radio systems, the term *Data rate* is used to describe the throughput in terms of bits per second. When describing computer networks, and in this thesis, *bandwidth* is used interchangeably with data rate and throughput. In signal processing on the other hand, bandwidth represents width of frequencies and is measured in hertz.

As roles and services can change over time, the registry should not only hold services, but also roles, and be updated when needed. That process is outside the scope of this thesis and it is assumed that the end-user equipment can send descriptions of itself.

1.4.3. Service invocation

Adding QoS support to Web services introduces a number of challenges related to service invocation: What shall be done when an invocation with a higher priority arrives and all bandwidth is taken? Who shall be the first to go? Round-robin at each layer, starting bottom up with *best effort*? This is obviously not something the client neither needs to know nor should be able to know, but is a set of policies which the broker must enforce. The answers to these questions will be addressed in the design and testing of the admission control broker presented in this thesis.

In this thesis the services are given, but the client must compare its own capabilities or requirements with the available services' QoS descriptions and figure out which one to invoke, or maybe implement fail back options if one service is saturated.

The COTS clients will use HTTP over TCP, but this can be optimized by using non-standardized solutions for military networks [44].

1.4.4. Service discovery

Discovery can be cumbersome in military networks, but good experimental solutions exist [33], and it is considered outside the scope of this thesis. It assumed that all service descriptions are distributed to all clients and the broker.

1.4.5. Network information

Typically, users want to get a higher quality version if they can, though the network will often limit them. Accurate network information is therefore important, in order to give the most important users priority, and at the same time accept as many non-priority, *best effort*, users as possible.

Feedback from the network is important to realize QoS, because then an application can adapt to the conditions. This is unsolved for Web services and an important aspect to cover.

1.5. Admission control

In contrast with the normal Internet, where demand for more bandwidth and guarantees have been met with over provisioning, military networks often do not have that option. In radio based networks, which deployed military networks often are, radio uses a fixed frequency specter, and thus limiting the number of simultaneous radios which can be used.

To manage the available resources and to make sure high priority traffic gets precedence over low priority traffic, a local bandwidth broker can be used to make

sure that applications will correctly priority tag their traffic according to the current network state.

The broker must know the current network utilization and potentially future allocations, as well as information about the network topology.

There exists no standardized QoS framework for Web Services. This thesis will therefore implement a partial experiential prototype as a proof of concept.

1.6. Limitations

As this thesis was written at and for the Norwegian Defence Research Establishment (FFI), some restrictions were made to the use of military sources to keep the thesis unclassified and freely distributable. Therefore no real usage patterns of services are used, and all mappings between services, roles and priorities are significantly simplified. The upper bandwidth limitations of some frequently used military radios are shown in Table 2.1, but frequencies, ranges and other properties can not be disclosed.

NATO stresses interoperability, thus only mature Web service standards are considered.

Military networks have strict security policies and requirements. Even though some civilian security standards exist for Web services, current policies mandate either link or IP cryptography. It is therefore henceforth assumed that the required security is in place, and thus plain Web services can be used, with security provided in lower layers.

1.7. Research method

In [15], three main scientific approaches for the computer science discipline are defined: theory, abstraction, and design. These approaches should be seen as equally important. The approaches all follow an iterative process, but the process steps differ from approach to approach. This thesis uses the design approach, because it is the most suitable one in the development of a prototype, rather than an abstract framework. A design approach can be realized by using an engineering process. The engineering process consists of four stages: 1) Perform requirements analysis, 2) derive a specification based on the requirements, 3) design and implement the system, 4) test the system. In the engineering approach the hypothesis is that the system fulfills the specification and thereby meets the requirements.

1.8. Research claims

NNEC requires QoS for Web services, but no standards exists. To enforce admission control of often very limited bandwidth, a role based QoS with a broker which uses network information can be used. In this thesis a QoS-aware admission control broker, which supports prioritization and preemption is examined. The goal is to obtain high

client satisfaction for the most critical roles. This thesis is built around two research claims that are proved through the thesis:

1. *QoS admission control can be implemented in a broker.*
2. *A role based QoS combined with a broker will lead to higher client satisfaction.*

1.9. Thesis outline

The remainder of this thesis is organized as follows: **Chapter 2** discusses the background of Quality of Service and Web Services, which covers the two first steps of the research method. **Chapter 3** describes the design of the broker mechanism and **Chapter 4** goes through the details of the implementation and evaluates the results. These two chapters cover the third and fourth step. **Chapter 5** concludes the thesis. **Chapter 6** discusses aspects which need further research.

2. Background

2.1. Network layer QoS

QoS has historically been implemented on the network layer, or if using Multiprotocol Label Switching (MPLS) [62], between the network and data link layer. Typically network layer QoS can attempt to optimize the network usage based on a number of different properties, with the most common being:

- bit rate
- packet loss
- delay, typically for interactive services as IP telephone
- jitter, same as above, but also for streaming

In military networks the bit rate is low, because they are more focused on range and resilience against jamming. Packet loss can occur due to the nature of wireless networks, but can be mitigated using heavy forward error correction. High delay is the norm due to low bandwidth, and even more so because some radios, e.g., HF-based, can be slow at changing the radio from reception mode to sending, and vice versa. This thesis will focus on bit rate as a resource.

2.2. Network QoS infrastructure

QoS in a military wired network is often enforced using a combination of the well established IETF standards Integrated Services (IntServ) [6] and Differentiated Services (DiffServ) [5, 24]. The two models are fundamentally different, as IntServ uses end-to-end resource reservations, while DiffServ will differentiate the traffic per hop. A further discussion on IntServ and DiffServ in a military setting can be found in [41].

2.2.1. IntServ

IntServ requires end-to-end resource reservations for distinct flows using the Resource ReSerVation Protocol (RSVP) [7, 71]. RSVP requires that all routers on the path between the two corresponding nodes accepts the resource reservation before IntServ can accept the traffic, and thereby essentially creating a circuit-switched network on top of the packet-switched network. All the routers must perform resource management and continuously monitor the bandwidth usage. If the topology or link speed changes,

which often happens in a Mobile Ad-Hoc Network (MANET) [13], then a new RSVP procedure must be performed for each current connection. On slow radio links, the signaling overhead, not only on network changes, but as well on connection setup, maintenance and tear down, can lead to too much of the available bandwidth being used.

2.2.2. DiffServ

DiffServ maps the type-of-service IP header bits to traffic classes. These classes define a deterministic Per Hop Behavior (PHB) and allows DiffServ to prioritize IP packets accordingly. DiffServ is purely PHB-based and performs neither admission control nor any resource management.

In tactical networks PHB is a huge advantage because it does not require any signaling across the network. Furthermore, when changes in topology occurs, no new setup phase is needed, but either one can keep using the existing PHB-table or change to a new one, if needed.

Several RFCs cover DiffServ and PHB, where the most important ones are: RFC 2474 (Definition of the Differentiated Services Field in the IPv4 and IPv6 Headers) [50], RFC 2475 (An Architecture for Differentiated Service) [5], RFC 2597 (Assured Forwarding PHB Group) [29], RFC 3140 (Per Hop Behavior Identification Codes) [4], RFC 3246 (An Expedited Forwarding PHB) [14], RFC 3260 (New Terminology and Clarifications for DiffServ) [24], and RFC4594 (Configuration Guidelines for DiffServ Service Classes) [2].

2.3. Application

Extending QoS up to higher layers, as the application gives more granularity on who and what to prioritize. For example, if a computer is used by multiple persons during a day which have distinct roles, then on the network layer traffic would either be prioritized or not, while an application QoS model can filter on the actual person logged in, the service requested, physical location, proximity, etc.

2.4. Tactical router

The military uses various radio technologies, each with different capabilities regarding range, data rate and frequencies, to support the often agile behavior of units. Some common radios are listen in Table 2.1. Traditionally, the tactical networks were isolated in disjunct Combat Net Radio (CNR) networks, where the radios provided all the functionality on all network layers to communicate with other CNRs within the network, but without any possibility to reach other networks. Protocols used within each CNR network were often non-standard, thus further hindering intercommunication.

The interoperability requirement which NNEC imposes in tactical environments, requires more integration. This obviously adds challenges as administration, admission

Link type	Max bandwidth (Kbit/s)	Expected bandwidth (Kbit/s)
High Band UHF (HB-UHF)	300.0	300.0
Sub Network relay (SNR)	64.0	64.0
Personal role radio (PRR)	38.4	38.4
STANAG 5066 / 4538	9.6	0.15 - 9.6
Satellite (unstabilized antenna)	2.4	2.4
Multi role radio (MRR) / Light field radio (LFR)	2.4	1.0
Universal Improved Data Modem (UIDM)	16.0	16.0

Table 2.1.: Military radios and their max and expected bandwidths, from [34].

control and routing of traffic flows. In [41], an IP-based network architecture in CNR networks and between them is described, in order to enable tactical networks for use in NNEC operations. The Norwegian Army Transformation and Doctrine Command (TRADOK), together with FFI and industrial partners have therefore developed a router which uses IP as an universal carrier in all the tactical networks [28]. These routers are called Intelligent Tactical IP Routers (ITR). A prototype from Thales, who later won the development contract, can be seen in Figure 2.1. Thales' ITR runs a customized version of Vyatta [70], an open source network operating system with a Linux kernel.

2.4.1. Intelligent networks

To provide reliable network connectivity to users operating in the field, despite difficulties as varying terrain and jamming attempts, the network must support multiple transmission technologies and routing paths. Long range low throughput links for reach back links to head quarters and short range high throughput network for local communication must be supported side by side, and be as transparent for the user as possible. This requires routing and network information to be spread, and to differentiate traffic according to the links' capabilities.

The ITR uses DiffServ to differentiate traffic. Routing information and other traffic which is used to keep the network up and running have top priority, while the other traffic classes can be configured as needed.

2.4.2. Routing in mobile networks

While traffic flows can be shaped with DiffServ, as in any other civilian network, routing in a mobile environment is much more complex than in a wired network. Multiple ITRs, either some/all stationary, will form a MANET, which requires other routing protocols

Figure 2.1.: The Intelligent Tactical IP Router designed by Thales is based on a ruggedized PC/104 platform offering four Ethernet ports, two serial ports, one X.25 port and two USB ports [28].

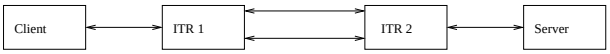

Figure 2.2.: Two ITRs each with two carriers.

than plain Open Shortest Path First (OSPF) [49] or Routing Information Protocol [47], due to MANETs being much less reliable than a regular network.

The ITR supports multiple unicast MANET routing protocols, most importantly: Optimized Link State Routing Protocol (OLSR) [12], wireless extension of OSPF [52], Ad hoc On-Demand Distance Vector (AODV) [56] and OSPF-MT (Multi Topology) [58]. OLSR and OSPF are good for networks which have a lot of communication, as it will measure and calculate routing information all the time. This is in contrast to AODV, which will only lookup paths to a destination when needed, and thus saving bandwidth. An AODV lookup may take some time, and therefore OLSR or OSPF are often preferred in a military setting, to ensure that critical traffic will be handled as soon as possible. The ITR also supports various types of multicast routing protocols, but only unicast routing will be used in this thesis, because COTS Web Services use unicast SOAP over HTTP over TCP.

2.4.3. Military specific limitations

Military tactical networks often form *disadvantaged grids*, which [22] defines as:

> Disadvantaged grids are characterized by low bandwidth, variable throughput, unreliable connectivity, and energy constraints imposed by the wireless communications grid that links the nodes.

Furthermore the radio equipment is expected to last 10-15 years, and therefore low bandwidth radios will be common for a long time.

COTS Web Service clients use SOAP over HTTP over TCP, and therefore require a direct end-to-end connection between the client and the server. Experiments have shown that TCP in Windows XP stops working when the bandwidth is less than 400 bits per second, and likewise with UDP when reaching 300 bits per second [64], which are not considered very low speeds in tactical networks. 40 bits per second per user is not uncommon, rendering TCP not usable in most tactical wireless environments [21]. TCP is connection-oriented, therefore given the unreliable nature of a disadvantaged grid, low link utilization will occur due to the congestion control algorithm compensating for packet loss and high latency. Likewise, HTTP also has problems in disadvantaged grids due to it being synchronous. Thus when a SOAP request is sent over HTTP, the connection is kept open until a reply is received. Traversing multiple radios yields a high possibility of a connection timeout, and then force a client to restart the request.

2.4.4. DSproxy

To improve the reliability and performance of Web Services in disadvantaged grids, FFI has developed a SOAP proxy, the Delay- and Disruption-Tolerant SOAP Proxy (DSproxy). The DSproxy is able to cache responses, provide XML compression and supports HTTP and TCP, store-and-forward, and is done without modifying any SOAP messages [64]. For the best result with COTS clients, a proxy should be placed as close to the client as possible, in order to minimize the usage of HTTP and TCP [45].

2.5. Brokers

QoS brokers are not something new, as already in 1994, IntServ with RSVP existed as an example of a bandwidth broker. The new aspect with the broker proposed in this thesis is that it provides feedback from the network to the application, and that this is done for Web Services. See Chapter 3 for details of the design, and Chapter 4 for the implementation and evaluation.

2.6. QoS in Web Services

Standards are important for Web Services, and there is an ongoing attempt to standardize QoS-information for Web Services [40]. This has not been completed, and therefore it is important to consider what been done of experimental solutions.

WS-QoS [67] is a solution which solves a specific part of the QoS-puzzle. The author performs a mapping from proprietary QoS-information on the application layer to traffic classes on the network layer, with respectively DiffServ and UMTS used for wired and wireless networks.

Even if experimental solutions are used, it is important that the choices which are made are not incompatible with existing standards. This implies that one can not

change for instance the description of Web Services, by adding QoS-information in the WSDL. This is because the standard not supporting it. However, some experimental solutions do employ this approach [72, 65], but that brings in requirements for proprietary clients and services which can understand the extended service descriptions. A better approach is to create a loose coupling, with the QoS metadata outside the WSDL. In this thesis the latter approach is chosen, see the design in Chapter 3.

3. Design

A complete solution for QoS will require many different components. In this thesis one sub-problem is chosen, namely admission control using bandwidth as a metric, because bandwidth is the most limiting factor in disadvantaged grids. As previously stated, no existing solution solves the military-specific QoS requirements for Web Services. Therefore, a viable solution must take military limitations into account, and be able to take advantage of feedback provided by an ITR.

Admission control mechanisms can be implemented in many ways, but this chapter will discuss a broker-based solution. A broker-based approach was chosen, because it is a proven concept for QoS management. For example, IntServ with RSVP.

This proof of concept broker will provide the broker mechanism as an API, which all clients must invoke before invoking the desired service. This extra step requires modifications of the clients, as this is a proprietary extension. Clients which request services without using the broker first will be placed in the best effort class by the ITR. In a production system the broker itself should be a Web Service, to let non-local users use the broker.

It is important to emphasize that only the broker-API is proprietary, the Web Services themselves are not changed in any way and invocation is still done using SOAP over HTTP over TCP. Therefore, COTS clients will be able to work side-by-side with proprietary clients, but will not get the benefit of being prioritized by the ITR.

