FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO
Departamento de Engenharia Electrotécnica e de Computadores

QoS Abstraction Layer in 4G Access Networks

Gustavo João Alves Marques Carneiro

Dissertação submetida para satisfação parcial dos
requisitos do grau de mestre
em
Redes e Serviços de Comunicação

Dissertação realizada sob a supervisão do
Professor Doutor Manuel Alberto Pereira Ricardo,
do Departamento de Engenharia Electrotécnica e de Computadores
da Faculdade de Engenharia da Universidade do Porto

Porto, Novembro de 2005

Abstract

In the context of heterogeneous wireless networks, or 4G networks, a number of problems are yet to be solved. One of these problems is the lack of Layer 2 QoS provisioning in heterogeneous networks, mainly due to the non-uniform nature of the QoS models and service interfaces among different wireless technologies. In addition, 4G access networks may include Access Points with no dedicated channel to control their QoS capabilities. An even more demanding scenario involves concatenated Access Points and the need to control and coordinate the QoS provisioning among them. Other problems in need to be addressed include coordination of L3 QoS with L2 QoS and mobility, and L2 multicast with QoS.

In order to help solving these problems, we propose a QoS Abstraction Layer, which is located between layers 2 (link) and 3 (network), in the control plane; it hides from the upper layers the QoS reservation details of the technologies in use, as well as the layer 2 network topology. An abstract QoS service interface is provided to L3 QoS management modules, supporting primitives to reserve or modify L2 QoS resources, represented as abstract QoS parameters. An associated signalling protocol makes the QoS requests be communicated to the relevant QoSAL-enabled L2 nodes; discovery of the correct path to the mobile node in an L2 access network, and translation of abstract QoS parameters into technology-specific QoS parameters are also properties of the QoSAL.

The service interface also features primitives to query available resources in Access Points, as well as notifications when the free resources drop below a certain threshold, which can be used to trigger network initiated handover for purposes of load balancing. Moreover, asynchronous notifications of degradation in existing QoS reservations are also defined, allowing applications to adapt to changing link conditions, among other cross-layer optimisations. Also featured are primitives designed for integration with L3 mobility, and multicast QoS. Finally, the architecture follows a modular design, with technology-dependent aspects separated into their own "driver" modules, allowing simpler development of QoS support for new technologies, without changing any other part of the access network infrastructure.

As proof of concept, a prototype implementation of the QoS Abstraction Layer was developed and tested. The results obtained show that the prototype is correctly implementing the QoS and mobility integration primitives, as well as automatic path discovery to the mobile node.

The resulting work represents a contribution to QoS in packet switched wireless networks, as it is specifically designed to bridge the gap between L3 QoS and L2 QoS. The definition of the abstract QoS interface is by itself an important achievement; also relevant is the way in which the signalling protocol solves the problems associated with the forwarding algorithm of IEEE 802.1D learning bridges.

Resumo

No contexto das redes sem fios heterogéneas, ou redes 4G, encontra-se ainda por resolver um conjunto de problemas. Um destes problemas é a ausência de mecanismos uniformes reservas e pedidos de Qualidade de Serviço (QoS) de nível 2 (L2) em redes heterogéneas; os mecanismos existentes são variados e dependem fortemente das tecnologias rádio em uso. Para além disso, as redes de acesso 4G podem incluir Pontos de Acesso sem canais dedicados ao controlo dos mecanismos de QoS. Um cenário de comunicação mais exigente pode ainda incluir Pontos de Acesso concatenados que devem ser controlados e coordenados relativamente à QoS. Outros problemas a resolver incluem a coordenação de QoS de nível 3 com QoS de nível 2 e mobilidade, e o *multicast* nível 2 com QoS.

Para ajudar a resolver estes problemas, propomos o componente *QoS Abstraction Layer* (QoSAL), que se encontra entre os níveis 2 (ligação) e 3 (rede), no plano de controlo; esta abstracção esconde das camadas superiores os pormenores de reserva de QoS das tecnologias rádio, assim como a topologia da rede de nível 2. É oferecida aos módulos de suporte de QoS de nível 3 uma interface serviço de QoS abstracta, com primitivas para reservar ou modificar recursos de QoS de nível 2, que são representados como parâmetros de QoS abstractos. Um protocolo de sinalização associado permite comunicar os pedidos de QoS aos nós de nível 2 relevantes que suportam QoSAL; a descoberta do caminho correcto até ao terminal numa rede de acesso nível 2 e a tradução entre parâmetros de QoS abstractos e parâmetros de QoS dependentes da tecnologia são outras das características do QoSAL.

A interface de serviço fornece ainda primitivas para se obter informação sobre os recursos livres em Pontos de Acesso, e primitivas de notificação de diminuição de recursos, que podem ser usadas para despoletar o *handover* iniciado pela rede para efeitos de balanceamento de carga. Para além disso, são ainda definidas notificações assíncronas de degradação de QoS das reservas existentes, permitindo que as aplicações se adaptem às flutuações da qualidade da ligação, e que sejam optimizados mecanismos inter-camada. Estão ainda disponíveis primitivas para integrar a QoS com a mobilidade nível 3 e QoS multicast . O QoSAL tem uma arquitectura modular, com os aspectos dependentes de tecnologia localizados em módulos bem identificados, permitindo assim a integração fácil de novas tecnologias.

Para demonstrar o conceito foi desenvolvido e testado um protótipo do QoSAL. Os resultados mostram que o protótipo implementa correctamente as primitivas de QoS e de integração com mobilidade, para além da descoberta automática do caminho para os terminais.

Os resultados deste trabalho representam uma contribuição para a QoS em redes sem fios com comutação de pacotes, pois permitem a integração simples da QoS de nível 3 com a QoS de nível 2, em redes heterogéneas. A definição da interface abstracta de QoS é, por si só, um resultado importante; relevante é também a forma como o protocolo de sinalização resolve os problemas associados ao algoritmo de encaminhamento das *learning bridges* IEEE 802.1D.

Acknowledgements

The author would like thank a number of people for making this work possible. First and foremost, prof. Manuel Ricardo for always believing in me and for invaluable advice and help.

I would also like to thank my colleagues at INESC Porto for their support, in particular Jaime Dias for useful advice and interesting QoS discussions, but also Filipe Sousa, Rui Campos, Ricardo Morla, and Carlos Pinho, for their valuable feedback. The INESC Porto institution in general also deserves my appreciation, for providing me with an excellent environment to complete this work, as well as the University of Porto.

Finally, I would like to thank my family for all the help I received throughout my entire life.

The work presented in this report was partially funded by the EU project IST-2002-506997 “Daidalos” [Dai].

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the brief history of wireless telecommunications, there has been an ever repeating pattern as telecommunications systems are upgraded from one generation to the next. What we see in each upgrade is that the new technology brings great improvements: greater transmission speeds, more quality, less power consumption, and more services. Analysing in detail, it can equally be observed that each technology upgrade brings improvements essentially at the *physical layer*, while the rest of the architecture is more or less reinvented. As a result of this, upgrading from one physical layer to another is usually a difficult and costly process:

1. Most of the infrastructure of operators needs to replaced;

2. Users need to replace their cellphones;

3. Operator or third-party services, currently much tied to the access technology, need also to be adapted or replaced.

The so called "Fourth Generation Wireless Network" (4GW) is currently being researched, and it is expected to replace the current telecommunications systems: 2G (GSM), 2.5G (GPRS), and 3G (UMTS). 4GW will incorporate two disparate communications paradigms — circuit-switched and packet-switched networks — to create a new communications infrastructure sharing properties of both systems, and at the same time adding a few unique properties of its own. These 4GW networks are an attempt to solve the problem previously mentioned by employing a technology-independent layer to insulate most of the operator equipments and services from the lower layers. This technology-independent layer is, in some approaches, the Internet Protocol version 6 (IPv6) [DH98, Sta96, LLM$^+$98]. Some characteristics of 4GW networks are the following:

1. Terminals will be offered a set of services, such as voice and video calls, streaming, and web browsing, which may run on top of IPv6;

2. The particular wireless technology used at any moment will be abstracted. Mobile terminals will be allowed to use multiple wireless technologies to access the same network;

3. Terminals may have multiple wireless cards, and are allowed to perform a "vertical handover". For example, a user that is entering an 802.11 hotspot may switch from the more expensive UMTS access to 802.11, without losing connectivity;

4. Quality of Service (QoS) will be built into the architecture from the ground up, so that the quality that people are used to from circuit-switched calls is preserved in packet-switched networks.

These goals will take wireless communications to a new level of functionality, with advantages for both users and operators. Users are offered flexibility to choose the most convenient access technology available at any time and place (Always Best Connected); operators see the technology-specific part of the network decoupled from the rest of the network, and may offer the same services over a wider range of access methods, with little duplication of network components.

1.1 Reference Scenario

The base scenario addressed in this thesis pushes the boundaries of wireless communications somewhat further. This scenario consists of a mobile node (MN) that uses an Access Router (AR) to obtain connectivity to an operator's network. However, the AR does not offer directly wireless connectivity to the MN. Instead, it is connected (via Ethernet, for example) to one or more Access Points, which ultimately provide connectivity to the MN. This example is made even more complex, but still realistic since we may have concatenated wireless links as shown in Fig. 1.1. IEEE 802.16 (point-to-multipoint broadband wireless access) link (Base Station + Subscriber Station), followed by two APs in parallel, one 802.11 (Wireless LAN) and one 802.15.1 (Bluetooth). The goal is to support a generic L2 network composed of an arbitrary number of equipments interconnected to form a tree rooted at the AR.

Most packet-switched networks offer, by default, only a "best effort" service. This means that there are no guarantees regarding packet delivery, jitter, or delay. Most applications are able to easily adapt to such a network environment, but a few applications, named real-time applications, cannot cope with this lack of guarantees. Usually, support for a certain level of packet transmission guarantees is an add-on service provided by the network, and it is called *Quality of Service* (QoS). Unlike in the Internet, in 4G networks the real time applications will play an important role, thus QoS will certainly become a fundamental requirement in such networks. In order to provide end-to-end QoS to applications, it is first necessary to provide QoS in the wireless part of the network, which is also the most significant bottleneck in end-to-end QoS.

Unlike in fixed/wired networks, which are mainly composed of dedicated point-to-point links interconnecting routers, in wireless networks layer 3 QoS is not in itself

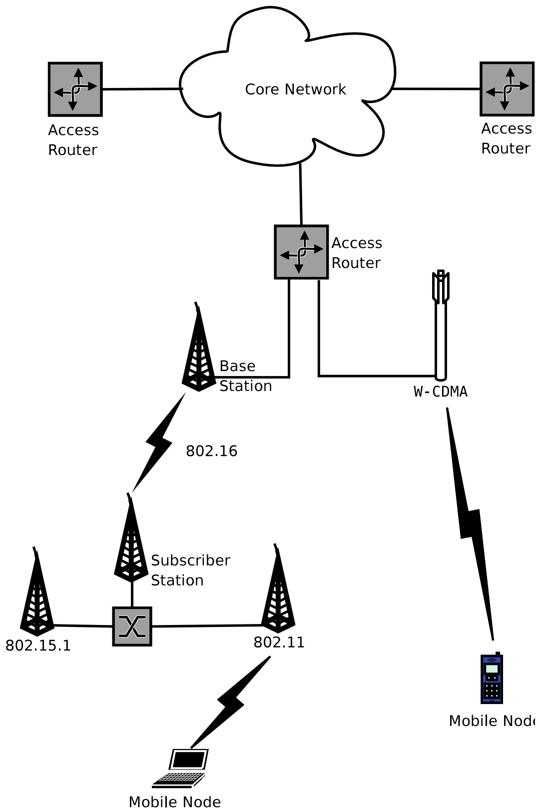


Figure 1.1: Example of multi-hop wireless scenario

sufficient to ensure an acceptable overall level of QoS. Some of the major differences between wired and wireless networks include the much larger transmission delay and bit error rate in wireless networks in comparison to wired ones. Moreover, bandwidth in wireless networks is relatively low and fluctuates very rapidly. Wired networks usually solve the QoS problem by over-provisioning, but wireless networks have much less capacity to begin with, so effort must be made to use efficiently the little bandwidth that is available. In current wireless networks, access to the medium is naturally shared and distributed. To avoid the penalty, in terms of delay and bandwidth, of frame transmission collisions, centralised coordination protocols are sometimes built on top of the shared (at physical level) medium. Delays are still higher than in wired links, but the MAC protocol can be—and often is—tweaked to give different priorities to different terminals or even individual flows. This is what is usually called *layer 2 QoS*. While L2 QoS may not have significant impact on wired networks[1], in wireless networks it can be used to significantly improve the overall QoS level. Therefore, 4G networks will require not only L3 QoS but also L2 QoS.

Most wireless technologies in 4G networks already have built-in QoS support at the MAC layer. The difficult part will be to properly configure and adapt L2 QoS parameters for the end-to-end (L3) QoS requirements, for each flow. This is usually termed *L3/L2 QoS mapping*. QoS mapping would normally be a relatively simple task, but in heterogeneous wireless networks it becomes more complicated since we need to support at the same time multiple L2 technologies, each with a different QoS model and configuration interface. Add to that the uncertainty of the L3 QoS architecture that will be adopted, be it RSVP (with mobility-enhanced variants) NSIS, DiffServ, QoS Brokers, QoS-enabled SIP proxies, some of which may even coexist in the same system, and we come to the conclusion that QoS mapping is indeed a complex task in heterogeneous networks.

In theory, given M distinct L3 QoS models and N L2 QoS interfaces, we should need $M \times N$ different mappings if following the most direct one-to-one mapping approach (Fig. 1.2(a)). A different alternative is to define an abstraction (or convergence) layer, and perform the mapping in two steps: (1) mapping from L3 QoS to the abstraction layer; (2) mapping from the abstraction layer into L2 QoS. In this case (Fig. 1.2(b)) we would need $M + N$ different mappings. Solving the equation

$$M + N < M \times N$$

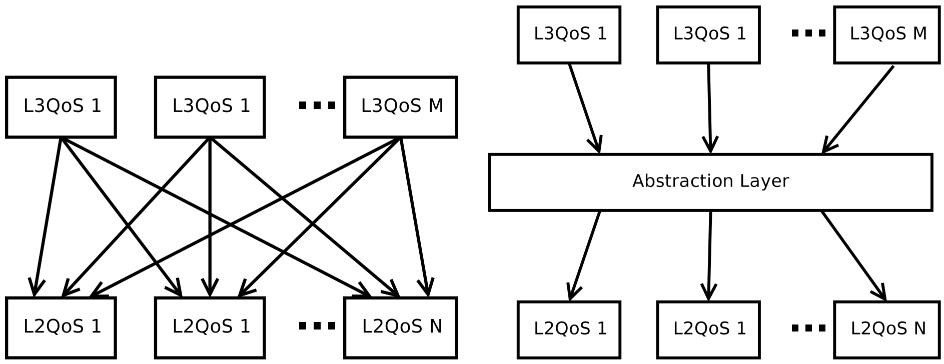
to find out when abstraction layer is simpler to implement than direct mapping, yields

$$\frac{M \times (N-1)}{N} > 1$$

Let us suppose we have two L3 QoS models to consider: DiffServ and IntServ. Replacing $M = 2$ we obtain

$$N > 2$$

[1]Moreover, in wired point-to-point links it has absolutely no impact



(a) Direct mapping

(b) Mapping through an abstraction layer

Figure 1.2: Analysis of the complexity of L3/L2 QoS mapping

Considering a 4G scenario, this will likely be true; for instance, UMTS, 802.11 and 802.16.