The thesis will use an object oriented approach to describe the design, but that does not imply that the broker must be implemented using either object orientation or an object oriented language. Instead, it is used as a convenient method to describe relations between components in the broker.

3.1. Policing

In order to provide QoS in a production network, policing must be performed to ensure that the clients and servers adhere to the broker's decisions, otherwise the decisions are worthless.

One common way of implementing a policing mechanism is by dividing the functionality into two components, a Decision Point (DP) and an Enforcement Point (EP). The DP in this design is the broker, because it determines which users that can invoke services. The task o a DP is to make decisions, while the EP is responsible for enforcing the decisions make by the DP. When a client is accepted the DP must inform the EP that a certain ID shall be mapped to a certain QoS class and for how long the ID is valid. The ID's validity can be revoked by the DP if needed, e.g., if forced to remove

low priority clients to make room for a high priority client.

The EP may run on the tactical router, the ITR, in order to be able to enforce the DP's decisions, if not run on the ITR it must be on the path between the client and the ITR. If the DP also runs on the ITR, then the EP can be a built-in part of the DP and not a stand alone service. Enforcing is done by looking into incoming SOAP messages and matching the ID found with the information received from the DP. With a valid ID, then a traditional network prioritization mechanism, for example DiffServ, is used to prioritize the packet. If the ID is expired, then the client's packets are rejected. Finally if the SOAP message has no ID, then it is assumed that is has been invoked by a COTS client and the message will therefore be placed in a best effort traffic class.

An EP is assumed to exist, because it can easily be implemented using existing solutions. The rest of the design will cover the DP, henceforth only referred to as the broker, as it covers non-existing functionality, which requires further research.

3.2. Assumptions

An admission control mechanism is not a stand-alone solution, but requires integration with other components. This section lists a few of those, and will assume they exist.

3.2.1. Bandwidth

The bandwidth used for calculations in the broker is assumed to be dedicated to the broker, either through a DiffServ class or that the link does time-division multiplexing. Any other traffic such as routing information and COTS Web Service traffic is assumed to use other DiffServ classes or occupy other slots if using time-division multiplexing.

3.2.2. Preemption and priority

In the NNEC FS, being able to preempt and prioritize clients is stressed. Preemption means that a task can be interrupted in order to let a another task run, and this is done transparently and without any consent from the running task. The preemption can be achieved by either pausing or stopping the running task. A lower priority task which is paused indefinitely while higher priority tasks are running will experience *starvation* [43]. Starvation is usually avoided by periodically preempting the higher priority tasks and allowing the low priority one run for a short time before pausing it again. In military networks ensuring delivery of high-priority information is considered more critical than avoiding starvation of low-priority traffic, which means that starvation in same cases can be accepted.

The NNEC FS requires that a new client must be able to preempt lower priority clients. However, network applications can not be preempted for an extended amount of time, because the opposite side of the connection requires feedback and has no idea that an application has been preempted. Consider two clients, a low priority client A, and a high priority client B. If A is downloading a file, and then B also wants

to download a file, B can preempt A in order to get all the bandwidth. Unless B's download is finished very fast, A's connection has probably timed out and A must then restart the download. Due to the low data rate in military networks, this time-out problem will occur even for smaller data requests, and network are therefore stopped rather than paused while preempted.

It is assumed that the capacity the best effort DiffServ class provides to COTS clients will be reduced to a minimum, or even to zero, when the bandwidth allocated to the broker is maximized. Otherwise best effort clients could continue using the link, while higher priority clients could be rejected due to a high priority client already running. This concept is called *priority inversion* and should be avoided. Priority inversion and starvation are further discussed in [43].

3.2.3. Roles and their priority

The broker implements role based admission control. Mapping the combination of roles and services to priorities are policy decisions which are beyond the scope of this thesis, though it is assumed that such a mapping exists and is available to the broker.

A client can not request a priority directly, only indirectly by stating its role and the service it requests. The combination of a role and service will yield one priority, and therefore it results in a [roles *x* services] matrix. This matrix can either be hard coded in the broker or provided by other means, but must be present when starting up.

3.2.4. Service registry

It is assumed that all services which the client will invoke are known to the broker. How the registry is built up, either through a discovery protocol or hard coded in both the client and broker does not matter as long they have the same WSDLs.

3.2.5. QoS description for service

Though a new version of WSDL exists, 2.0 [9], version 1.1 [10] is the most commonly used and will be used in this thesis.

To be able to provide QoS for a service, it is necessary to know how much resources it consumes. Also, some services may need to be prioritized independent of who is requesting the service, e.g., Web Services for setting off alarms.

In a normal Web Service request and response, the request is a small message, and almost the same size for all services, while the response is usually much larger than the request and varies a lot. The resource which is most important is the reply size, that is, the total length of the message replied to the client, including the XML. To let the broker easily make a decision, it needs to know the reply size for every service the client requests. This can be solved by creating unique services for each possible reply, though this might vastly increase the number of available services. A mapping between services and their reply sizes must be present in the broker, either hard coded into the broker, read in from a file at start up, or obtained through a service discovery

process. Experimental service discovery protocols that support QoS, e.g., SAM [37], may be used.

The WSDL itself cannot contain the reply size, as such a field is not standardized and would therefore break COTS clients. This has however been done in other solutions, see [65, 72], but they require proprietary clients. See Section 2.6 for more details about why the WSDL should not be modified, and why this design choose another approach.

3.3. Tactical router

Figure 3.1 shows that the packets which are routed between the two ITRs can use two paths. As the two paths indicate two different radio links, often with vastly different bandwidth, it is important to know which is used presently. The router will only use one link at the time, and it will choose the fastest one available. By looking at the routing table it can be established which link is currently used.

The ITR will run a HTTP service which will return the link ID which is currently active. A mapping between link ID and network capacity can either be hard coded in the broker service, or described in a file which the broker will read at startup. The Universal Resource Identifier (URI) for the service is <http://routername/routeInfo>.

3.4. Client API

Network QoS is not a part of the Web Services standards. Other QoS aspects such as reliability, transactions, and security are standardized and can be supported by COTS clients [34], but neither of those QoS properties are focused in this design, as they will be transparent to the broker.

Figure 3.1 shows the steps when a client successfully invokes the broker service, which are:

1. Client requests QoS from the broker service.
2. The broker informs the EP about a new client, and its QoS level.
3. Broker accepts the client.
4. The clients invokes the Web Service.
5. The Web Service response is sent to the client.

Also as seen in Figure 3.1, the broker is not between the client and the ITR, thus COTS clients do not need to know about the broker. They will instead skip the first three steps above, and immediately invoke the Web Service. However, in this case they will receive no guarantees, and only best effort is provided.

The rest of this section covers the brokers only public function, namely *handleRequest*.

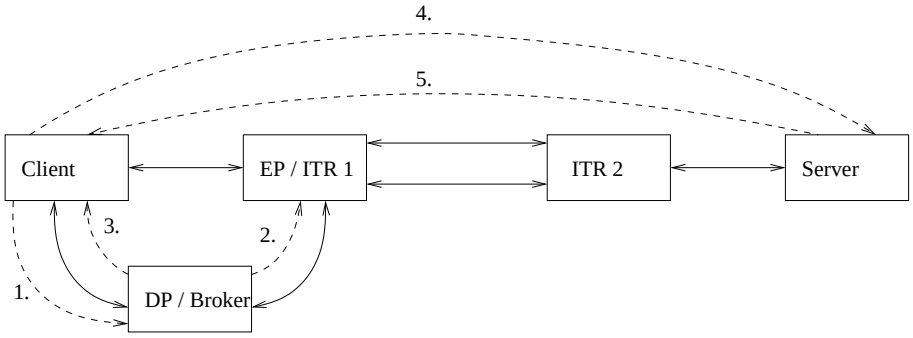

Figure 3.1.: Overview of client, broker and server

3.4.1. Invocation

handleRequest is the function a client must call in order to possibly gain priority. The function can either be exposed using a Web Service, thus allowing remote users to access the broker, or directly through the API. Most likely the broker will be used as a Web Service, because then it can serve any nearby client which understands how to use the broker.

3.4.2. Parameters

The three parameters listed below are all mandatory. Lacking either of them will cause the request to be rejected.

Time limit

The deadline before the request must be finished. As the request itself might be delayed due to slow links, this parameter should be an absolute date, and not a relative one, e.g., at latest at 1290519944 seconds after Epoch[1], instead of «within 30 seconds». This assumes that time synchronization is provided. The parameter's type is a 64-bit integer, instead of using the classic time type of a signed 32-bit integer, which will wrap early in 2038. This is known as the «year 2038 problem».

Role

This parameter is a one word string which describes the users role, and can consist of up to 32 characters. Valid characters are specified in section 2.2 in the XML specification [8]. Example: "gunner". The definition of roles in a QoS setting is discussed in Section 1.4.1.

[1] 00:00:00 UTC, January 1, 1970

Service

Identifies the Web Service the requester wants by using a unique string. This string has the same limitations as «Role»'s string. The content of the string is up to the implementation. An example is the service endpoint, which is an unique URI containing «http://server.name/serviceName». Another example is a hash of the Web Service's WSDL, which globally uniquely identifies a service *instance*. Conversely, if the endpoint is removed from the WSDL, then the hash will uniquely identify a service *type* [37]. Hashing is on the other hand a more CPU expensive operation than just using the endpoint as an identifier, though it could be beneficial if used in conjunction with service discovery.

The Web Service must have a known reply size. If none is found the client is rejected.

3.4.3. Response to client

The broker will always return a key based list, e.g., a dict in Python or a HashMap in Java. It contains at least one element. If it has a key with name «rejected», the client is rejected and must behave accordingly. Unless rejection the client is accepted.

Reject

If a client gets a reject, it should display the error, as given by looking up the value for «rejected» in the list. The end user can then later manually retry to invoke the service, or such a retry could be automated, depending on the system.

Accept

If accepted the key based list contains two elements, whose keys are «id» and «delay». On acceptance the client must first wait the delay specified, and then the client must add the id to the outgoing SOAP request when invoking the service. The id must be sent to the EP by the DP, together with the QoS class and period when the id is valid.

The client could have been allowed to label its packets with the correct QoS class, but clients are not considered trustworthy enough, and since an EP is required anyway to support preemption, the EP can label the packets with the corresponding DiffServ class.

3.5. Broker internals

This section will first briefly describe the data flow for a request when it arrives, and then describe all the elements needed to build a broker.

3.5.1. *handleRequest*

An invocation can be seen in Figure 3.2. The parallelograms are input or output, diamond shaped boxes are conditionals and rectangles are general processing steps.

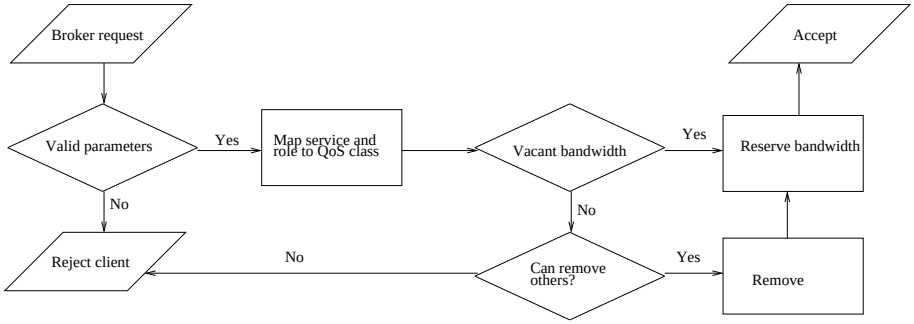

Figure 3.2.: High level view of an invocation of the broker service.

First the request arrives at the broker, and then the broker checks if the parameters are valid according to the description in Section 3.4.2.

Map role to QoS class

Assuming the client has valid parameters, the combination of a role and service will yield a priority from the Constant class's *getPriority* function.

Vacant bandwidth

The bandwidth broker's main task is to account for how much bandwidth is used presently, by whom and for how long. Similar if queuing is supported, then future reservations must be accounted for as well.

Estimating the amount of bytes to reserve is more complex than dividing the expected reply size by the network speed. COTS clients use TCP over HTTP, and TCP will reduce its sending rate if packet loss is detected, which is highly likely in a disadvantaged grid, thus making accurate calculations difficult.

After a reply size is calculated, the vacant bandwidth can be found by searching the broker's ring buffer, more info in Section 3.5.3, which contains all the accounting. If there is a large enough interval in the ring buffer before the clients time limit expire to hold the request, then the bandwidth must be reserved and then the broker accepts the request.

Can remove others?

In a military context policy is that priority trumps, thus any request of a lower priority can be put on hold or discarded if necessary, in order to fulfill higher priority requests. How aggressive this demand for preemption shall be met is a matter of policy. For instance, should a client be stopped to let a higher priority client run, even though both time limits could be met? The broker should support different kinds of policies,

configured by a config file, in order to support deployment in networks with varying policies.

If no lower priority clients can be removed, then the client must be rejected.

Remove client

The broker has no contact with client the after it has requested priority, and can not communicate with it. Therefore, the broker can not ask the client to stop its ongoing invocation, or if a future reservation, ask it not to invoke in the future. The broker must withdraw its acceptance, by revoking the client's ID as a valid token. Then the client will be blocked by the EP, and if it wants to retry to get the Web Service it must issue a new request to the broker and re-invoke the service.

Reserve bandwidth

If the request reaches as far as here, then it shall be accepted. Bandwidth reservation is done by adding the client to the ring buffer.

3.5.2. Client class

Each client must be represented in a Client object. This object must contain the client's time limit, priority, role and the service it requested. For statistical purposes, a status variable is used. This is an integer, whose values are not specified, but a mapping between the values used and a textual representation must be present. For example, 0 can be mapped to "Success".

3.5.3. Ring buffer class

A ring buffer, or circular buffer, is a data structure similar to a linked list, except that the last element points to the first, thus forming a ring. Caution must be made to not blindly loop through the buffer when searching, but check for a sentinel value, typically the ID of the first element where the iteration started.

To store bandwidth reservations, a ring buffer was chosen, because it has some convenient aspects, such as: fixed amounts of space, and therefore implicit set up an upper threshold for reservations. Fast to search in for finding intervals which can be used for requests. Combined with a maintenance thread which rotates the ring buffer, old entries will automatically expire, given that the maintenance process clears the slot after rotating.

The ring buffer will have three public variables, *currentSlot* and *lock* and *kbpts*. *currentSlot* points to the «start» of the ring buffer, i.e., the buffer slot whose time is closest to the present time. To ensure the integrity of the ring buffer, the *lock* variable must be marked as locked before modifying the time slots. *kbpts* is the number of KB per time slot, and must be updated every time the network speed changes.

maintenance

The maintenance process must run as often as the length of each slot, thus pruning invalid and expired clients and thereby maintaining the integrity of the ring buffer. It must also set *currentSlot* to the next slot.

Time slots

Each element in the ring buffer consist of a TimeSlot object. These objects are the ones that actually stores the data in the ring buffer. A TimeSlot object contains five variables:

- *next*, pointing to the next time slot.
- *time*, the time the slot will expire. Note this is not used by the maintenance process, but when iterating the ring buffer to find a suitable interval for a request.
- *id*, an unique id
- *usedBytes*, how many bytes which have been reserved in this time-slot
- *clients*, a key based list, where a client's object is the key and it's reservation in bytes in the current slot as the value. A client may span multiple time slots, if it requires more bandwidth than one time slot can provide.