Even if the above was not true, defining an abstraction layer is important for additional reasons, such as: (1) to discover common ground of all L2 QoS interfaces; (2) to define—and at the same time limit and simplify—the functionality L3 QoS modules will need from L2; 3) to allow future L2 technologies to be cleanly included into the architecture without significantly affecting L3 modules.

For these reasons, a component called “QoS Abstraction Layer” (QoSAL) has been developed and is described in this thesis; it is a generic protocol for QoS reservation in L2 wireless networks that hides from the hosts and access routers the topology and heterogeneity of the underlying L2 network.

1.2 Objectives

The objectives defined for this work are the following:

- Specify a service interface (primitives and parameters) for the QoS abstraction layer which considers:
  - The QoS interfaces of the most important L2 wireless technologies likely to be used in 4G networks;
  - The requirements of the L3 QoS architectures expected to be used in 4G networks;
- Define a protocol for reserving QoS across an L2 network that is composed of several L2 elements between a single pair of L3 nodes, and which requires that QoS is reserved separately in each segment connecting them;

- Create a working prototype as proof-of-concept, and use it to perform tests.

1.3 Original Contributions

The following contributions have been produced during the course of this work:

1. A model for L2 QoS reservations, including primitives and abstract QoS parameters, which can be easily mapped into the most common wireless networks in use, such as UMTS, 802.11, Bluetooth, and 802.16. This QoS model is simple and allows mapping into future L2 technologies to be easily implemented.

2. An associated signalling protocol which allows the L2 QoS reservations to be conveyed into remote access points, and which transparently supports concatenated IEEE 802 networks (eg. 802.16 followed by 802.11) robust and efficiently.

1.4 Result

The work performed around this thesis has led to the development of a prototype. This prototype implements the abstract L2 QoS interface, and, among other capabilities, is able to communicate with the layers above it in order to negotiate QoS reservations, among other capabilities. In addition, the prototype implements the QoSAL signalling protocol to allow extending abstract QoS reservations to remotely located Access Points.

1.5 Organisation of the Thesis

The remainder of this thesis is organised as follows. Chapter 2 provides some background on the most important concepts and technologies behind QoS in packet-switched networks. Chapter 3 explains the QoS architecture of the IST DAIDALOS project, which is important to understand the context of this work. Then, Chapter 4 describes the QoS Abstraction Layer, including architecture, interfaces, and protocol. This description is complemented in Chapter 5 by a description of the prototype implementation and respective tests, used to validate the concept. Finally, Chapter 6 concludes with some final remarks about the results and possible directions for future work.

Chapter 2

QoS in Packet-switched Networks

2.1 Introduction

Packet-switched networks are usually based on a “best-effort” service model. This service model consists on the ability of transmitting information in the form of packets, but with an implicit assumption that there are no absolute guarantees on how much time transmission will take, or even whether it will succeed. In addition, the order of packets is not guaranteed to be preserved from sender to receiver. To work around these problems, some upper layer protocols, such as TCP, implement their own packet retransmission and reordering schemes, thus offering a reliable service on top of a unreliable one.

One of the things that TCP cannot assure, however, is limited and low delay. Moreover, real-time applications, such as voice and video conferencing, often use RTP instead of TCP, since TCP favours reliability in detriment of delay; it sometimes retransmits packets that are lost even though the retransmitted packet would arrive too late at the receiver to make any difference. RTP has much less overhead than TCP, but in case of real-time applications they both fail to provide an adequate service on top of a best-effort network.

In packet switched networks there exist some architectures that are able to provide some limited guarantees to selected packet flows. This is normally called *Quality of Service* (QoS). The current chapter will present an overview of the most significant QoS architectures that have been developed, focusing on Layer 3 QoS (Sec. 2.4) systems, then Layer 2 technologies providing QoS (Sec. 2.5), and finally some of the most relevant QoS systems currently under research (Sec. 2.7). But first, Sec. 2.2 contains a quick introduction to multicast, and Sec. 2.3 introduces some general QoS concepts that are important to understand the remaining sections.

2.2 Multicast

A recurring problem in packet switched networks happens when a sender host needs to send the same information to multiple receivers. The straightforward solution consists in sending the same packet multiple times with different destination addresses, once for each receiver. It is easy to see that, in most cases, the route from a sender to multiple receivers has many common links. In such links, basically the same information is being transmitted a certain number of times, with only the destination address changing. This is not scalable to a large number of receivers, and it wastes a lot of bandwidth. A different approach that could be taken would consist on transmitting a single copy of each packet, but with a *broadcast* address. This avoids the duplication of packets; on the other hand, the information is replicated *everywhere*, even to parts of the network that do not have any receivers. This approach is impractical in the Internet in general, although it is more feasible in a small local area network.

A common solution for this problem is to use *multicast*. Multicast is a service offered by some packet-switched networks that consists in allowing a sender application to send packets to multiple receivers in a scalable and efficient way. From the point of view of the sender, a list of receivers is represented simply as a *multicast address*, sometimes also called *multicast group*. The multicast address is used as destination address for packets that are to be delivered to a list of receivers. It is then the job of the network to deliver a packet to all nodes interested in receiving it. Usually, the intermediate nodes find out about interested receivers for any give multicast group from the receivers themselves, through appropriate multicast and group management protocols.

Multicast exists both at layer 3 and layer 2. For instance, in IPv6 there exists a block of addresses [HD03] allocated for multicast, and the Multicast Listener Discovery [VCL04] protocol is used by routers and receiver nodes to build a “multicast tree” representation, so that routers are able to forward multicast packets only to the neighbouring nodes that are interested in receiving them. At layer 2, there is also multicast. In IEEE 802 networks, a block of MAC addresses (starting with 33:33:...) is allocated for multicast. IPv6 multicast addresses are easily mapped [Cra98] into IEEE 802 MAC multicast addresses. The IEEE 802.1D [Tel98] GARP Multicast Registration Protocol (GMRP) can be used to manage multicast trees in IEEE 802.1D bridges, i.e. switches or access points. Even UMTS has now support for multicast, as shown in Sec. 2.5.2.

2.3 QoS Concepts

2.3.1 Shaping, and Policing

In most QoS architectures, two of the most common operations to be performed on packet flows are 1) shaping and 2) policing.

Shaping can be defined as an operation performed on an arrival flow in order to transform it into a departing flow with the same packets but changing the shape of the departure curve relatively to the arrival one. The objective is to obtain departure curve that follows certain predefined parameters. Usually, shaping a flow is used to smooth the flow by eliminating or limiting bursts in some way. A token bucket algorithm, as described in Sec. 2.3.2 is often used to accomplish this. The main benefits of shaping flows are to generally decrease jitter in networks and allow networking equipments to have smaller buffers.

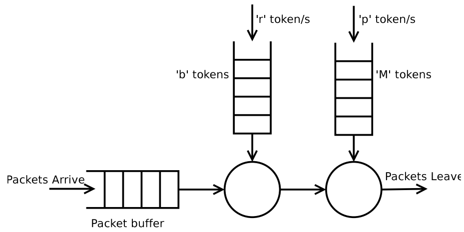


Figure 2.1: The token bucket algorithm

Policing is very much like shaping, except that the arriving flow is merely parameterised, and packets that exceed parameter limits, called *non-conforming packets*, are simply dropped or moved to a best effort queue. This operation is used when there is some agreement, in terms of flow parameter limits, between one entity that generates a flow and another entity that processes it, and the second entity has to prevent the sender from violating that agreement.