The class has two functions: *reset* which resets *usedBytes* to zero and clears *clients*. *clearReservations* is called by the ring buffer maintenance process, to reset the slot, but also if policy dictates, the broker can revoke any client's ID if the slot was last one containing the client.

canAccept

This function will attempt to fit a client into the ring buffer, and is able to remove other lower priority clients if needed and supported. The behavior of this function is heavily dependent of policies, such as whether queuing or preemption is allowed, and it must be able to handle different policy settings.

The function's parameters are a client object, a size, and optionally a start time slot. It returns a key based list, which contains 'rejected' on failure, with a reject reason; 'startSlot', where it can be added by the *add* function; and 'delay', if delayed starting is needed.

If the function was called with the optional start slot, then that is used as the start slot, otherwise *currentSlot* is used as the start slot. The start slot defines where the function will start searching from in the ring buffer, while the search always stops when reaching *currentSlot*.

First it will assert that the request's size is less or equal to the entire ring buffer. Then the easiest case is when no clients are in the buffer, then just add the client and return. The seconds easiest is when queuing is disabled. Pseudo code shown below, where *QUEUING* indicates the queuing policy, *PREEMPT* is the preemption policy, and *client* is the parameter to *canAccept*:

```
clients = getAllClients()
if QUEUING == false:
    c = clients[0]
    if c.priority < client.priority:
        if PREEMPT == true:
            preempt(c)
        else:
            return 'rejected': \
                   'No queuing or premption, will finish lower priority'
    else:
        return 'rejected': 'No queuing, too low priority'

    return 'startslot': startslot
```

Two simple cases exists when queuing is enabled: first that the start slot has room, and secondly that the time limit can be kept after adding after the presently last client in the ring buffer.

If neither of the simple queuing cases are possible, then preemption must be enabled if a client shall be accepted. To fit a client in a full ring buffer, the function must do a 2-pass through the ring buffer. First round is to check if there is room in the ring buffer, including the possibility of terminating low priority clients. It is important to not terminate any clients before we *know* there is room for the entire request, e.g. the client is of medium priority, and the first one in the ring buffer is a low priority client, whose size is less than the new medium priority client, while the rest of the buffer is full and only consists of high priority clients. Thus terminating the low priority client would be premature and would just reduce the overall client satisfaction. The second pass will terminate lowest possible/permitted priority, if needed.

add

This function adds a client to the ring buffer. Its parameters are the client, a size and an optional start slot, which is handled as in *canAccept*. It returns nothing.

For new requests this function must not be called unless *canAccept* has successfully been called, as it will unconditionally attempt to insert the client into the ring buffer, starting from start slot. An error is raised if the function reaches *currentSlot*, or it reaches a time slot whose *usedBytes* equals *kbpts*, which indicates that the slot's capacity has already been reached.

networkUpgrade

If the network capacity is increased, all reservations in the ring buffer will be finished much faster than previously calculated. Thus the reservations should be recalculated in order to make room in the ring buffer for more clients. No communication with the clients is possible, therefore the start time for each reservation must be unchanged,

while each reservation will span less time slots. This may cause a heavily fragmented ring buffer, but *canAccept* can handle that by inserting potential now requests into the empty slots.

networkDowngrade

When the network capacity is decreased, not only will the link be slower, but no traffic has been sent between the clients and the server in the interval between the primary link is down and before the backup link is up and running. The delay is dependent on the routing protocol used, see Section 4.2.1 for a discussion on how to minimize the delay when using OSPF. Moreover, TCP will perform exponential backoff when no ACKs are received, thus it will not immediately resume the transfer when the backup link is up and running. The estimated bytes dropped must be added to the rest size of each reservation, but only if the reservation had started, because future requests have not experienced dropped packets.

As the capacity is reduced, the reservations must be recalculated and some of them must likely be removed. This is easily done by creating a list of all clients with their start slot and bytes left, where the clients are listed in the order the were added to the ring buffer. Then for each client, oldest first, call *canAccept* and if a positive answer, then *add* it to the ring buffer. Clients which are rejected by *canAccept* can not be informed that they are rejected, as Web Services has no feedback mechanism, but the requests will be stopped by the EP.

3.5.4. QoSbroker

This is the main class, though it has little functionality. It holds the ring buffer object, starts the network info update process, and it provides the *handleRequest* function, which mainly is a wrapper around the ring buffer's *canAccept* and *add* functions. If the optional Token Bucket mechanism below is implemented, then it will reside in this class.

network info updater

A process which will poll the ITR periodically to check for changes in the network topology. If the network capacity is increased, call the ring buffer's *networkUpgrade*, while if decreased, call *networkDowngrade*. The polling should be done often, at least every second, to ensure that the broker swiftly adapts to network topology changes.

Token Bucket for small requests

A Token Bucket mechanism is often used when providing a limited resource, with burst support. Only requests with small reply sizes are allowed to use the mechanism, as a way to compensate for possibly uncertain bandwidth estimations and mitigate the problem were tiny high priority requests will preempt any lower priority requests. If

accepted, the clients are not added to the ring buffer, and are permitted by the DP to start immediately.

4. Implementation and Evaluation

In this chapter the implementation of the design from the previous chapter is presented, together with a discussion on implementation-specific limitations and how they were dealt with. Then follows the setup of the test bed and description of the tested Web Services. Finally, the results from the experimental evaluation are discussed.

4.1. Implementation

Python [60] was chosen as the programming language to implement the broker, because it is a powerful scripting language which allows rapid development of proof of concept implementations.

The broker was implemented as a DP with some functionality of a built-in EP, i.e., preemption. Creating a complete EP is beyond the scope of this thesis. A broker object will exist to let the clients directly invoke *handleRequest* and then start according to the answer. Each client would be a python thread, using Python's threading library [59]. Revoking of client's access would be done by stopping the client's thread.

4.1.1. Threading limitation

The majority of the code for the broker was already written when a major limitation with Python's threading library [59] was discovered: a thread cannot be preempted or stopped, but can be controlled by setting a sentinel value. An example in pseudo code is given below, which shows the only way to stop a Python client using the threading module. One must call the client's *stop* function, and then wait until *doSomething* has finished.

```
class Client:
    running = True
    func forever:
        while running == True:
            doSomething()

    func stop:
        running = False
```

Lacking preemption violates the NNEC requirements, as discussed in Section 3.2.2. Python's lack of preemption support is not something new, as already in 2000 this was requested as a feature [30]. There, Guido van Rossum, the original author of Python, replied

> This is a very controversial feature. If a thread owns a resource (e.g. it holds a lock), what happens to the resource when the thread is killed?
>
> The best idea I can come up with is to make it possible to send asynchronous exceptions to threads – but this doesn't help for threads that are blocked in an I/O operation. That might be solved through sending signals, but that's a Unix-only solution, and probably isn't even consistent across different Unix thread implementations.

The discussion continues, though the signaling approach is abandoned as a cross-platform solution is required. In the end, van Rossum added the feature request to Python Enhancement Proposals number 42 [31], where it still remains after more than 10 years have passed.

None of the clients invoking the broker would hold any locks, so the resource argument is moot for this implementation, though it is certainly something to consider in other projects. Many other programming languages support preempting threads. For example, C via *pthread_cancel* [68] and Java [53] via its *Thread* library's [55] *Stop* function, though the latter is is deprecated [54]. In Java, all locks would be unconditionally released when calling the Stop function. C's pthreads also supports clean up routines.

4.1.2. Partial solution

Following van Rossum's suggestion of an «asynchronous exception to threads», Tomer Filiba implemented such an extension to the threading library [19]. Though still as van Rossum wrote, this will not work for threads which are blocked in I/O operations. Unfortunately, the Web Service requests cause the threads to block while they wait for a reply, and even more unfortunate this can take a while because the network links used are very slow.

In contrast to the pseudo code in the previous subsection, the extension provides a possibility to stop any code which occurs outside the local scope in a thread. That is, code inside the *doSomething* function can be aborted, and thereby there is no need to wait for it to finish and then check the sentinel value.

Stopping threads by asynchronous exceptions was deemed sufficient to provide an experimental broker, and thus it was not rewritten in for instance C or Java.

Except for the issues with Python threads, the rest of the implementation was done according to the design.

4.2. Test setup

Two laptops were used for respectively client and broker, and the Web Services server. The client and broker laptop ran FreeBSD 8.1 using Python 2.6.6. It had 4 GB RAM and an Intel Core2 Duo T9600 CPU. This laptop is hereafter only referred to as the broker laptop.

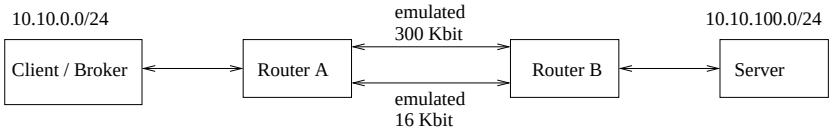

Figure 4.1.: The testbed, showing the client and broker machine, connected to the server via the ITRs.

The server ran Ubuntu 10.04, with the Web Services deployed in GlassFish via NetBeans 8.6 using Oracle's Java 1.6.0. It also had 4 GB RAM, but an Intel Core2 Duo T9550 CPU.

Figure 4.1 shows the four machines used in the testbed. Each laptop was directly connected with a network cable to a router. The network speed was 100 Mbit/s. The client IP address was 10.10.0.3 and router A's IP was 10.10.0.1. Similarly, the servers IP was 10.10.100.3, and router B's IP was 10.10.100.1.

It is assumed that the server providing the service has sufficient CPU and network resources to reply to a request in a timely manner, and therefore, the only bottleneck between the client and server is the link between the two tactical routers.

No network tuning were performed to either of the installations, they both ran with all default options, and their TCP congestion control algorithm was the default New Reno [20]. Both machines support SACK [48], which can improve throughput on links with high delay.

The ring buffer, as discussed in Section 3.5.3, was setup with 0.2 second time slots, and spanning 40 seconds, thus a total of 200 time slots.

4.2.1. Tactical router

Two machines were installed with Thales' tactical router software, a modified Vyatta distribution, to provide a network topology with redundant paths, though with varying bandwidth. OSPF-MT was used as routing protocol, with the highest capacity link as the primary topology, and the slower one as secondary topology.

A HTTP daemon, from Script B.2.2, ran on router A to provide the Broker with topology information.

Physical link emulation

Using real radios was not an option, because the project was unable to procure radios for testing, but emulating slow radio links was considered sufficient, because bandwidth is easy to emulate correctly. Furthermore, using real radios might show buffer sizes or other properties which could be considered a reason to classify the thesis. Bandwidths for two commonly used radios from Table 2.1 were chosen, namely a 300 Kbit/s HB-UHF and a 16 Kbit/s UIDM.

The Vyatta routers were interconnected using 100 Mbit/s full-duplex switches, so in order to behave like military radios, first and foremost the bandwidth must be severely

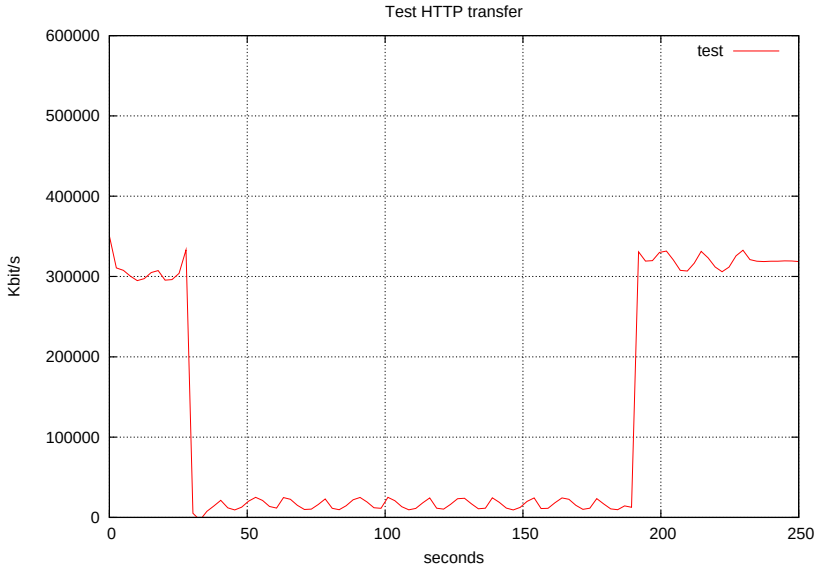

Figure 4.2.: A HTTP transfer over the ITRs, where the primary link is disabled and then re-enabled.

reduced. See Figure 4.1 for the setup. Vyatta comes with a suitable command, *tc*, short for «traffic control», a command from the iproute2 [42] collection. The command tc enables varying types of queuing and bandwidth limitations. Script B.2.1 was used to limit the traffic for each link. As seen in the script, only packets whose destination is the client network, 10.10.0.0/24 and source is the Web Service server network, 10.10.100.0/24, and vice versa, have got bandwidth limitations. This is because otherwise too many OSPF packets would be delayed and then the routers would assume that the other router was dead. Not shaping traffic which originated at either Vyatta router and was sent to the other solved the problem, and can be viewed as if DiffServ was giving highest priority to OSPF packets.

The HTTP daemon was not only used for getting topology information, but also emulating loss of radio connectivity. When radio connectivity is lost, packets are just lost in the void and no feedback occurs. Connectivity was controlled with *iptables* [63], which blocked network traffic on the ITR both in, out and through an interface, that is packets originating, to, from, and forwarded by the router. Support for iptables was built into the HTTP daemon, providing new commands. The most commonly used were: flushIPTables, disablePrimary and enablePrimary.

Figure 4.2 shows an HTTP transfer from the server to the client, where the primary link is disabled and later re-enabled. Notice that the transfer is stalled for a few seconds when the primary link is disabled, before resuming transfer on the slower link. This is in contrast to when the link is re-enabled, and the transfer speed increases without having a pause first.

Hello – Dead	Direction	average	max	min	std dev
10 – 40	Downgrade	33.13	35.96	31.15	1.60
	Upgrade	13.11	15.00	9.41	1.90
1 – 3	Downgrade	2.83	3.22	2.48	0.20
	Upgrade	4.55	6.38	2.33	1.29

Table 4.1.: OSPF upgrade and downgrade delay for 10-40 vs 1-3 Hello and Dead intervals. $N = 10$.

OSPF

It is important for the broker to be informed of network topology changes as soon as possible, because its decisions must be made on the present network topology. Especially network downgrades are important, as bandwidth is then suddenly much more scarce, as seen in Figure 4.2.

Each router in an OSPF based network must periodically, default every 10 second, unicast to their directly connected neighbor routers that they are alive. This is done with a «Hello message». The same message is also sent to two multicast addresses, 223.0.0.5 and 223.0.0.6, with a Time To Live (TTL) value of 1, to ensure that only routers on the same subnets receive the message. If a neighboring router has not received a Hello message in «Dead Interval», default 4 times the Hello interval, 40 seconds, it will recalculate the routing table without the failing adjacent router. It is important to set both the Hello and Dead interval to the same value on all routers in the network, otherwise the routing will be unstable.