2.3.2 The Token Bucket Algorithm

An important concept in QoS is that of the *token bucket filter*. This algorithm, which is based on the earlier *leaky bucket* design [PG93], can be represented as the block diagram in Fig. 2.1. Conceptually, it consists of two "buckets" that receive "tokens" arriving at predetermined rates. Let the rate at which tokens arrive in the first and second buckets be denoted r and p respectively. Now consider that the buckets have a limited depth and cannot hold more than b and M tokens, respectively. Now, as packets enter the system, their size is measured in terms of tokens. For each packet that arrives, if the number of tokens in each bucket is greater or equal than the size of the packet, then packet goes directly to the output of the system and at the same time consumes a number of tokens from each bucket equal to its size. However, if this condition is not met then the packet is either placed in the input queue, waiting for enough tokens to arrive at the buckets or, if there is no input queue, is simply dropped. The four token bucket parameters, r, b, p, M, as well as an additional parameter m that sometimes is also used, have the following meaning:

r is the *mean data rate*, in token/s (one token is usually one byte or one bit). It represents the maximum allowed sustained rate that the token bucket allows;

b is called *maximum burst size*, and is used to limit the size/length of bursts that may leave the token bucket;

p is the *peak data rate*, which is the maximum rate of any bursts that may leave the token bucket;

M is the maximum packet size; since the second bucket cannot hold more than M tokens, we are effectively limiting the size of packets that may leave the system to M tokens;

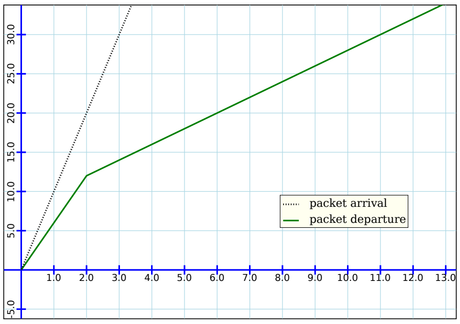


Figure 2.2: Arrival and departure curves of an input flow with an arrival rate much higher than the token bucket r parameter

m is the *minimum policed unit* parameter, which sometimes is also present, and it means that the token bucket algorithm must consider packets smaller than m tokens as being of size m.

To summarise all this, we can say that a token bucket can be used to limit the rate of traffic. In steady-state, it imposes a certain limit r on the output rate of packets, but is flexible to allow occasional bursts, but limited both in maximum rate (p) and maximum duration (b), while at the same time imposing a hard limit on packet sizes (M). As an example, consider the arrival and departure curves (accumulated byte counts) in Fig. 2.2 of a token bucket subjected to an arriving flow with much higher rate than the mean data rate that was configured. It contains *arrival/departure curves*, which are representations of the number of bytes received or transmitted over time. In this kind of plot, the inclination of a segment gives an indication of traffic rate: more inclination means higher packet rate. We can also observe the delay observed by packets in the system by projecting an imaginary horizontal line over the arrival and departure curves, and comparing the time values of the two intersections. In this case we can see that, while initially the token bucket limits the incoming flow to a rate p, when the first bucket has excess tokens, at some point in time the tokens in the first bucket (limited by b) are exhausted, and from there on the outgoing packet flow rate is limited to r.

The token bucket algorithm is very useful for QoS in many contexts. Not only does it allow to perform both shaping and policing by controlling the existence of the input buffer (without buffer it does policing, with it does shaping) but, most importantly, it is used to describe the shape of traffic that is to be expected—or is acceptable—to many different systems, including IntServ, UMTS, and DiffServ.

2.4 Layer 3 QoS

The provision of QoS can occur at either Layer 3 (IP) or Layer 2 (link technology). In case of L3 QoS, two main concerns have to be addressed:

1. How to coordinate the actions of all L3 nodes (routers) between the sender and

receiver hosts in order to meet the end-to-end QoS requirements. Usually some kind of signalling has to take place for this to happen;

2. Each individual host in the path must employ some kind of mechanism to ensure that, for each flow with QoS reservation (either explicit or implicit), the QoS parameters are obeyed. In a pure L3 QoS system, the link technology (L2) has a single Service Access Point (SAP) for packet transmission, thus all packets are given the same treatment. Therefore, L3 QoS usually has a single queue leading up to the network interface, containing packets from all different flows, which are then reordered so that QoS for all flows is met. For instance, packets from real-time flows will likely jump ahead of packets from best-effort applications in the transmission queue.

Three of the most important L3 QoS systems to date—IntServ, DiffServ, and MPLS, are briefly summarised in the following sections.

2.4.1 IntServ

The Integrated Services framework, along with the associated RSVP protocol, is extensively defined by IETF in RFCs 2205–2216. It describes a QoS architecture wherein both sender and receiver applications communicate to the network elements their QoS requirements for each distinct packet flow. The network elements in turn try to ensure the requested QoS is observed throughout the network during the duration of the reservation. The IntServ framework contemplates multiple service levels, but only two standard services types have been defined:

Controlled Load Defined in [Wro97], this is the worst service that is available to IntServ applications. When reserving a Controlled Load (CL) QoS service, applications indicate to the network elements an approximate "envelope" for the traffic that is going to be generated. This envelope is described in terms of TBF parameters. The service that is then received by applications closely resembles the best-effort service in a lightly loaded network. This means that most packets will get through with moderately low delay, but there are no strict guarantees. The CL service is implemented by network elements by a simple admission control scheme, i.e. no scheduling is performed on packets. This is actually both the greatest weakness and greatest feature of this service; it does not provide a lot of guarantees, but what it provides is good enough for most applications and, most importantly, is relatively simple to implement;

Guaranteed The Guaranteed[SPG97] service allows applications to request to the network that an upper bound be placed on the total queueing delay along the end-to-end path for any particular flow. The sender application first informs the network elements of the characteristics (once more, as token bucket parameters) of the flow that is to be transported; then, the receiver requests that a certain bandwidth R be reserved for this flow, as well as guaranteeing that the buffers in routers are correctly dimensioned to ensure that there will be no packet losses due to queueing overflow. It is possible to compute an upper bound for the end-to-end delay given the token parameters that describe the flow and the reserved bandwidth R. The parameter R must be always greater or equal than the mean data rate of the flow r, otherwise the delay will have no guaranteed bound, and some packets will be lost. If the receiver is flexible to accept a little additional delay, it can include a non-zero *slack term* S to improve the probability that the reservation will be accepted. If, on the other hand, the delay bound computed for $R = r$ is not satisfactory, it is possible to request a higher R value, this way obtaining a small reduction in the end-to-end delay bound.

The recommended protocol for IntServ applications to communicate with network elements is the Resource ReSerVation Protocol, RSVP[BZB$^+$97]. RSVP signalling can be briefly summarised this way:

1. The *sender* application starts by sending a *PATH* message to the receiver. This message contains information that describes the shape of the flow that is to be transmitted in terms of *Token Bucket Flow* (TBF) parameters;

2. When the *receiver* application sees this message, it sends a *RESV* message back to the sender, this way informing all the routers in the path towards the sender of the QoS it wishes to receive;

3. If the receiver requested confirmation in the RESV message, a *ResvConf* is sent back from sender to receiver to confirm that the reservation succeeded;

In RSVP, there is a lot of flexibility in the way that packets can be identified as belonging to the same reservation. The packets are always filtered on the destination values *DestinationAddress*, *ProtocolID*, and *DestinationPort*, where DestinationAddress can be either an IPv4 or IPv6 address, including multicast addresses. On the part of the sender, several options are available. On one hand, there can be either *wildcard* or *explicit* sender selection. A wildcard selection means to match packets from any sender as belonging to the same reservation, while an explicit selection requires that a list of sender address/port pairs be stated in the reservation. On the other hand, a reservation can request resources that are to be shared for packets from all senders, or it can request individual reservations for each sender. The most common filter used is called Fixed Filter, which selects a single sender.

This whole complexity regarding sender selection is partly related to the multicast-centric IntServ/RSVP designed. In fact, the good multicast support is still regarded in these days as one of the best features in IntServ/RSVP. Unfortunately, it also has a few design problems of its own, such as:

Mobility Although RSVP foresees the need to perform what is called “local repair” in response to routing table changes, the update of the reservations takes a certain time in the order of a few seconds. This time scale is inadequate for preserving QoS during handoff. There are many other problems with Mobile IP and RSVP, as described in [LC03].