Furthermore this leads to the expected time for the router to detect a link failure to be 3 x Hello interval + 0.5 Hello interval = 35 seconds. Minimum time is slightly more than 30 seconds and maximum is 40 seconds.

35 seconds is very long, and can easily be improved as discussed in [61] and [11], by reducing both the Delay and Hello intervals. While many modern routers supports a sub-second Hello interval, Vyatta supports neither sub-second Hello interval nor does OSPF support fractional intervals, thus in order to receive at least one Hello message in each Dead interval, the Dead interval must be 2 seconds or more.

In the setup, Vyatta was configured with a 1 second Hello interval and a 3 seconds Dead interval, and the difference in detecting network changes can be seen in Table 4.1. 1 second Hello interval was chosen because that was the minimum allowed, and 3 second Dead interval because that would allow one Hello to be lost without prematurely re-routing and thus let the broker needlessly perform a downgrade.

Table 4.2 shows, as expected, that the average delay for the broker to detect a change in the routing table is lower when polling the ITR every 0.05 second versus every 0.5 second. The downgrade delay is consistently lower for the most frequent polling, but the upgrade delay has a higher standard deviation. Measurements done with Script B.3. In the evaluation the ITR was polled every 0.05 second.

When Vyatta detects a physical link change, e.g., someone pulls out a network cable,

Polling interval	Direction	average	max	min	std dev
0.05	Downgrade	2.71	3.12	2.36	0.23
	Upgrade	4.17	7.15	2.47	1.82
0.5	Downgrade	2.83	3.22	2.48	0.20
	Upgrade	4.55	6.38	2.33	1.29

Table 4.2.: OSPF upgrade and downgrade delay when polling the router at different frequency, for 1-3 Hello and Dead intervals. $N = 10$.

Method	average	max	min	std dev
iptables	2.68	3.00	2.40	0.19
Indirectly removed	2.82	3.85	2.30	0.42
Directly removed, local	2.45	2.95	1.10	0.52
Directly removed, remote	1.96	3.10	0.65	0.73

Table 4.3.: Time between ICMP packets on network degrade. OSPF with 1-3 Hello and Dead intervals. $N = 10$.

it will issue an interrupt to OSPF which will bypass the Dead interval timer and immediately recalculate the routing table, as recommended in [61]. Though, as seen in Table 4.3, «immediately» still takes some time.

Table 4.3 lists the four different ways to force a downgrade on the network, and the time in seconds between last ICMP reply before the downgrade and the first one after. Each test was performed 10 times. «iptables» means all traffic to, from and through the primary interface was blocked with iptables, and to emulate loss of a radio link. Same effect was emulated with «Indirectly removed», where two switches were placed between the routers, and the cable connecting the two switches was disconnected. «Directly removed» means a network cable directly connected to a router was removed. where «local» is when the cabled was removed from router A and «remote» when removed from router B.

The measurements were done with a combination of two scripts, B.3 and B.4. The first one pinged the Web Service server continuously, while the primary link was disabled and enabled 10 times. iptables used the router's HTTP daemon, while the three other ones were done by manually removing the network cable. The second script was used to parse the log from the pinging.

Indirect removal of a cable and iptables have almost the same expected downgrade delay, though iptables have less variations, as seen in Table 4.3. In the rest of this thesis iptables was used to drop packets, not reject, in order to emulate loss of a radio link.

Bandwidth		average	max	min	std dev
16 Kbit	Received	828.80	834.00	782.00	15.60
	Sent	912.00	912.00	912.00	0.00
	Time (s)	0.39	0.50	0.01	0.19
300 Kbit	Received	834.00	834.00	834.00	0.00
	Sent	912.00	912.00	912.00	0.00
	Time (s)	0.01	0.01	0.01	0.00

Table 4.4.: GPS Web Service sent and received bytes, and time in second spent. $N = 10$.

Router configurations

The configuration for Vyatta running on router A and router B can be seen in Appendix A.1 and Appendix A.2, respectively.

4.2.2. Web Services

For test purposes two Web Services have been created, GPS and Camera. The Camera Web Service can present the results from one web camera in three different resolutions. Both services were implemented in Java and deployed in Glassfish.

GPS

The GPS Web Service provides a timestamp together with longitude and latitude, using only 38 bytes. Though as seen in Table 4.4, much more data is sent and received, when wrapped in a Web Service. The directions are from the client's point of view.

Web camera

The Camera Web Service has three operations, *captureHigh*, *captureMedium*, and *captureHigh*, which provides a resolution of 800x600, 640x480 and 320x200, respectively. The broker matches Web Service name with reply size. Thus, three Web Services, *CameraHigh*, *CameraMed*, and *CameraLow* were created, and they call the respective Camera operation. In the rest of the thesis the three wrapped Web Services are the ones referred to when referring to Camera Web Services.

Each of the Camera Web Services read a jpg file, convert it to base64 [39] and serve the result. The pictures were captured using Script B.6. Running deterministic measurements became difficult because the picture sizes varied a lot and therefore affected the broker's decisions on whether accepting or rejecting. To provide consistent results between each measurement the capturer was stopped, and the same reference files were used throughout the measurements.

The reference files measured 70955, 46480 and 16123 bytes as jpg files. The XML response to clients must not contain binary data, therefore the files must be converted

Bandwidth	Resolution		average	max	min	std dev
16 Kbit	Low	Received	23122	24467	22967	448.53
		Sent	1649	1732	1640	27.60
		Time (s)	11.60	11.68	11.43	0.08
	Medium	Received	64905	64905	64905	0.00
		Sent	3102	3102	3102	0.00
		Time (s)	32.81	32.90	32.71	0.06
	High	Received	98879	100229	98279	450.00
		Sent	4299	4346	4294	15.60
		Time (s)	50.04	50.57	49.72	0.21
300 Kbit	Low	Received	22915	22967	22915	15.60
		Sent	1380	1380	1380	0.00
		Time (s)	0.59	0.60	0.54	0.02
	Medium	Received	64873	64905	64853	25.47
		Sent	2322	2322	2322	0.00
		Time (s)	1.70	1.74	1.68	0.02
	High	Received	98682	98729	98677	15.60
		Sent	3150	3150	3150	0.00
		Time (s)	2.62	2.66	2.61	0.01

Table 4.5.: Camera Web Service sent and received bytes, and time in seconds spent. $N = 10$.

to a base64 representation, which yield 94806, 61974 and 21500 bytes, respectively. This is a consistent 33% increase, which is expected for base64.

Table 4.5 shows the three web camera services used on each link. While using the 300 Kbit link all requests are performed quite rapidly, but on the much slower 16 Kbit link especially the highest resolution is immensely slow. Also notice that the number of sent bytes increases as the link speed is reduced. This happens because the data is received more slowly and thereby TCP's sliding window is increased more slowly, and thus requires more ACKs to be sent.

4.2.3. Roles

To make it more obvious why a certain role got higher priority, the role names were set to possibly real role names, instead of «role1», «role2», etc. The following four roles were defined:

- «gunner» – A user out in the field, and they are often prioritized.
- «recon» – Needs to know movements and generally be informed.
- «planner» – Is not in a hurry and usually requires maximum resolution.
- «chef» – The chef is curious and wants to see what is happening.

	gunner	*recon*	*planner*	*chef*
GPS	High	High	Low	Best effort
CameraHigh	High	Medium	Low	Best effort
CameraMed	High	Medium	Low	Best effort
CameraLow	High	Medium	Low	Best effort

Table 4.6.: Priority matrix used in the evaluation.

4.2.4. Priority matrix

Four levels of priorities were defined: High, Medium, Low and best effort. Table 4.6 shows the priority matrix used in the evaluation.

4.2.5. Time limits

The «gunner» role had 15 second time limits for GPS and Camera Web Services, while the other three roles had 30 seconds. «gunner» had the shortest time limit, because it was assumed he was in the field, and because of that he requires information within a short time frame for targeting purposes.

4.2.6. Web Service requests

Two test sets were created for Web Services. One short called «niceset» which would not require any preemption or queuing on 300 Kbit. The other one was called «hammering», and as the name dictates it was used to stress test the broker by hammering it with requests. In the evaluation, «hammering» is the test set referred to, unless explicitly stated otherwise.

The distribution of priorities is important, because if too many requests have the highest priority, the effect of having a high priority will be much less effective. On the extreme side, if everyone has the highest priority it would be equivalent with no one having it. In the hammering test-set the distribution was 5% High, 15% Medium, 30% Low and 50% best effort, spread over 100 requests. The High and Medium priorities are fairly rare, as they are likely to be in an actual deployment. Between most of the requests in the test-set there will be a short delay to let some of the requests finish. The delay is between 0.1 and 5 seconds. In Section B.1 «niceset» is shown, to provide an example of how to create test requests. The first field defines the role, second is the delay, and the third is a tuple containing the Web Services. Each Web Service is used as an argument to the broker, until one has been accepted or all have failed.

4.2.7. Emulated networks

One of the most interesting aspects about the broker is that it can get feedback from the ITRs on which link is currently active, and thereby how fast the current connection

is. Five different network scenarios were defined, in order to test the broker's ability to adapt to network changes. The following networks were tested:

- «PrimaryOnly» – Always use the primary 300 Kbit link.
- «BackupOnly» – Always use the backup 16 Kbit link.
- «PrimaryThenBackup» – Start with the primary link active, after 30 seconds disable it and wait for the backup link to be active.
- «BackupThenPrimary» – Start with the primary link disabled, and then after 30 seconds enable it and thus let provide more bandwidth.
- «VaryingNetwork» – Will simulate a mobile unit going up and down on hills, where the primary link is only available near and at the top. The test will start will the primary link enabled. Then it enters a loop where it disables the primary link after 20 seconds, and then after 20 seconds more re-enables it.

The emulated network is stopped when all requests are either finished or preempted, and that will depend on the runtime for each test set.

4.2.8. Queuing options

Policies for queuing can vary from network to network. The implementation had five options which could either be enabled and disabled. All permutations were tested, a total of $2^5 = 32$ combinations. In the evaluation these permutations are referred to both by their representation, e.g, True-True-False-True-False, and the permutation's number. The list of representations and their number can be seen in Table 4.7.

The following options were implemented:

1. «QUEUE_DELAYED_START» – Support a delayed start. If a client is running and a new client can be run after the running client is finished, and still keep its time limit, then reserve bandwidth after the running client and inform the client that it must delay its start. If the new client has higher priority and it cannot wait until the current running is finished, it will preempt it.
2. «QUEUE_ALWAYS_HIGHEST_PRI» – Always preempt lower priority clients. The negative aspect is that the network resources have already been spent, and that the lower priority client will probably try again later. Positive aspect, for the high priority clients at least, swifter responses.
3. «QUEUE_TOKEN_BUCKET» – Enables the mechanism as described in Section 3.5.4. The largest reply size supporting the Token Bucket was set to 1 KB, or slightly more than the GPS Web Service. Replenish would happen every 1 second, were the primary link gains 1 token at each tick, up to a maximum of 5 tokens. The backup link was limited to 1 token, and gaining 0.3 token each tick.

Number	Delayed start	Always highest	Token Bucket	Enforce timeslots	Slow start adjust
1	False	False	False	False	False
2	False	False	False	False	True
3	False	False	False	True	False
4	False	False	False	True	True
5	False	False	True	False	False
6	False	False	True	False	True
7	False	False	True	True	False
8	False	False	True	True	True
9	False	True	False	False	False
10	False	True	False	False	True
11	False	True	False	True	False
12	False	True	False	True	True
13	False	True	True	False	False
14	False	True	True	False	True
15	False	True	True	True	False
16	False	True	True	True	True
17	True	False	False	False	False
18	True	False	False	False	True
19	True	False	False	True	False
20	True	False	False	True	True
21	True	False	True	False	False
22	True	False	True	False	True
23	True	False	True	True	False
24	True	False	True	True	True
25	True	True	False	False	False
26	True	True	False	False	True
27	True	True	False	True	False
28	True	True	False	True	True
29	True	True	True	False	False
30	True	True	True	False	True
31	True	True	True	True	False
32	True	True	True	True	True

Table 4.7.: Queuing options permutations and their number

First both links used the primary link's values, but the result was way too intrusive on the backup link. Recall that a token corresponds to a request up to 1 KB, which is half of the backup link's capacity for one second.

4. «QUEUE_ENFORCE_TIMESLOTS» – If a client's ID should be revoked after the reservation has expired from the ring buffer. This might be long before the client's actual time limit has expired, and the option does not consider if any more requests are in the ring buffer. Thus potentially a client might be revoked and leave the link idle.

5. «SLOW_START_ADJUST» – Estimating transfer time is more complicated than dividing transfer size by speed per second. This is mainly due to TCP not utilizing the full capacity of the link at all times. To compensate for slow start, the requests was slightly increased by the formula `min(KBpts * 0.4, size*2)`, where *KBpts* is KB per time slot and *size* is the request's original size. The formula will mostly help when invoking tiny Web Services, such as the GPS, because otherwise the reservation would almost have passed before the client has invoked the Web Service.

Option 1 and 2 are the most intrusive ones. For example, leaving both off will disable queuing and always allow the current client to finish. This may lead to priority inversion, as a best effort client would cause any other clients, even higher priority ones, to be rejected.

4.3. Evaluation

The broker was evaluated over both test sets, *niceset* and *hammering*, five emulated networks and the 32 permutations of queuing options, giving a total of 320 tests. Additionally, reference testing without the broker was done for both test sets with the five emulated networks, thereby adding 20 more tests. This section will show how the broker performed in the various tests, compared to skipping the broker. The evaluation took 9.5 hours to run all 340 tests.

Service	Priority	#	Status						
			Success	Rejected	Too late	Fail downgrade	Exceeded slots	Terminated	Failed
GPS	High	3	100.00	0.00	0.00	0.00	0.00	0.00	0.00
	Medium	0	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Low	9	33.33	33.33	0.00	0.00	0.00	0.00	33.33
	BE	25	24.00	72.00	0.00	0.00	0.00	0.00	4.00
Camera-High	High	2	100.00	0.00	0.00	0.00	0.00	0.00	0.00
	Medium	13	38.46	38.46	15.38	0.00	0.00	0.00	7.69
	Low	20	10.00	75.00	0.00	0.00	0.00	0.00	15.00
	BE	22	4.55	77.27	0.00	0.00	0.00	0.00	18.18
Camera-Medium	High	0	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Medium	7	0.00	85.71	0.00	0.00	0.00	0.00	14.29
	Low	2	0.00	100.00	0.00	0.00	0.00	0.00	0.00
	BE	10	0.00	100.00	0.00	0.00	0.00	0.00	0.00
Camera-Low	High	0	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Medium	6	50.00	33.33	0.00	0.00	0.00	0.00	16.67
	Low	1	0.00	100.00	0.00	0.00	0.00	0.00	0.00
	BE	12	0.00	75.00	0.00	0.00	0.00	0.00	25.00

Table 4.8.: Request status. options: False-True-False-False-True, network: VaryingNetwork, testset: hammering

Each test creates a table which shows how the clients behaved. A random sample is shown in Table 4.8, and states that the test was using the «hammering» test-set, with option set number 10 on the «VaryingNetwork» network, and shows the clients' status for all services. While the test-set contains 100 requests, see Section 4.2.6, it can be seen in the table's third column, named #, that the number of requests is 132. These numbers are different because a rejection is counted every time a client gets a rejection from the broker. That is, if a client wants CameraHigh or CameraMed, and the broker rejects the request for CameraHigh but accepts CameraMed, then both a rejection and a possible success is counted. The clients' statuses are listed in percent of the total number of requests.

A client's status can be either of the following:

- success – The client finished successfully within the given time limit.
- rejected – The broker rejected the client. Will only get this if *all* the services it wants are rejected.
- too late – The client finished, but after the given time limit.
- fail downgrade – If a client was preempted due to not being allowed to continue when the network capacity was downgraded, see Section 3.5.3.
- exceeded timeslots – Preempted due to the client being too slow and not finished within the reservation, and the option «QUEUE_ENFORCE_TIMESLOTS» was enabled.
- terminated – The client was preempted too give room for a higher priority request in the ring buffer.
- failed – As discussed in Section 2.4.3, TCP will have problems on heavily utilized links. This status is set when network problems occurs.

4.3.1. Latency overhead

During the evaluation, the broker's *handleRequest* function was called 23472 times. Using the broker requires all clients to first send a request to the broker and wait for a response before invoking the actual service. This will lead to all clients experiencing a small time delay for each request. The average delay was 0.01 seconds, while the maximum was 0.50, and the minimum 0.00. Standard deviation was 0.02. This overhead seems negligible.

One must bear in mind that the broker was invoked directly through a Python object, and not as a Web Service. Table 4.4 shows that a small Web Service like the GPS can be served quite fast, even on the 16 Kbit link, and thus if the broker was published as a Web Service, the overhead would still be low. This was however not implemented, as that would require a more sophisticated EP than the client's thread manipulation used by this broker.

4.3.2. Saturated vs non-saturated network

In a non-saturated network, the broker should not reject traffic which a QoS-less network will handle. Similarly, high-priority requests should be handled at least as well as in the saturated network, given that there are also some not high-priority, e.g., medium, low, and best effort, requests attempting to use the link. Using the «niceset» test-set, a non-saturated network can be emulated when using the 300 Kbit link. The «hammering» test set will saturate both network links.

The rest of this subsection will show how the broker behaves with the different queuing options. In the following figures, the numbers 1 to 32 correspond to the queuing options as seen in Table 4.7. The results from not using the broker are indicated by the «No broker» columns.

niceset

The niceset test set contains, as seen in Appendix B.1, 8 requests. First 4 Camera requests are issued, with a 3 seconds pause between each of them, and an additional 10 second pause after the last one, before 4 GPS requests are issued, each with a 0.3 second pause in between. The time between the first and last request is just shy of 20 seconds, and the test set is finished when all clients are either finished or preempted.

As seen in Figure 4.3, the «niceset» test set always succeeds with PrimaryOnly. For all other networks, there exists at least one option set where one or more clients failed. Queuing options 1 to 16 disable queuing, and options 8 to 16 will unconditionally preempt lower priority clients. The figure clearly shows the drawback of disabling queuing, where columns 1 to 16 are significantly lower than columns 17 to 32. Furthermore, column 8 to 16 shows the drawback of unconditionally preempting lower priority client. Combined with the strict enforcement of the timeslots in option sets 11, 12, 15 and 16, then no clients passed in neither BackupOnly nor BackupThenPrimary. Not using the broker is equal or better than more than half of the non-queuing option sets.

When limiting to only high-priority clients, only 2 in niceset, then the results are almost at the same level, though the option sets with queuing enabled are best, as seen in Figure 4.4.

hammering

The hammering test set contains 100 requests, with 5 being high priority, 15 medium priority, 30 low priority and 50 best effort. The requests are a mixture of GPS and Camera requests, with a 0.1 to 5 seconds pause between each request. 137 seconds between first and last request, thus with the emulated VaryingNetwork network, 3 downgrades and 3 upgrades will be performed.

Figure 4.5 shows that 3 option sets, 9, 10, and 14, manage to satisfy all high priority requests on every network, while a fourth one, 13, lacks one client. The common

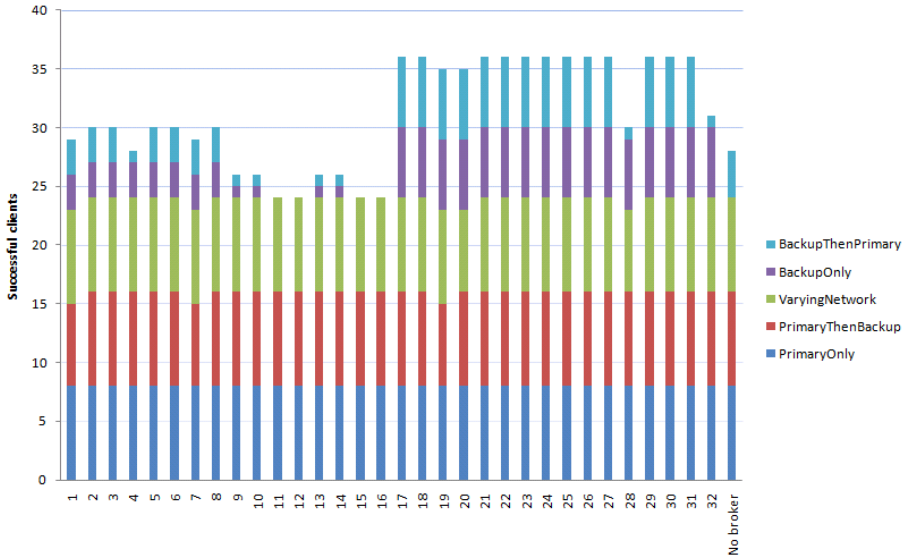

Figure 4.3.: Successful clients in *niceset* with any priority. Max is 8 per emulated network, 40 in total.

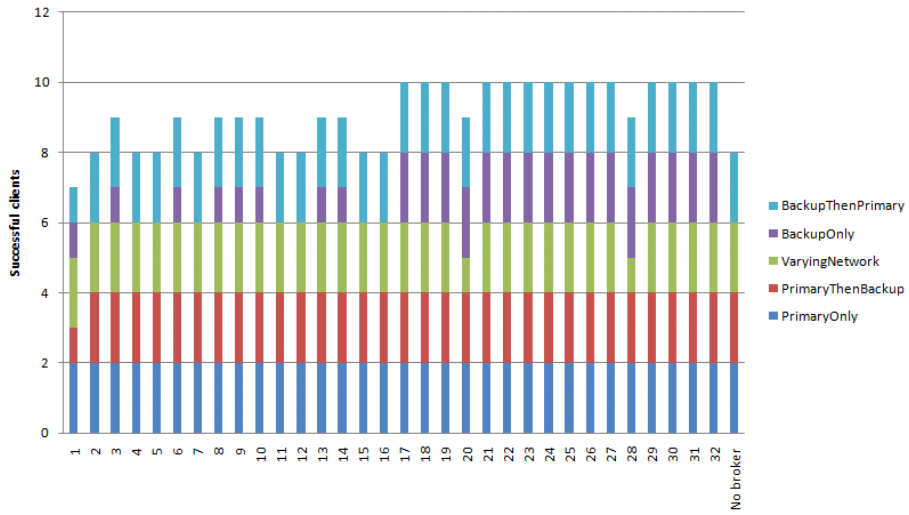

Figure 4.4.: Successful clients in *niceset* with high priority. Max is 2 per emulated network, 10 in total.

options for these four are that queuing is off, preemption is on and time slot enforcing is off. This combination clearly yields the best experience for high priority users.

The other distinctive feature is that 28 performs as bad as without the broker, while 27 has one more successful client. 27 and 28 have time slot enforcing enabled, which clearly has a negative effect on the success rate of the high-priority clients. The same can be seen for option set 11 and 12, which are between the four best performing option sets.

As seen in Table 4.7, time slot enforcing is enabled in every other pair, starting with 3 and 4, and continuing with 5 and 6, etc. In Figure 4.6, the highly negative outcome of this option is easily spotted when comparing 11, 12, 15, and 16 with 9, 10, 13, and 14, which are option sets with queuing disabled and preemption of lower priority clients enabled. The reason why they have a very low client success rate, is due to Python's lack of thread preemption. The asynchronous solution used in this implementation will not be able to stop clients immediately, but they will instead continue to consume network resources a short while. The broker assumes these resources are available. Therefore, the newly added client to the ring buffer will not be able to finish as early as predicted, and will instead be preempted by the ring buffer's management thread.

In Figure 4.6 one can see that queuing is helping when more requests with different priority are taken into consideration. Still 9, 10, 13 and 14 perform best by far, though the queuing half is catching in. This figure is the only one where all options are better than running without the broker.

Figure 4.7 shows the trend that queuing is better for a larger set of priorities, and it is the first figure where all but one of the queuing option sets perform better than the non-queuing option sets.

In Figure 4.8 one can see that when all four levels of priorities are considered, the queuing option sets are vastly superior to the non-queuing option sets. It could seem that not using the broker will perform better than almost all non-queuing option sets, and clearly better than 11, 12, 15, and 16. This is not true, as looking at the «no broker» column more closely one will see that the PrimaryOnly network covers a huge fraction of the successful clients, and that most non-queuing option sets perform better is all the other networks.

Also as in Figure 4.6 and 4.7, the time slot enforcing enabled in option sets 27, 28, 31, and 32, are clearly limiting the client satisfaction success rate.

Table 4.9 shows the results from the hammering test set applied to the BackupOnly network without using the broker. Only a single GPS request was successful, while all other requests were unsuccessful.

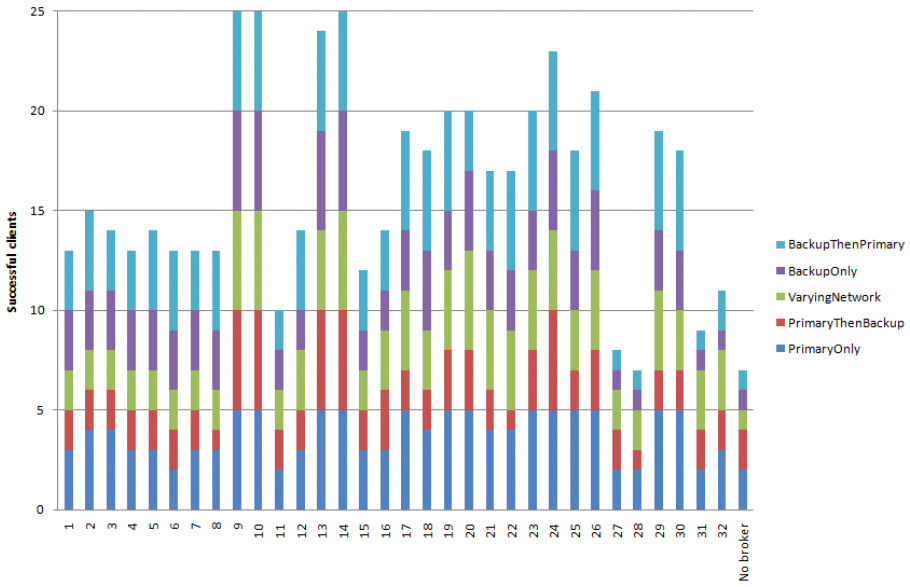

Figure 4.5.: Successful clients in *hammering* with high priority. Max is 5 per emulated network, 25 in total.

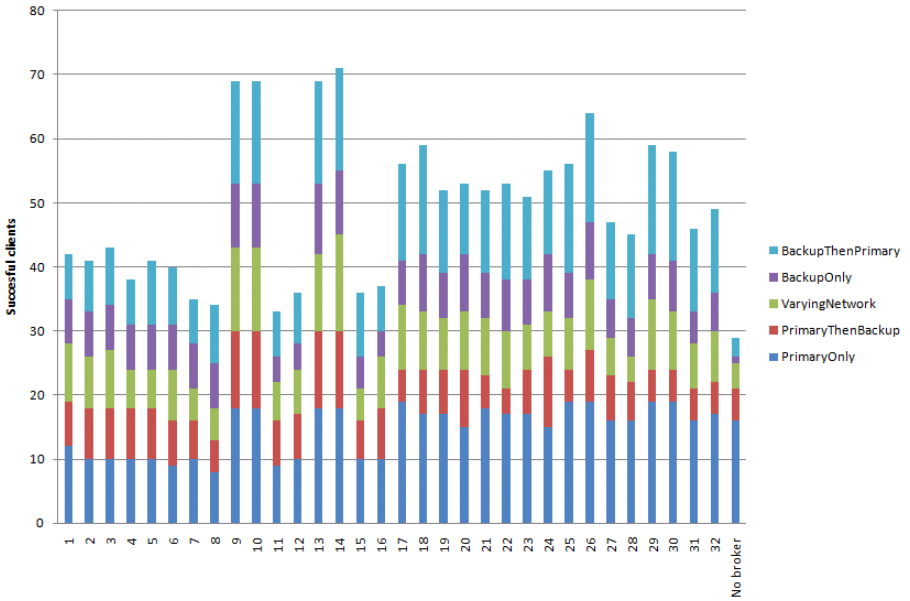

Figure 4.6.: Successful clients in *hammering* with high and medium priority. Max is 20 per emulated network, 100 in total.

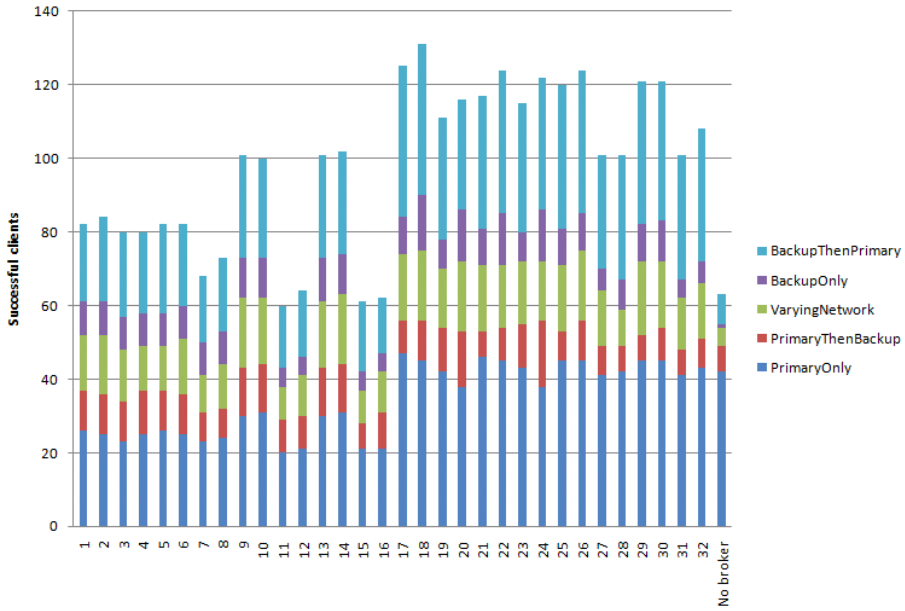

Figure 4.7.: Successful clients in *hammering* with high, medium and low priority. Max is 50 per emulated network, 250 in total.