Link layer indications RSVP is not prepared to receive indications from layer 2 regarding changes in available bandwidth, among other things. The effect of this design flaw is only now being felt, given the recent the proliferation of wireless networking; when RSVP was initially designed wireless networking was little more than a concept, thus RSVP was never designed for wireless links that have to change available bandwidth very fast as terminals move away or towards access points, for instance;

Scalability RSVP is not scalable; given that it is soft-state (thus requires periodic refresh) and it requires per-flow signalling, as we move more towards high-performance core networks it becomes clear that the routers that often have to process millions of packets cannot afford to waste time processing RSVP signalling messages and implement packet classification and scheduling for all RSVP flows. Of course this is only really a problem in *core* networks; it is perfectly feasible to have RSVP-aware routers only in the access part of the network, and map RSVP flows into another QoS system such as MPLS or DiffServ at the core network boundaries.

2.4.2 DiffServ

The Differentiated Services (DiffServ) framework, as defined in [BBC$^+$98], was created within IETF partly to solve the scalability problems of RSVP/IntServ, and actually takes the opposite approach by considering that QoS is to be provided only to *aggregate* flows. Contrary to the RSVP/IntServ approach of performing QoS reservations per flow and on demand, the DiffServ framework is based on a contract between a network operator and a client, prior to any flows being created. The DiffServ part of such contract is called Service Level Specification (SLS), which includes a set of parameters that describe the service level that the network operator pledges to enforce. Also part of the SLS is a Traffic Conditioning Specification (TCS), containing parameters that describe how packets are to be classified and shaped/policed before entering the DiffServ domain.

When entering a DiffServ domain, packets are classified and aggregated as specified by the SLS, then (re-)marked with a Diffserv Codepoint (DSCP) value that uniquely identifies to which aggregate flow each packet belongs. The advantages of this procedure are evident:

- There exists a limited and always low number of different DSCPs; consequently, DiffServ-enabled routers can easily implement the necessary queues and scheduling;
- No signalling is necessary inside the DiffServ domain; the QoS parameters to apply for each DSCP are statically pre-configured.

For these reasons, DiffServ is much more easily deployed as a service to be offered by network providers. On the other hand, the level of guarantees that can be provided by a DiffServ network is not always satisfactory. Since DiffServ QoS is applied only to aggregated flows, it is possible for individual microflows to be temporarily degraded, in spite of the aggregated flow as a whole being treated fairly. Also, DiffServ contracts are rather static and cannot be easily dynamically updated as the need arises for the user to launch more QoS-demanding application sessions; although, to be fair, in the core network all users' flows are aggregated and, statistically, the QoS requirements for each DSCP change very slowly. For these reasons, DiffServ can be a good solution for core networks, which have huge amounts of bandwidth available anyway, but on access networks per flow signalling protocols like RSVP or NSIS[HKL04] are often preferable. It is also possible to merge the two approaches, for example by using end-to-end RSVP in the access networks and a DiffServ core network [BFY+00].

2.4.3 MPLS

The Multiprotocol Label Switching Architecture [RVC01], is based on the core concept that great performance gains can be achieved if, when switching a large number of similar packets (eg. DiffServ aggregate flows), a path is first established and recorded across a network, and then all subsequent packets just follow the recorded path. In MPLS terminology, the set of packets that follow the same path is called *Forwarding Equivalence Class* (FEC), and the path itself named *Label Switched Path* (LSP). The name for the LSPs comes from the fact that, once a path has been established, packets entering an MPLS domain are classified into FECs and then they are assigned a label based on the corresponding FEC. Inside the MPLS domain, packets are transmitted with a small MPLS header, named *shim header*, that contains the assigned label and little more information. The MPLS labels always have local meaning for each node/port pair, thus packets' labels have to be swapped as they travel through the nodes of an LSP. The label switching algorithm can be summarised like this:

1. A packet arrives on an input port;
2. **if** packet contains an MPLS shim header **then:**
   (a) Look at the $(label_{input}, port_{input})$ pair, use a lookup table to map into a FEC;
   (b) Use another table lookup to determine the $(label_{output}, port_{output})$ pair from the FEC;
   (c) Swap $label_{input}$ for $label_{output}$ in the packet shim header;
   (d) Queue de packet for transmission in $port_{output}$.
3. **else:**
   (a) Use traditional IP-based routing.

Considering that the MPLS label is just a simple 20-bit integer value, it's easy to see that label switching is more efficient than L3 switching. But more importantly, MPLS allows *traffic engineering* to be deployed on high performance core networks with little or no performance penalty. This is due to the fact that complex matching rules for determining the FEC for packets may be placed on lightly loaded edge routers (the so called *Label Edge Routers*), while at the MPLS core the paths are already established and switching is based on MPLS labels as usual, meaning that the cost of switching best-effort and traffic engineered packets/flows is practically the same. This is an extremely important property for implementing QoS in high speed core networks, considering that one of the greatest hurdles for deploying QoS in these conditions is that introducing packet classification into high performance core routers significantly degrades their performance. With MPLS, both IP forwarding and QoS classification processes are simplified, at least in the user plane,

The MPLS core specification does not specify how LSPs are to be created. That is intentional, and is to allow multiple LSP signalling protocols to be developed independently. Examples of such protocols are the *Label Distribution Protocol* [ADF+01], and RSVP-TE [ABG+01].

2.5 Layer 2 QoS

The main difference between L2 and L3 QoS techniques is that L2 QoS takes advantage of the characteristics of each L2 technology, such as MAC protocol, in order to provide at least some limited QoS guarantees to some datagrams in detriment of others with no such requirements.

A quick overview of some L2 technologies that provide some form of QoS is presented below. This list is by no means complete, however.

2.5.1 IEEE 802.1D

Although lacking any significant deployment in current networking equipment, the IEEE standard 802.1D [Tel98] defines a user_priority value[1] that may optionally be associated with individual frames. The user_priority value for each frame may be obtained by a number of different ways: (1) based on the port a frame arrives on, which is administratively configured, (2) from an optional 802.1Q header, (3) from the basic frame header, if the technology supports it[2]. Table 2.1 lists the user_priority values that have been defined for use in IEEE 802 networks. In IEEE 802 equipments that support it, there is on each output port a separate queue for each possible user_priority value. Frames arriving at the equipment are forwarded to the corresponding queue in the correct output port. The scheduling between queues is a simple priority system, where queues with higher priority[3] always take precedence over queues with lower priority. Naturally, this may lead to starvation problems if, for instance, a station starts flooding the network with “Voice” and “Video” packets at a very high rate, thus completely preventing any Best Effort traffic to be transferred. That is at least the basic algorithm specified in

[1] This was previously specified in a different standard document, 802.1p, and was later merged into 802.1D

[2] For instance, 802.3 and 802.11 do not support it, while 802.5 (Token Ring) does indeed have this field in the basic frame header.

[3] But note that user_priority 1 (Background) actually has less priority than 0 (Best Effort)

the standard, which however leaves the possibility open for administratively selecting different scheduling algorithms that the equipment may support.

Table 2.1: List of IEEE 802.1D user_priority values

user_priority	**Acronym**	**Traffic type**
1	BK	Background
2	–	Spare
0 (Default)	BE	Best Effort
3	EE	Excellent Effort
4	CL	Controlled Load
5	VI	"Video," $<$ 100 ms latency and jitter
6	VO	"Voice," $<$ 10 ms latency and jitter
7	NC	Network Control

2.5.2 UMTS

The *Universal Mobile Terrestrial Service* (UMTS) is an evolution of GSM and GPRS systems that offers mobile connectivity with available bandwidth an order of magnitude higher than the previous (2G and 2.5G) systems that it replaces. In Fig. 2.3, TE and MT are two subsystems of the same Mobile Node. The Radio Access Network (RAN) subsystem includes Node-Bs ("base stations") and Radio Network Controllers (RNCs). The Core Network Edge Node is actually the Serving GPRS Support Node (SGSN), which handles mobility management, authentication, and authorisation. The CN Gateway, called Gateway GPRS Support Node (GGSN), is a router that provides interconnection between the UMTS packet switched domain and external IPv4/6 networks, while at the same time supporting charging functions for all connections. The GGSN is actually the first real IP router from the point of view of the mobile terminals; all other equipments before only provide L2 bridging capabilities.

The UMTS architecture provides QoS functions at essentially two layers: 1. at the UMTS Bearer Service, and 2. at the Radio Access Bearer Service.

At UMTS Bearer Service layer, the QoS interfaces exposed consist of the ability of creating L2 tunnels between the mobile terminal and the GGSN. These tunnels, called PDP Contexts, are identified by unique numbers called Network Service Access Point Identifiers (NSAPIs), and can have a number of QoS attributes attached to them, which are summarised in Table 2.4. As show in Table 2.2, at the terminal side this layer provides primitives for creating, modifying and releasing primary, secondary and MBMS (Multimedia Broadcast/Multicast Service) PDP Contexts. The main difference between primary and secondary PDP Contexts is that there can be only one primary PDP Context for each IP address assigned to the terminal, while multiple secondary PDP contexts can coexist, often with different QoS, attached to the same primary PDP Context, thus sharing the same IP address. Usually a terminal has a single primary PDP

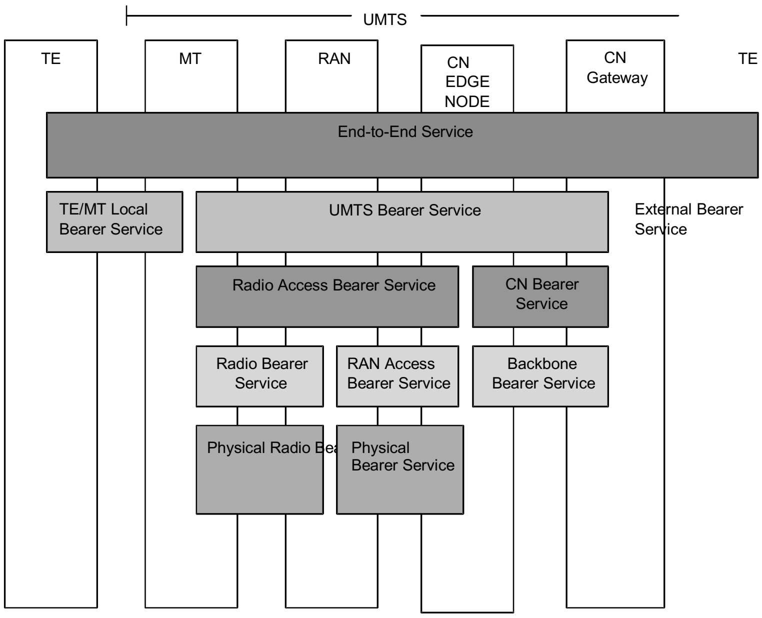


Figure 2.3: UMTS Rel-6 architecture (source: 3GPP 23107-630 specification document)

Context, thus a single IP address, for signalling IP traffic, and multiple secondary PDP Contexts, each with different QoS settings, for transporting application flows. Also MBMS [3GP05, Iva05] sessions are supported using this PDP Context concept. At the network side, this UMTS Bearer Service also provides some primitives (Table 2.3), although only limited functionality is available from this side. In fact, the GGSN is not allowed to activate PDP Contexts with QoS, and anything the GGSN may request, such as PDP Context activation, modification, and deactivation, has to be authorised by the terminal. This is a deliberate design decision that takes into account the fact that the terminal is the one that is paying for the services, thus should be in full control of the resources that it is paying for.

At the Radio Access Bearer layer, on the other hand, the primitives are more network oriented. As we can see in Table 2.5, it is the network that has to take initiative of establishing new RABs, in contrast with the Bearer Service layer, where PDP Contexts are always created by the terminal. It should be noted, however, that since PDP Contexts are mapped into RABs, the terminal is still in control of the radio resources, albeit indirectly through SGSN authorisation.

2.5.3 Bluetooth

The Bluetooth [MW05] wireless technology was first defined by a consortium of several equipment manufacturers led by Ericsson called *the Bluetooth SIG*, and later adopted as standard (802.15.1) by IEEE. The main focus of this technology is to allow small and cheap devices to become wireless interconnected. This is a deliberate design decision with

Table 2.2: List of UMTS Rel-6 SMGREG (terminal side) primitives

Primitive	**Parameters**	**Purpose**
SMREG-PDP-ACTIVATE-REQ	PDP address, QoS, NSAPI, APN, Protocol configuration options	Requests activation of a (primary) PDP Context
SMREG-PDP-ACTIVATE-CNF	PDP address, QoS, NSAPI, Protocol configuration options	Confirmation from network that the request PDP activation succeeded
SMREG-PDP-ACTIVATE-REJ	Cause, NSAPI, Protocol configuration options	Rejection of (primary) PDP Context activation
SMREG-PDP-ACTIVATE-IND	PDP address, APN, protocol configuration options	Request from network suggesting activation of a primary PDP Context
SMREG-PDP-ACTIVATE-REJ-RSP	Cause, PDP address, APN, protocol configuration options, MBMS protocol configuration options	The network request PDP Context activation failed
SMREG-PDP-DEACTIVATE-REQ	NSAPI(s) tear down indicator, cause, protocol configuration options, MBMS protocol configuration options	Request deactivation of a PDP Context
SMREG-PDP-DEACTIVATE-CNF	NSAPI(s), protocol configuration options, MBMS protocol configuration options	Confirmation from network that the request PDP deactivation is complete
SMREG-PDP-DEACTIVATE-IND	NSAPI(s) (s), tear down indicator, cause, protocol configuration options, MBMS protocol configuration options	A PDP was deactivated from the network side
SMREG-PDP-MODIFY-IND	QoS, NSAPI, protocol configuration options	Network is requesting modification (eg. QoS parameters) of a PDP context
SMREG-PDP-MODIFY-REQ	QoS, NSAPI, TFT, protocol configuration options	Request modification (eg. QoS parameters) of a PDP Context
SMREG-PDP-MODIFY-CNF	QoS, NSAPI, protocol configuration options	A PDP modification was successfully concluded
SMREG-PDP-MODIFY-REJ	Cause, NSAPI, protocol configuration options	A PDP modification was rejected
SMREG-PDP-ACTIVATE-SEC-REQ	QoS, NSAPI, TFT, Primary NSAPI, protocol configuration options	Request activation of a "secondary" PDP Context
SMREG-PDP-ACTIVATE-SEC-CNF	QoS, NSAPI, protocol configuration options	Activation of a secondary PDP context has been concluded successfully
SMREG-PDP-ACTIVATE-SEC-REJ	Cause, NSAPI, protocol configuration options	Activation of a secondary PDP context was rejected
SMREG-MBMS-ACTIVATE-REQ	Multicast address, supported MBMS bearer capabilities, NSAPI, APN, MBMS protocol configuration options	Request activation of a MBMS PDP Context
SMREG-MBMS-ACTIVATE-CNF	Multicast address, NSAPI, MBMS protocol configuration options	A MBMS PDP Context activation has been completed successfully
SMREG-MBMS-ACTIVATE-REJ	Cause, NSAPI, MBMS protocol configuration options	A MBMS PDP Context activation has been rejected
SMREG-MBMS-ACTIVATE-IND	Multicast address, APN, MBMS protocol configuration options	Network is requesting activation of an MBMS PDP Context