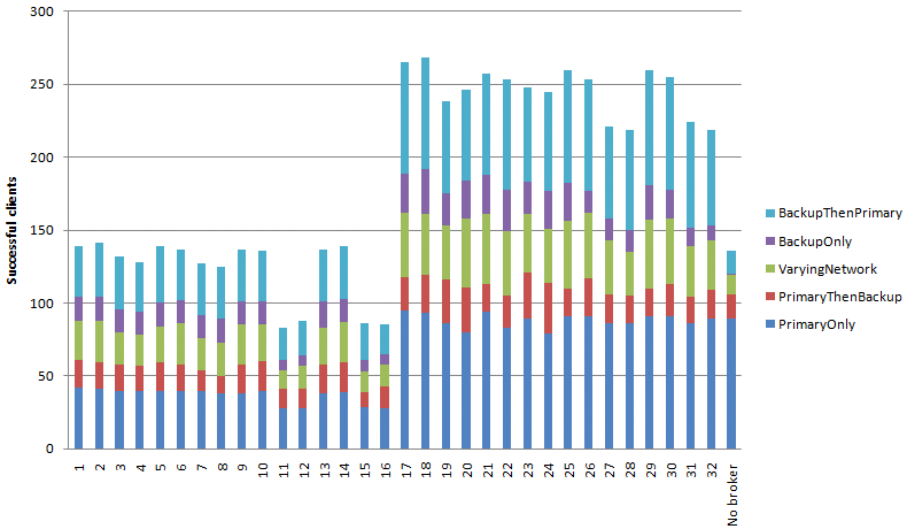

Figure 4.8.: Successful clients in *hammering* with any priority. Max is 100 per emulated network, 500 in total.

Service	Priority	#	Status						
			Success	Rejected	Too late	Fail downgrade	Exceeded slots	Terminated	Failed
GPS	High	3	33.33	0.00	0.00	0.00	0.00	0.00	66.67
	Medium	0	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Low	9	0.00	0.00	11.11	0.00	0.00	0.00	88.89
	BE	25	0.00	0.00	8.00	0.00	0.00	0.00	92.00
Camera-High	High	2	0.00	0.00	0.00	0.00	0.00	0.00	100.00
	Medium	13	0.00	0.00	7.69	0.00	0.00	0.00	92.31
	Low	20	0.00	0.00	5.00	0.00	0.00	0.00	95.00
	BE	22	0.00	0.00	13.64	0.00	0.00	0.00	86.36
Camera-Medium	High	0	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Medium	2	0.00	0.00	0.00	0.00	0.00	0.00	100.00
	Low	0	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	BE	0	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Camera-Low	High	0	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Medium	0	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Low	1	0.00	0.00	0.00	0.00	0.00	0.00	100.00
	BE	3	0.00	0.00	0.00	0.00	0.00	0.00	100.00

Table 4.9.: Request status. No broker, network: BackupOnly, testset: hammering

4.4. Result overview

In the evaluation given in the previous section, five different queuing options has been tested under both high and low load, and for several different network behaviors. When, and if, the different queuing options should be used will depend on the policies of the networks employing these mechanisms.

If policy dictates that high client satisfaction for high priority clients is the most important metric, then queuing should be turned off, always preempt lower priority clients turned on, and time slot enforcing turned off. This corresponds to option sets 9, 10, 13 and 14, and as seen in Figure 4.5 these options will give high satisfaction for high priority clients.

On the other hand, if optimizing for highest overall client satisfaction, then option set 18 should be selected. Options set 17, was almost as good as 18, but as it did not have Token Bucket enabled for small requests it became the second best.

The major difference between maximizing for overall- and high-priority client satisfaction is that overall satisfaction requires queuing to be enabled, as seen in Figure 4.8, where 1 to 16 do not have queuing, while 17 to 32 have queuing.

5. Conclusion

The goal of this thesis was to create a QoS-aware admission control mechanism for Web Services, based on the requirements from the NNEC FS. Among them are, role based prioritization of messages and support for preemption (as discussed in Section 3.2.2). The work was focused around the two claims from Chapter 1:

1. *QoS admission control can be implemented in a broker.*

2. *A role based QoS combined with a broker will lead to higher client satisfaction.*

Claim 1 was shown probable by the research described in Section 2.6. An admission control broker was then designed in Chapter 3 and the implementation in Chapter 4 proved this to be true.

In order to test claim 2, a series of experiments was performed, in which five different options for QoS handling were described and tested. These options were tested both alone and in combination with each other, in order to determine which option configuration(s) best fulfill claim 2. The tests were performed under varying network load and network behavior, and showed that which options are most suitable will depend on the optimization criteria that policy dictates for each network these mechanisms are applied to.

Tests showed that while the client satisfaction varied with the options used, almost all option combinations yielded better client satisfaction than not using the broker, thus proving claim 2.

In summary, the goal was reached.

6. Future work

This chapter contains an overview of ideas which arose while writing this thesis, but were either outside the scope or skipped because of insufficient time.

6.1. Network

Currently the broker is only available through a Python object, which is a show stopper for non-local clients and local clients not written in Python. Therefore the interface should be available as a Web Service, to provide a platform independent broker.

As discussed in Section 4.1.1, Python's threading module cannot immediately stop threads, so if a broker shall support local clients, another programming language with better threading support should be used.

Both the client and the server in the test rig used TCP's New Reno congestion control. Many other variants exists, each with different focus, e.g., high delay, wireless, etc. An evaluation can be done to see if another congestion control algorithm should be used.

One could also evaluate if only high priority clients should use the token bucket mechanism, as then small, but important, requests could be expedited side-by-side with the queued requests and thus cause less disruption for lower priority users.

If a client can be modified to issue a request to the broker before it invokes a service, then the client should also be able to report back to the broker when its request has finished, thus prune itself from the ring buffer. This should not be a requirement, because the broker must be able to continue even if a client looses radio connectivity.

6.2. Security

Information contained in a SOAP message used in Web Services may be subject to integrity and confidentiality requirements. Furthermore as intermediate nodes may process or even modify the message, traditional end-to-end encryption as SSL/TLS, will be broken, as seen in Figure 6.1.

To meet security with flexibility demands, OASIS and W3C have standardized several specifications for security in XML itself and when utilized in Web Services.

6.2.1. XML security

The broker is compatible with COTS SOAP, therefore it supports standards which extend SOAP. If application layer security gets approved in the future, it can be utilized in the following ways:

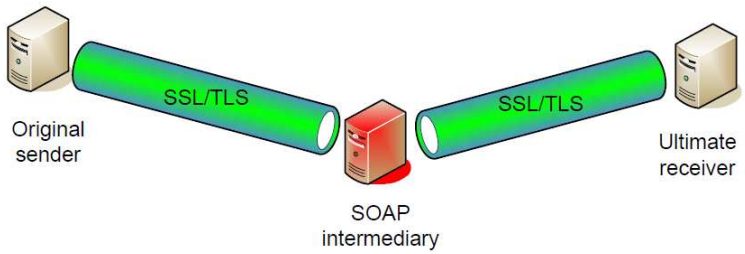

Figure 6.1.: The provided transport layer security is broken at the intermediary SOAP node. From [51].

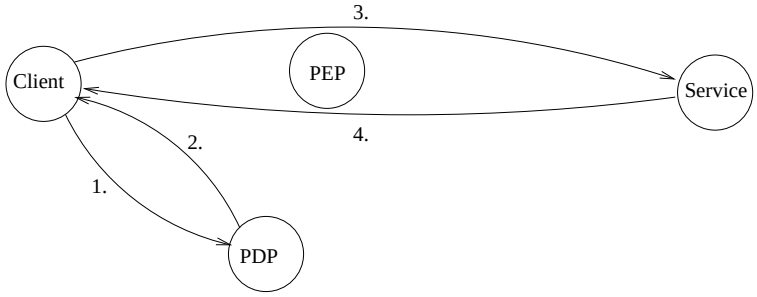

Figure 6.2.: The four steps to invoke a service through a policy decision point.

XML security is handled with XML Signature, first standardized by OASIS in [3] and later by IETF in [16], and XML Encryption [32], which are both well established standards.

- «The use of digital signatures is a common method for ensuring message integrity, authentication and non-repudiation» [51]. XML Signature provides a standard interoperable format for signing XML. An XML signature may cover multiple resources in an XML document, both plain-text and binary elements. For each resource a digest will be added, while only one signature is added. SHA-1 and DSA with SHA-1, are the only required algorithms to be supported for respectively digest and signature, though many implementations support stronger algorithms, interoperable issues will set limitations.

- XML Encryption can encrypt both parts of, and the entire XML document, depending on the confidentiality requirements. The specifications requires support for 3DES, AES-128 and AES-256.

Both encrypting and signing needs a way to manage keys in a scalable manner. XML Key Management Specification (XKMS) [27] does well in normal civilian networks, but is not ideal in a disadvantaged military network, due to the need for decentralization.

The concept of dividing functionality into a DP and an EP is a well known approach, but Policy Decision Point (PDP) and Policy Enforcement Point (PEP) are the standardized terms for these concepts. It would be convenient to use these security standards to express the roles needed for QoS DPs and EPs, and more. A PDP and a PEP are shown in Figure 6.2.

A more thorough discussion of XML and Web Services security can be found in [51].

Appendices

A. Vyatta configuration

A.1. Router A

```
interfaces {
    ethernet eth0 {
        address 10.10.100.1/24
        duplex auto
        hw-id 00:1e:ec:29:b4:c8
        ip {
            ospf {
                dead-interval 3
                hello-interval 1
                priority 1
                retransmit-interval 5
                topology 1 {
                    cost 10
                }
                topology 2 {
                    cost 10
                }
                transmit-delay 1
            }
        }
        smp_affinity auto
        speed auto
    }
    ethernet eth1 {
        address 10.10.200.100/25
        duplex auto
        hw-id 00:a1:b0:50:05:4c
        ip {
            ospf {
                dead-interval 3
                hello-interval 1
                priority 1
                retransmit-interval 5
                topology 1 {
                    cost 10
                }
                topology 2 {
                    cost 10
                }
```

```
                transmit-delay 1
            }
        }
        smp_affinity auto
        speed auto
    }
    ethernet eth2 {
        address 10.10.200.200/25
        duplex auto
        hw-id 00:a1:b0:50:06:17
        ip {
            ospf {
                dead-interval 3
                hello-interval 1
                priority 1
                retransmit-interval 5
                topology 2 {
                    cost 100
                }
                transmit-delay 1
            }
        }
        smp_affinity auto
        speed auto
    }
    loopback lo {
        address 10.10.10.1/32
        ip {
            ospf {
                dead-interval 3
                hello-interval 1
                priority 1
                retransmit-interval 5
                topology 1 {
                    cost 10
                }
                topology 2 {
                    cost 10
                }
                transmit-delay 1
            }
        }
    }
}
protocols {
    ospf {
        area 0 {
            mt-only
            network 10.10.100.0/24
```

```
            network 10.10.200.0/25
            network 10.10.200.128/25
            network 10.10.10.1/32
        }
        parameters {
            abr-type cisco
            router-id 10.10.10.1
        }
    }
}
service {
    ssh {
        allow-root
        port 22
        protocol-version v2
    }
}
system {
    host-name laptop
    login {
        user root {
            authentication {
                encrypted-password $1$BopacY7Z$uO627VlxSyvM63fiaz/SB1
            }
            level admin
        }
        user vyatta {
            authentication {
                encrypted-password $1$bs0BXBV2$HQVGRoeMe24J/hx/OnBax0
            }
            level admin
        }
    }
    ntp-server 0.vyatta.pool.ntp.org
    package {
        auto-sync 1
        repository community {
            components main
            distribution stable
            password ""
            url http://packages.vyatta.com/vyatta
            username ""
        }
    }
    syslog {
        global {
            facility all {
                level notice
            }
```

```
            facility protocols {
                level debug
            }
        }
    }
    time-zone Europe/Oslo
}
topology 1 {
    description Pri1
    priority 100
    traffic-class 1 {
    }
}
topology 2 {
    description Pri2
    priority 200
    traffic-class 0 {
    }
}


/* Warning: Do not remove the following line. */
/* === vyatta-config-version: "quagga@1:nat@3:cluster@1:wanloadbalance@2:
   dhcp-relay@1:dhcp-server@4:ipsec@2:webproxy@1:firewall@3:system@3:
   webgui@1:vrrp@1" === */
/* The three lines above must be remerged, as the line was wrapped to
   please \verbatiminput in latex. */
/* Release version: 999.deltattr.06092242 */
```

A.2. Router B

```
interfaces {
    ethernet eth0 {
        address 10.10.0.1/24
        duplex auto
        hw-id 00:0b:cd:73:19:b9
        ip {
            ospf {
                dead-interval 3
                hello-interval 1
                priority 1
                retransmit-interval 5
                topology 1 {
                    cost 10
                }
                topology 2 {
                    cost 10
                }
```

```
            transmit-delay 1
        }
    }
    smp_affinity auto
    speed auto
}
ethernet eth1 {
    address 10.10.200.101/25
    duplex auto
    hw-id 00:a1:b0:10:03:53
    ip {
        ospf {
            dead-interval 3
            hello-interval 1
            priority 1
            retransmit-interval 5
            topology 1 {
                cost 10
            }
            topology 2 {
                cost 10
            }
            transmit-delay 1
        }
    }
    smp_affinity auto
    speed auto
}
ethernet eth2 {
    address 10.10.200.201/25
    duplex auto
    hw-id 00:c0:f0:46:07:f7
    ip {
        ospf {
            dead-interval 3
            hello-interval 1
            priority 1
            retransmit-interval 5
            topology 2 {
                cost 100
            }
            transmit-delay 1
        }
    }
    smp_affinity auto
    speed auto
}
loopback lo {
    address 10.10.10.2/32
```

```
        ip {
            ospf {
                dead-interval 3
                hello-interval 1
                priority 1
                retransmit-interval 5
                topology 1 {
                    cost 10
                }
                topology 2 {
                    cost 10
                }
                transmit-delay 1
            }
        }
    }
}
protocols {
    ospf {
        area 0 {
            mt-only
            network 10.10.0.0/24
            network 10.10.200.0/25
            network 10.10.200.128/25
            network 10.10.10.2/32
        }
        parameters {
            abr-type cisco
            router-id 10.10.10.2
        }
    }
}
service {
    ssh {
        allow-root
        port 22
        protocol-version v2
    }
}
system {
    host-name desktop
    login {
        user root {
            authentication {
                encrypted-password $1$BopacY7Z$uO627VlxSyvM63fiaz/SB1
            }
            level admin
        }
        user vyatta {
```

```
            authentication {
                encrypted-password $1$bs0BXBV2$HQVGRoeMe24J/hx/OnBax0
            }
            level admin
        }
    }
    ntp-server 0.vyatta.pool.ntp.org
    package {
        auto-sync 1
        repository community {
            components main
            distribution stable
            password ""
            url http://packages.vyatta.com/vyatta
            username ""
        }
    }
    syslog {
        global {
            facility all {
                level notice
            }
            facility protocols {
                level debug
            }
        }
    }
    time-zone Europe/Oslo
}
topology 1 {
    description Pri1
    priority 100
    traffic-class 1 {
    }
}
topology 2 {
    description Pri2
    priority 200
    traffic-class 0 {
    }
}


/* Warning: Do not remove the following line. */
/* === vyatta-config-version: "webproxy@1:quagga@1:firewall@3:
   dhcp-relay@1:nat@3:dhcp-server@4:wanloadbalance@2:webgui@1:
   system@3:cluster@1:ipsec@2:vrrp@1" === */
/* The three lines above must be remerged, as the line was wrapped to
   please \verbatiminput in latex. */
```

```
/* Release version: 999.deltattr.06092242 */
```

B. Various scripts

B.1. The «niceset» test set

```
niceset = (
'niceset',
('chef', TL_CHEF_CAMERA, (CameraHigh,)),
3,
('planner', TL_PLANNER_CAMERA, (CameraHigh,CameraMed)),
3,
('recon', TL_RECON_CAMERA, (CameraHigh,CameraMed,CameraLow)),
3,
('gunner', TL_GUNNER_CAMERA, (CameraHigh,CameraMed,CameraLow)),
10, # sum section, 19 seconds
('chef', TL_CHEF_GPS, (GPS,)),
0.3,
('planner', TL_PLANNER_GPS, (GPS,)),
0.3,
('chef', TL_CHEF_GPS, (GPS,)),
0.3,
('recon', TL_RECON_GPS, (GPS,)),
2,)
```

B.2. ITR

B.2.1. Link shaper

```
#!/bin/bash
#  From: http://www.topwebhosts.org/tools/traffic-control.php
#  Copyright: Scott Seong, 2006 - 2010 Broadband Media Inc.
#  Accessed: 21st December 2010
#
#  Modified by Øyvind Kolbu to support multiple interfaces with
#  different link speeds.
#
#  tc uses the following units when passed as a parameter.
#  kbps: Kilobytes per second
#  mbps: Megabytes per second
#  kbit: Kilobits per second
#  mbit: Megabits per second
#  bps: Bytes per second
#       Amounts of data can be specified in:
```

```
#        kb or k: Kilobytes
#        mb or m: Megabytes
#        mbit: Megabits
#        kbit: Kilobits
#  To get the byte figure from bits, divide the number by 8 bit
#
#
# Name of the traffic control command.
TC=/sbin/tc

# Networks
CLIENTNET=10.10.0.0/24
SERVERNET=10.10.100.0/24

start() {
    for IF in $@;
    do
        case "$IF" in
            eth1)
                DNLD=300kbit
                UPLD=300kbit
                ;;
            eth2)
                DNLD=16kbit
                UPLD=16kbit
                ;;
            *)
                echo "Unknown interface: $IF"
                exit 1
                ;;
        esac
# Filter options for limiting the intended interface.
        U32="$TC filter add dev $IF protocol ip parent 1:0 prio 1 u32"
# We'll use Hierarchical Token Bucket (HTB) to shape bandwidth.
# For detailed configuration options, see Linux man page
        $TC qdisc add dev $IF root handle 1: htb default 30
        $TC class add dev $IF parent 1: classid 1:1 htb rate $DNLD
        $TC class add dev $IF parent 1: classid 1:2 htb rate $UPLD
        $U32 match ip dst $CLIENTNET flowid 1:1
        $U32 match ip dst $SERVERNET flowid 1:1
        $U32 match ip src $CLIENTNET flowid 1:2
        $U32 match ip src $SERVERNET flowid 1:2
    done
# The first line creates the root qdisc, and the next two lines
# create two child qdisc that are to be used to shape download
# and upload bandwidth.
#
# The four last lines creates the filters to match the interface.
# The 'dst' IP address is used to limit download speed, and the
```

```
# 'src' IP address is used to limit upload speed.
}

stop() {
# Stop the bandwidth shaping.
    for IF in $@; do
        $TC qdisc del dev $IF root
    done
}

restart() {
    stop $@
    sleep 1
    start $@
}

show() {
# Display status of traffic control status.
    for IF in $@;
    do
        $TC -s qdisc ls dev $IF
    done
}

usage () {
    echo "Usage: $0 {start|stop|restart|show} if [if2 ...]"
    exit 1
}

[ $# -lt 3 ] && usage

cmd=$1
shift

case "$cmd" in
  start)
    echo -n "Starting bandwidth shaping for $@: "
    start $@
    echo "done"
    ;;
  stop)
    echo -n "Stopping bandwidth shaping for $@: "
    stop $@
    echo "done"
    ;;
  restart)
    echo -n "Restarting bandwidth shaping for $@: "
    restart $@
    echo "done"
```

```
    ;;
  show)
    echo "Bandwidth shaping status for $@:"
    show $@
    echo ""
    ;;
  *)
    usage
    ;;
esac
```

B.2.2. HTTP daemon

```
#!/usr/bin/env python
# coding: latin-1
#
# Author: Øyvind Kolbu, 2010-2011, oyvink@ifi.uio.no

"""HTTP daemon which can return the current used interface, as
well as disable and enable primary and backup interfaces. """

import re
import shlex, subprocess
import sys
import time

from BaseHTTPServer import BaseHTTPRequestHandler,HTTPServer

# Get the Broker module
sys.path.append('..')

import Broker.Constants as co
from Broker import Logger

## The interfaces between the two tactical routers
IF_PRIMARY='eth1'
IF_BACKUP='eth2'

## Config for extracting routing info
IP_RE=r'\b(?:\d{1,3}\.){3}\d{1,3}\b'
REMOTE_NET='10.10.100.0/24'
route_re = r'^%s via %s dev ' % (REMOTE_NET, IP_RE)

re_primary = re.compile(route_re + IF_PRIMARY, re.MULTILINE)
re_backup = re.compile(route_re + IF_BACKUP, re.MULTILINE)

## Config for simulating network link loss, i.e., by
## blocking OSPF updates
IPTABLES='/sbin/iptables'
BLOCK_IN = IPTABLES + ' -A INPUT -i %s -j DROP'
```

```
BLOCK_OUT = IPTABLES + ' -A OUTPUT -o %s -j DROP'
BLOCK_FORWARD_IN = IPTABLES + ' -A FORWARD -i %s -j DROP'
BLOCK_FORWARD_OUT = IPTABLES + ' -A FORWARD -o %s -j DROP'
RULES_BLOCK = (BLOCK_IN, BLOCK_OUT,
               BLOCK_FORWARD_IN, BLOCK_FORWARD_OUT)

UNBLOCK_IN = BLOCK_IN.replace(' -A ', ' -D ')
UNBLOCK_OUT = BLOCK_OUT.replace(' -A ', ' -D ')
UNBLOCK_FORWARD_IN = BLOCK_FORWARD_IN.replace(' -A ', ' -D ')
UNBLOCK_FORWARD_OUT = BLOCK_FORWARD_OUT.replace(' -A ',' -D ')
RULES_UNBLOCK = (UNBLOCK_IN, UNBLOCK_OUT,
                 UNBLOCK_FORWARD_IN, UNBLOCK_FORWARD_OUT)

class RouterHandler(BaseHTTPRequestHandler):

    def do_GET(self):
        # Skip the first /
        cmd = self.path[1:]

        if cmd in ('routeInfo',      'flushIPTables',
                   'disablePrimary', 'enablePrimary',
                   'disableBackup',  'enableBackup',
                   'disableBoth',    'enableBoth',):
            self.send_response(200)
            self.send_header('Content-type', 'text/html')

            resp = getattr(self, cmd)()
            if resp == None:
                resp = 'OK'
        else:
            Logger.log('Unknown command: ' + cmd)
            self.send_response(404)
            self.send_header('Content-type', 'text/html')
            resp = 'Unknown request: "%s"' % self.path

        print 'resp, "%s"' % resp
        self.end_headers()
        self.wfile.write(resp)

    def routeInfo(self):
        ROUTE_CMD='/sbin/ip route show table 2'
        args = shlex.split(ROUTE_CMD)
        p = subprocess.Popen(args, stdout=subprocess.PIPE).\
          communicate()[0]

        if re_primary.search(p):
            return co.NETWORK_LINK_PRIMARY
        elif re_backup.search(p):
            return co.NETWORK_LINK_BACKUP
```

```
            else:
                Logger.log('No link. Input was: %s' % p)
                return co.NETWORK_LINK_NONE

        def flushIPTables(self):
            subprocess.call(shlex.split('/sbin/iptables -F'))

        def _filterInterface(self, interface, rules):
            for cmd in rules:
                args = shlex.split(cmd % interface)
                err = subprocess.Popen(args, stderr=subprocess.PIPE).\
                  communicate()[1]
                # Ignore duplicate rule removal
                if err and \
                  'iptables: Bad rule (does a matching rule exist' not in err:
                    Logger.error("%s, %s, %s" % (interface, cmd, err))

        def _disableInterface(self, interface):
            self._filterInterface(interface, RULES_BLOCK)

        def _enableInterface(self, interface):
            self._filterInterface(interface, RULES_UNBLOCK)

        def disablePrimary(self):
            self._disableInterface(IF_PRIMARY)

        def enablePrimary(self):
            self._enableInterface(IF_PRIMARY)

        def disableBackup(self):
            self._disableInterface(IF_BACKUP)

        def enableBackup(self):
            self._enableInterface(IF_BACKUP)

        def disableBoth(self):
            self.disablePrimary()
            self.disableBackup()

        def enableBoth(self):
            self.enablePrimary()
            self.enableBackup()

    def main():
        try:
            server = HTTPServer(('', 80), RouterHandler)
            print 'starting Router http server'
            server.serve_forever()
        except KeyboardInterrupt:
```

```
        print 'aborting...'
        server.socket.close()

if __name__ == '__main__':
    import sys
    sys.exit(main())
```

B.3. Handover delay

```
#coding: latin-1
#
#Copyright (c) 2010-2011 Øyvind Kolbu

"""
Measure handover delay from Primary -> Backup, and vice versa.
"""

import random
import shlex
import subprocess
import sys

from time import time,sleep

sys.path.append('..')

import Broker.Constants as co
import Broker.Utils

from Broker import Logger
from Broker.QoSBroker import BrokerException,Service

from lib.Thread2 import Thread

log = Logger.getLogger('handover-delay')
pinglog = Logger.getLogger('pinglog', timestamp=False)

class test(Thread):

    def __init__(self, filename):
        Thread.__init__(self)
        self.t = Broker.Utils.TCPdump(filename)

    def run(self):
        PING_CMD = shlex.split('/sbin/ping -c 3500 -i 0.05 %s' \
          % co.WS_SERVER)

        p = subprocess.Popen(PING_CMD, stdout=subprocess.PIPE)
```

```
            log('waiting for ping..')
            output = p.communicate()[0]
            pinglog(output)
            p.wait()
            log(output)
            log('done waiting.')
            self.t.stop()

changes = []
networklog = Logger.getLogger('networkinfo')
def mynetworklog(msg):
    networklog(msg)
    changes[-1].append(time())
    assert len(changes[-1]) == 2

def myChangeSpeed(url):
    Broker.Utils.changeSpeed(url)
    changes.append([time()])

broker = Service()
# Reset network to pristine condition
Broker.Utils.changeSpeed(co.ROUTER_URL_FLUSH_IPTABLES)
sleep(7)
broker.networkInfoUpdater.log = mynetworklog

t = test('ospf-1-3-0.05-iptables')
t.start()
for i in xrange(10):
    print "TESTSET", i
    myChangeSpeed(co.ROUTER_URL_DISABLE_PRIMARY)
    #sleep(random.uniform(38,50)) # OSPF 10-40
    sleep(random.uniform(4,7)) # OSPF 1-3
    myChangeSpeed(co.ROUTER_URL_FLUSH_IPTABLES)
    sleep(random.uniform(6,10)) # OSPF 1-3
    #sleep(random.uniform(38,50)) # OSPF 10-40
t.join()
broker.stop()

downgrade = []
upgrade = []
for i in xrange(len(changes)):
    print i, changes[i]
    diff = changes[i][1] - changes[i][0]
    print i, changes[i], diff
    assert diff > 0
    if i%2:
        upgrade.append(diff)
    else:
        downgrade.append(diff)
```

```
for text, data in (('Upgrade', upgrade), ('Downgrade', downgrade)):
    print ['%.2f' % i for i in data ]
    print '%s: %s' % (text, Broker.Utils.stats(data))
```

B.4. Parse ping log

```
#coding: latin-1
#
#Copyright (c) 2010-2011 Øyvind Kolbu

from __future__ import with_statement

import sys
sys.path.append('..')

from Broker.Utils import stats

"""Finds gaps in ping times, and thus the rerouting time"""

def usage(msg):

if len(sys.argv) < 3:
    print >>sys.stderr, 'Error: missing arguments'
    print >>sys.stderr, 'Usage: %s interval pinglog' % sys.argv[0]
    sys.exit(1)

interval=float(sys.argv[1])
times = []
with open(sys.argv[2], 'r') as f:
    prev = False
    for line in f.readlines():
        line = line.rstrip('\n')
        if not line:
            break
        # pinglog format:
        #PING 10.10.100.3 (10.10.100.3): 56 data bytes
        #64 bytes from 10.10.100.3: icmp_seq=0 ttl=62 time=0.918 ms
        #64 bytes from 10.10.100.3: icmp_seq=1 ttl=62 time=0.957 ms
        #64 bytes from 10.10.100.3: icmp_seq=2 ttl=62 time=0.820 ms
        icmp_seq=line.split()[4]
        if icmp_seq == 'data':
            continue
        seq=int(icmp_seq.split('=')[1])

        if prev == False:
            pass
        elif prev+1 < seq:
            diff=seq-prev
```

```
            assert diff > 1
            timediff = diff*interval
            print timediff,diff, seq, prev
            times.append(timediff)
        prev = seq

print '%s: %s' % (sys.argv[2],stats(times))
```

B.5. Measure bytes sent and received

The following script lists the difference between the bytes sent and received vs XML request and response size vs the answer part of the XML response, i.e., the actual result from the Web service.

```
import subprocess
import shlex
import sys
import time

sys.path.append('..')

import Broker.Constants as co

from Broker.Utils import TCPdump,stats
from Services.GPS import GPS
from Services.Camera import CameraLow,CameraMed,CameraHigh


CMD_SETUP = ('/sbin/kldload ipfw',
             '/sbin/ipfw -q add 65000 allow ip from any to any',)

CMD_START = ('/sbin/ipfw -q delete 100',
             '/sbin/ipfw -q delete 101',
             '/sbin/ipfw -q add 100 count ip from any to %s' %\
               co.WS_SERVER,
             '/sbin/ipfw -q add 101 count ip from %s to any' %\
               co.WS_SERVER,)

CMD_IN_BYTES = ('/sbin/ipfw show 101',)
CMD_OUT_BYTES = ('/sbin/ipfw show 100',)

TESTS = (GPS, CameraLow, CameraMed, CameraHigh,)

def _run(cmds):
    ret = []
    for cmd in cmds:
        args = shlex.split(cmd)
        ret.append(subprocess.Popen(args, stdout=subprocess.PIPE).\
          communicate()[0])
    return ret
```

```
_run(CMD_SETUP)
for t in TESTS:

    recv = []
    sent = []
    sizes = []
    times = []
    for i in xrange(10):
        _run(CMD_START)

        test = t()
        pre = time.time()
        test.start()
        test.join()
        post = time.time()
        times.append(post-pre)

        inbytes = _run(CMD_IN_BYTES)[0].split()[2]
        recv.append(int(inbytes))
        outbytes = _run(CMD_OUT_BYTES)[0].split()[2]
        sent.append(int(outbytes))
        print test.name, 'recv', inbytes, 'sent', outbytes
        if isinstance(test,GPS):
            sizes.append(sum(map(len,test.result.values())))
        else:
            sizes.append(len(test.result))

    print test.name
    for text, data in (('Received', recv), ('Sent', sent), \
                       ('Time', times)):
        print text, stats(data)
    print 'times', times

    print test.name, 'xml request', len(test._client.xml_request)
    print test.name, 'xml response', len(test._client.xml_response)
    if isinstance(test,GPS):
        size = sum(map(len,test.result.values()))
    else:
        size = len(test.result)
    assert size*10 == sum(sizes)
    print 'answer size: %s' % size
```