Table 2.3: List of UMTS Rel-6 SMGREG (network side) primitives

Primitive	**Parameters**	**Purpose**
SMREG-PDP-ACTIVATE-REQ	PDP address, APN, protocol configuration options	The nework initiates a PDP context activation; the terminal receives a SMREG-PDP-ACTIVATE-IND and is expected to proceed with a normal activation
SMREG-PDP-ACTIVATE-REJ	Cause, PDP address, APN, protocol configuration options	The network initiated PDP context activation failed
SMREG-PDP-DEACTIVATE-REQ	NSAPI(s), teardown indicator, cause, protocol configuration options, MBMS protocol configuration options	The network request the terminal to deactivate a PDP context
SMREG-PDP-DEACTIVATE-CNF	NSAPI(s), protocol configuration options, MBMS protocol configuration options	A network initiated PDP context deactivation has succeeded
SMREG-PDP-MODIFY-REQ	QoS, NSAPI, protocol configuration options	The network requests modification of an existing PDP context
SMREG PDP-MODIFY-CNF	NSAPI, protocol configuration options	A network initiated PDP context modification was concluded
SMREG PDP-MODIFY-REJ	NSAPI, protocol configuration options	A network initiated PDP context modification was rejected
SMREG-MBMS-ACTIVATE-REQ	Multicast address, APN, MBMS protocol configuration options	The network requests activation of an MBMS PDP context
SMREG-MBMS-ACTIVATE-REJ	Cause, multicast address, APN, MBMS protocol configuration options	A network initiated MBMS PDP context activation has failed

Table 2.4: List of UMTS Rel-6 PDP Context QoS Attributes

Attribute	**Value(s)**	**Description**
Traffic class	Conversational, Streaming, Interactive, or Background	An indication of the type of traffic that the PDP Context is expected to carry; the main differences between these classes is in terms of delay requirements
Maximum bitrate	(kbps)	Maximum bitrate that is allowed for burst traffic; this is analogous to the p parameter in the token bucket algorithm in Sec. 2.3.2
Guaranteed bitrate	(kbps)	Sustained bitrate that the network will allow; this is analogous to the r parameter in the token bucket algorithm in Sec. 2.3.2 and with a b parameter equal to the Maximum SDU size PDP QoS attribute
Delivery order	(y/n)	Whether in-sequence SDU delivery is required or not
Maximum SDU size	(octets)	Maximum datagram size that can be transported (= MTU)
SDU format information	(bits)	List of possible SDU sizes; this information can be provided to the network in order allow some optimisations
SDU error ratio	(float)	Maximum tolerable fraction of lost SDUs that the application can tolerate
Residual bit error ratio	(float)	Maximum tolerable fraction of erroneous bits per SDUs that the application can tolerate
Delivery of erroneous SDUs	(y/n/-)	Whether to deliver SDUs that contain erroneous bits; some applications, such as video-telephony, are able to tolerate packets containing a few erroneous bits; the alternative of discarding entire packets with only a few erroneous bits is, in this case, much more detrimental to the overall quality
Transfer delay	(ms)	Maximum tolerable delay
Traffic handling priority	(integer)	As an alternatively to specifying the bitrate attributes above, applications can just request traffic relative priorities
Allocation / Retention Priority	(integer)	When the access network becomes too loaded, it may be required to stop accepting some new PDP Contexts or to even discard existing ones; this value controls the relative priority of allocation and retention for PDP Contexts when this happens
Source statistics descriptor	('speech' / 'unknown')	Since speech has predictable traffic patterns, by indicating to the network that a PDP Context will be used for voice streams the network is able to better calculate the statistical multiplex gain, thus use resources more efficiently
Signalling Indication	(Yes/No)	Indicates if the SDUs to be transported are of signalling nature. This attribute is defined only for the Interactive traffic class.

Table 2.5: Overview of UMTS Rel-6 Radio Access Bearer (network side, SGSN-to-RNC) primitives

Primitive	**Parameters**	**Purpose**
RAB ASSIGNMENT REQUEST	RAB ID, RAB parameters (eg. QoS), User Plane Information, Transport Layer Information, etc.	Used to establish, modify, or release one or several RABs
RAB ASSIGNMENT RESPONSE	List of RABs successfully established or modified	Result of RAB ASSIGNMENT REQUEST
RAB RELEASE REQUEST	RAB IDs, Cause	Sent by the RNC to request the release of a RAB

direct impact on the characteristics of the wireless technology. In fact, due to the desire to make Bluetooth devices cheap and consuming little power, the available bandwidth and range are rather limited when compared to other wireless technologies, such as IEEE 802.11. Current Bluetooth (v1.2) devices support ranges from 10 to 100 meters and power consumption from 1 to 100 mW, while offering bandwidths of no more than 723.1 kbit/s, which is adequately matched to the use cases Bluetooth was designed for.

At the radio level, Bluetooth devices communicate in time slots (each of 625 μs), some assigned for downlink others for uplink transmission (Time Division Multiplexing) and Frequency Hopping (FH) (the carrier frequency changes in each time slot). Up to 8 Bluetooth devices may directly communicate, by forming a *piconet*. In a piconet, one device is elected to assume the role of *master*, while the remaining devices become *slaves*. The master controls essentially two aspects of the communication: (1) its address determines the FH sequence in its piconet; (2) slaves are only allowed to send packets following a reception of a packet from the master addressed to them (a polling scheme). Packets in Bluetooth can occupy 1, 3, or 5 time slots, so that the master always transmits in even time slots and slaves can only transmit starting on odd time slots.

Above the physical layer, Bluetooth defines *logical channels*. On one hand, there are channels for raw audio communication, focused for mobile wireless handset applications, for example. These are called Synchronous Connection-Oriented (SCO), since they provided a dedicated circuit-like channel with fixed bitrate. For digital data communications, Bluetooth also provides Asynchronous Connection-oriented Logical channels, ACL. ACL channels have a simple ARQ (stop-and-wait) system to provide the additional reliability that data communications require. There are 6 packet types that can be used in ACL channels: DM1, DM3, DM5, DH1, DH3, and DH5. The DM packets are designated *medium data rate* packets, since they employ a 2/3 FEC scheme to improve robustness against transmission errors. The DH variants have no FEC at all, only CRC, therefore can provide higher transmissions rates, although with more error probability.

The number 1–5 denotes the number of time slots occupied by packets.

The packet type with largest maximum payload available in Bluetooth is DH5, which can carry up to 339 bytes, while DM5 can only carry 224 bytes. Moreover, the reliability provided by the simple error detection and retransmission in ACL is not sufficient for for many data communication applications. Bluetooth defines another type of channels, called L2CAP (Logical Link Control and Adaptation Protocol), that are built on top of ACL channels and provide *segmentation and reassembly* functionality, which allows MTUs of up to 65535 bytes, as well as improved error detection. Most Bluetooth applications are built on top of L2CAP rather than the more basic ACL channels.

In order to support IP communications in Bluetooth, the Bluetooth Network Encapsulation Protocol (BNEP) has been defined by the Bluetooth SIG. BNEP provides an IEEE 802.3 adaptation layer, which allows regular IEEE 802.3 frames to be transmitted between two devices through L2CAP channels. Since most of the information of an 802.3 frame is redundant (eg. the two Bluetooth devices already know their own addresses, so including MAC addresses in the transmitted data is needless waste of bandwidth), the BNEP layer removes the 802.3 header and inserts its own (smaller) header, before transmitting all in the payload of an L2CAP frame. The receiver device does the reverse operation, thus reconstructs the 802.3 frame header and only then lets the upper layers process it. This way, network protocols such as IP (v4/v6) can work transparently on top of Bluetooth as if it were a regular 802.3 link.

Like everything else in Bluetooth, QoS is supported in a simple fashion. In Bluetooth, like in all polling based systems, the master holds the key to QoS. By controlling when each slave is allowed to transmit, the master has the power to give more or less priority for each slave. At the ACL layer, Bluetooth slave devices may request a specific QoS level using the LMP_quality_of_service_req primitive, containing a *polling interval* parameter. The polling interval is defined as the maximum number of slots a slave has to wait between two consecutive polls. This parameter controls, at the same time, both the maximum access latency and maximum bandwidth available to a slave. At the L2CAP layer, a richer set of QoS attributes is supported:

Service Type Two service types are defined: Best Effort and Guaranteed. Best Effort is the default, and when selected all other QoS parameters are ignored;

Token Rate Analogous to the r parameter in Sec. 2.3.2; average rate at which application transmits data, in octets per second;

Token Bucket Size Analogous to the b parameter in Sec. 2.3.2; used to limit the length and duration of bursts;

Peak Bandwidth Analogous to the p parameter in Sec. 2.3.2; used to limit the rate of bursts;

Access Latency This value represents the maximum delay that may occur between the time an L2CAP packet is queued and the time the packet actually starts being transmitted;

Delay Variation This is a merely informational value that represents the maximum delay variation, or *jitter*, in packet transmissions.