B.6. Web camera capturer

Takes a picture with the web camera periodically and store it in three resolutions: 800x600, 640x480 and 320x240.

```
#!/usr/bin/env python
# coding: latin-1
#
# Author: Øyvind Kolbu, 2010-2011, oyvink@ifi.uio.no

"""Takes a picture with the web camera peridically and store
it in three resolutions: 800x600, 640x480 and 320x240."""

from __future__ import with_statement

import fcntl
import os
import shlex, subprocess
import time

IMAGES = ('output-high.jpg','output-medium.jpg', 'output-low.jpg')
INTERVAL = 2 # How often to update the images
MIN_SLEEP = 0.5
CMD = ('/usr/bin/fswebcam -d /dev/video1 800x600 %s.tmp ' +\
       '--scale 640x480 %s.tmp --scale 320x240 %s.tmp') %\
      IMAGES

# Run until aborted
while 1:
    pre = time.time()
    out, err = subprocess.Popen(shlex.split(CMD), \
      stderr=subprocess.PIPE).communicate()
    for filename in IMAGES:
        with open(filename+'lock', 'a') as f:
            try:
                # We can afford to wait for an exclusive lock, because
                # the web service will lock the file very briefly while
                # reading it to memory.
                fcntl.flock(f.fileno(), fcntl.LOCK_EX)
                try:
                    os.rename(filename + '.tmp', filename)
                except OSError, e:
                    print ('%s ERROR: Could not rename %s.tmp to %s'+\
                           ', skipping: %s') \
                          % (time.ctime(), filename, filename, e)
            except IOError, e:
                print "ERROR: Could not get lock for %s: %s" % \
                  (filename, e)
    post = time.time()
    sleep = max(INTERVAL - (post - pre), MIN_SLEEP)
    time.sleep(sleep)
```

Bibliography

[1] D. S. Alberts, J. J. Garstka, and F. P. Stein. *Network Centric Warfare Developing and Leveraging Information Superiority*. CCRP, 1999. http://www.dodccrp.org/files/Alberts_NCW.pdf, accessed 2011-01-25.

[2] J. Babiarz, K. Chan, and F. Baker. Configuration Guidelines for DiffServ Service Classes. RFC 4594 (Informational), Aug. 2006. Updated by RFC 5865.

[3] M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon. XML Signature Syntax and Processing (Second Edition). http://www.w3.org/TR/xmldsig-core/, accessed 2010-08-29.

[4] D. Black, S. Brim, B. Carpenter, and F. L. Faucheur. Per Hop Behavior Identification Codes. RFC 3140 (Proposed Standard), June 2001.

[5] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Service. RFC 2475 (Informational), Dec. 1998. Updated by RFC 3260.

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

[7] 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, 5946.

[8] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan. Extensible Markup Language (XML) 1.1 (Second Edition), W3C Recommendation 16 August 2006, edited in place 29 September 2006. http://www.w3.org/TR/xml11/, accessed 2010-01-25.

[9] R. Chinnici, Jean-Jacques, A. Ryman, and S. Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, W3C Recommendation 26th June 2007. http://www.w3.org/TR/wsdl20/, accessed 2010-01-25.

[10] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1, W3C Note 15th March 2001. http://www.w3.org/TR/wsdl, accessed 2010-01-25.

[11] Cisco. OSPF Support for Fast Hello Packets. http://www.cisco.com/en/US/docs/ios/12_0s/feature/guide/fasthelo.pdf, accessed 2011-01-03.

[12] T. Clausen and P. Jacquet. Optimized Link State Routing Protocol (OLSR). RFC 3626 (Experimental), Oct. 2003.

[13] S. Corson and J. Macker. Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations. RFC 2501 (Informational), Jan. 1999.

[14] B. Davie, A. Charny, J. Bennet, K. Benson, J. L. Boudec, W. Courtney, S. Davari, V. Firoiu, and D. Stiliadis. An Expedited Forwarding PHB (Per-Hop Behavior). RFC 3246 (Proposed Standard), Mar. 2002.

[15] P. J. Denning, D. Comer, D. Gries, M. C. Mulder, A. B. Tucker, A. J. Turner, and P. R. Young. Computing as a discipline. *Commun. ACM*, 32(1):9–23, 1989.

[16] D. Eastlake 3rd, J. Reagle, and D. Solo. (Extensible Markup Language) XML-Signature Syntax and Processing. RFC 3275 (Draft Standard), Mar. 2002.

[17] R. Faucher et al. Guidance on proxy servers for the tactical edge. The MITRE corporation, MITRE Technical Report, MTR 060175, September 2006.

[18] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785.

[19] T. Filiba. sebulba: recipe thread2. `http://sebulba.wikispaces.com/recipe+thread2`, accessed 2011-02-03.

[20] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782 (Proposed Standard), Apr. 2004.

[21] A. Gibb. Challenges for Middleware Imposed by the Tactical Army Communications Environment, 2003. `http://handle.dtic.mil/100.2/ADA485284`, accessed 2011-01-02.

[22] A. Gibb, H. Fassbender, M. Schmeing, J. Michalak, and J. E.Wieselthie. Information Management over Disadvantaged Grids, Final report of the RTO Information Systems Technology Panel, Task Group IST-030 / RTG-012, RTO-TR-IST-030. 2007. `http://www.rta.nato.int/pubs/rdp.asp?RDP=RTO-TR-IST-030`, accessed 2011-01-28.

[23] C. Griwodz and P. Halvorsen. Further Protocols with/-out QoS support, 2009. `http://www.uio.no/studier/emner/matnat/ifi/INF5071/h09/undervisningsmateriale/05-more-protocols.pdf`, accessed 2010-03-10.

[24] D. Grossman. New Terminology and Clarifications for Diffserv. RFC 3260 (Informational), Apr. 2002.

[25] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), W3C Recommendation 27th April 2007. `http://www.w3.org/TR/2007/REC-soap12-part1-20070427/`, accessed 2010-01-25.

[26] H. Haas and A. Brown. Web Services Glossary, W3C Working Group Note 11th February 2004. `http://www.w3.org/TR/ws-gloss/`, accessed 2010-01-26.

[27] P. Hallam-Baker and S. H. Mysore. XML Key Management Specification (XKMS 2.0), W3C Recommendation 28th June 2005. `http://www.w3.org/TR/xkms2/`, accessed 2010-08-29.

[28] M. Hauga and S. Haavik. Intelligent Tactical IP Router. *FFI-rapport 2009/01708*, 2009.

[29] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski. Assured Forwarding PHB Group. RFC 2597 (Proposed Standard), June 1999. Updated by RFC 3260.

[30] D. Holtwick. Issue 221115: Kill threads from main thread. `http://bugs.python.org/issue221115`, accessed 2011-02-03.

[31] J. Hylton et al. PEP 42 – Feature Requests. `http://www.python.org/dev/peps/pep-0042/`, accessed 2011-02-03.

[32] T. Imamura, B. Dillaway, and E. Simon. XML Encryption Syntax and Processing, W3C Recommendation 10th December 2002. `http://www.w3.org/TR/xmlenc-core/`, accessed 2010-08-29.

[33] F. Johnsen, T. Hafsøe, A. Eggen, C. Griwodz, and P. Halvorsen. Web services discovery across heterogeneous military networks. *Communications Magazine, IEEE*, 48(10):84 –90, 2010.

[34] F. T. Johnsen. *Pervasive Web Services Discovery and Invocation in Military Networks*. PhD thesis, University of Oslo, 2010.

[35] F. T. Johnsen, J. Flathagen, T. Gagnes, R. Haakseth, T. Hafsøe, J. Halvorsen, N. A. Nordbotten, and M. Skjegstad. Web Services and Service Discovery. *FFI-rapport 2008/01064*.

[36] F. T. Johnsen, J. Flathagen, T. Hafsøe, M. Skjegstad, and N. Kol. Interoperable Service Discovery: Expermients at Combined Endavor 2009. *FFI-rapport 2009/01934*.

[37] F. T. Johnsen and T. Hafsøe. Service advertisements in MANETs (SAM): A decentralized Web services discovery protocol. In *GLOBECOM Workshops (GC Wkshps), 2010 IEEE*, pages 1674 –1678, 2010.

[38] F. T. Johnsen, T. Hafsøe, and K. Lund. Quality of Service considerations for Network Based Defence. *FFI-rapport 2006/03859*, 2006.

[39] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard), Oct. 2006.

[40] E. Kim and Y. Lee. Quality Model for Web Services. `http://www.oasis-open.org/committees/download.php/15910/WSQM-ver-2.0.do%c`, accessed 2011-02-04.

[41] Ø. Kure and I. Sorteberg. Network Architecture for Network Centric Warfare Operations. *FFI-rapport 2004/01561*.

[42] A. Kuznetsov. iproute2. `http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2`, accessed 2011-02-03.

[43] B. W. Lampson and D. D. Redell. Experience with processes and monitors in mesa. *Commun. ACM*, 23:105–117, February 1980.

[44] K. Lund, A. Eggen, D. Hadzic, T. Hafsøe, and F. Johnsen. Using web services to realize service oriented architecture in military communication networks. *Communications Magazine, IEEE*, 45(10):47 –53, 2007.

[45] K. Lund, E. Skjervold, F. Johnsen, T. Hafsøe, and A. Eggen. Robust web services in heterogeneous military networks. *Communications Magazine, IEEE*, 48(10):78 –83, 2010.

[46] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, R. Metz, and B. A. Hamilton. OASIS. Reference Model for Service Oriented Architecture 1.0, 12th October 2006. `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf`, accessed 2010-08-11.

[47] G. Malkin. RIP Version 2. RFC 2453 (Standard), Nov. 1998. Updated by RFC 4822.

[48] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), Oct. 1996.

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

[50] 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.

[51] N. A. Nordbotten. XML and Web Services Security. *FFI-rapport 2008/00413*, 2008.

[52] R. Ogier and P. Spagnolo. Mobile Ad Hoc Network (MANET) Extension of OSPF Using Connected Dominating Set (CDS) Flooding. RFC 5614 (Experimental), Aug. 2009.

[53] Oracle. Java. `http://www.java.com`, accessed 2011-02-03.

[54] Oracle. Java thread primitive deprecation. `http://download.oracle.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html`, accessed 2011-02-03.

[55] Oracle. Thread (Java Platform SE 6). `http://download.oracle.com/javase/6/docs/api/java/lang/Thread.html`, accessed 2011-02-03.

[56] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental), July 2003.

[57] J. Postel. Transmission Control Protocol. RFC 793 (Standard), Sept. 1981. Updated by RFCs 1122, 3168.

[58] P. Psenak, S. Mirtorabi, A. Roy, L. Nguyen, and P. Pillay-Esnault. Multi-Topology (MT) Routing in OSPF. RFC 4915 (Proposed Standard), June 2007.

[59] Python. 16.2. threading – Higher-level threading interface. `http://docs.python.org/library/threading.html`, accessed 2011-02-03.

[60] Python. Python Programming Language. `http://www.python.org`, accessed 2011-02-03.

[61] S. Rai, B. Mukherjee, and O. Deshpande. IP resilience within an autonomous system: current approaches, challenges, and future directions. *Communications Magazine, IEEE*, 43(10):142 – 149, 2005.

[62] E. Rosen and Y. Rekhter. BGP/MPLS IP Virtual Private Networks (VPNs). RFC 4364 (Proposed Standard), Feb. 2006. Updated by RFCs 4577, 4684, 5462.

[63] R. Russell. The netfilter.org "iptables" project. `http://www.netfilter.org/projects/iptables/`, accessed 2011-02-03.

[64] E. Skjervold, T. Hafsøe, F. T. Johnsen, and K. Lund. Delay and disruption tolerant Web services for heterogeneous networks. In *Military Communications Conference, 2009. MILCOM 2009. IEEE*, pages 1 –8, 2009.

[65] J. Sliwa and M. Amanowicz. A mediation service for Web Services provision in tactical disadvantaged environment. In *Military Communications Conference, 2008. MILCOM 2008. IEEE*, pages 1 –7, 2008.

[66] T. Szigeti and C. Hattingh. *End-to-End QoS Network Design: Quality of Service in LANs, WANs and VPNs*. Cisco Press, 2005.

[67] M. Tian. *QoS integration in Web services with the WS-QoS framework*. PhD thesis, Freie Universität Berlin, 2005.

[68] UNIX. pthread_cancel. `http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread_cancel.html`, accessed 2011-02-03.

[69] A. Vogel, B. Kerherve, G. von Bockmann, and J. Gecsei. Distributed Multimedia and QoS: A Survey. IEEE Multimedia, Vol. 2, No. 2, pp. 10-19, 1995.

[70] Vyatta. The Open Source Networking Community. `http://www.vyatta.org`, accessed 2010-11-19.

[71] J. Wroclawski. The Use of RSVP with IETF Integrated Services. RFC 2210 (Proposed Standard), Sept. 1997.

[72] B. Wu and X. Wu. A QoS-aware Method for Web Services Discovery. *Journal of Geographic Information System*, 2(1):40–44, 2010.

List of Figures

List of Tables

Nomenclature

AODV	Ad hoc On-Demand Distance Vector
CNR	Combat Net Radio
COTS	Commercial off-the-shelf
DiffServ	Differentiated Services
DP	Decision Point
DSproxy	Delay- and Disruption-Tolerant SOAP Proxy
EP	Enforcement Point
HB-UHF	High Band UHF
IETF	Internet Engineering Task Force
IntServ	Integrated Services
ITR	Intelligent Tactical IP Routers
MANET	Mobile Ad-Hoc Network
NATO	North Atlantic Treaty Organization
NNEC	NATO Network Enabled Capabilities
NNEC FS	NNEC Feasibility Study
OASIS	Organization for the Advancement of Structured Information Standards
OLSR	Optimized Link State Routing
OSPF	Open Shortest Path First
OSPF-MT	OSPF Multi Topology
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PHB	Per Hop Behavior

QoA	Quality of Availability
QoP	Quality of Perception
QoS	Quality of Service
RSVP	Resource ReSerVation Protocol
SOA	Service Oriented Architecture
STANAG	NATO standardized agreement
TTL	Time To Live
UHF	Ultra High Frequency
UIDM	Universal Improved Data Modem
URI	Universal Resource Identifier
W3C	World Wide Web Consortium
W3C	World Wide Web Consortium
WSDL	Web Services Description Language
XKMS	XML Key Management Specification
XML	Extensible Markup Language