The L2CAP QoS parameters are mapped into the single polling interval parameter at the lower layers. Nonetheless, having an accurate flow description allows the Baseband resource manager (Bluetooth module in charge of scheduling) to make a more efficient use of available resources.

2.6 The QoS Solution in ARROWS

In the IST ARROWS project [RDCR02], the main objective was to study Radio Resource Management (RRM) algorithms for UMTS networks. However, since QoS is ultimately an end-to-end issue, a global QoS framework was devised and deployed. In ARROWS, the choice was made to adopt the RSVP/IntServ framework for several reasons:

1. The QoS resources in UMTS are scarce, thus expensive, so a per-flow QoS solution was thought to be more adequate than something like DiffServ, for example, which deals with aggregate flows and has no explicit, "on demand" reservation;

2. As already stated in Sec. 2.4, it is still possible to combine IntServ at the access part of the network with DiffServ at the core, so the scalability concerns of RSVP don't apply;

3. Since UMTS already provides transparent mobility support at Layer 2, the RSVP problems with Mobile IP also are of no concern in this context.

Still, that left some other problems still to solve regarding IntServ/RSVP UMTS integration, namely:

1. How to map RSVP/IntServ QoS parameters into UMTS Bearer Service QoS attributes;

2. How to map IntServ flows into UMTS PDP contexts;

3. How to handle L2 indications from the network, SMREG-PDP-MODIFY-IND, stating that the QoS previously requested may no longer be available;

4. How to automatically bootstrap the mobile terminal connection (the primary PDP Context) when new IP data arrives for it;

5. RSVP flows are simplex (unidirectional), and RSVP always makes L2 reservations from the *sending* side, which can be either the GGSN or the terminal. In contrast, UMTS Bearer Service only allows activation of PDP contexts with QoS from the *terminal* side. How to deal with this asymmetry;

6. How to map IntServ parameters to the Traffic Class UMTS parameter. The IntServ Guaranteed service, which is the closest to the UMTS QoS model, only has quantitative QoS parameters, but nothing that can easily map into one of the four UMTS traffic classes.

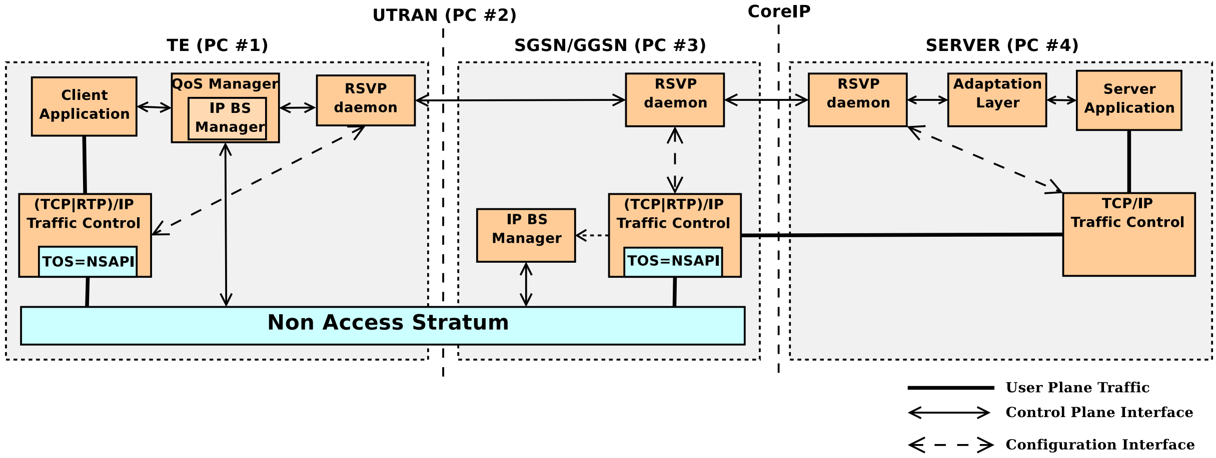


Figure 2.4: ARROWS end-to-end QoS architecture

The ARROWS end-to-end QoS architecture in Fig. 2.3 tries to address these issues. In this framework, applications on the terminal connect to a *QoS Manager* module their QoS requirements, instead of directly reserving resources using the RSVP daemon. The QoS Manager then creates a PDP context and makes an RSVP Guaranteed reservation. An *application type* is among the parameters that the application sends to the QoS Manager, allowing a direct mapping into the UMTS Traffic Class QoS attribute. Other attributes, such as average bitrate and delay, have a more simple mapping. For example, the RSVP reserved bitrate R is simply mapped into an equal value for the Guaranteed bitrate UMTS QoS attribute.

The RSVP protocol was extended to include an additional NSAPI object in PATH messages, as well as in the QoS Manager $\leftrightarrow$ RSVP interface. The RSVP daemon then uses the NSAPI to configure (modified) TBF shaping queues to mark packets' TOS field with this value. Packet marking is done only in the terminal, for uplink flows, and in the GGSN for downlink flows, since it is only required to identify, within the UMTS network, which PDP context each IP packet belongs to.

When the terminal is not communicating, the primary PDP context is shut down in order to save resources. When in this state, if a packet arrives at the GGSN destined to the terminal, the GGSN module IP Bearer Service Manager temporarily holds the packet, sends an SMREG-PDP-ACTIVATE-REQ primitive to the UMTS lower layers, waits for a primary PDP Context to be activated, and then allows the packet to proceed. On the terminal side, the QoS Manager receives an SMREG-PDP-ACTIVATE-IND, and immediately begins activation of a primary PDP context.

Finally, the QoS Manager also assumes responsibility for handling UMTS QoS degradation notifications. In UMTS, when a QoS degradation occurs, an SMREG-PDP-MODIFY-IND primitive is received, containing the new QoS level that can be provided. In ARROWS, the QoS Manager receives this primitive, and informs the respective application about the QoS degradation, which then decides the most appropriate action to take. As proof of concept, in ARROWS a streaming application was modified to handle these QoS degradation events. When QoS degradation occurs, and if the application accepts, the QoS manager automatically changes the RSVP end-to-end reservation to reflect the new local QoS limitations.

2.7 On going research

In this section, some of the more relevant technologies currently being researched are briefly presented.

2.7.1 Subnet Bandwidth Manager

When trying to deploy RSVP/IntServ into IEEE 802 based local area networks (LANs), an unexpected problem became evident: how to perform admission control? Unlike in core network routers, which usually have dedicated point-to-point links, in IEEE 802.3 the physical medium is shared between multiple workstations, and access to the medium is uncontrolled, which means no one single entity in the network keeps track of all the capacity/usage information for a LAN segment. The Subnet Bandwidth Manager [YHB+00, KJS] protocol addresses this problem by allowing a single entity in a LAN segment to be in charge of managing the available resources. Such an entity, called Designated Subnet Bandwidth Manager (DSBM) can be either administratively assigned or appointed through a distributed election algorithm. The DSBM is then used by all RSVP-enabled hosts for admission control.

Useful as SBM may be, it is not without its faults, namely:

- SBM only performs admission control, not scheduling;
- It is too much tied to the RSVP protocol, and not directly reusable outside the IntServ/RSVP framework
- SBM uses IP packets for signalling, which also makes it not reusable independently of the IP protocol;

2.7.2 IEEE 802.21

The IEEE 802.21 Working Group is currently working on handover and interoperability between heterogeneous networks. The latest draft at the time of this writing (July 2005) describes IEEE 802.21 as defining a protocol/layer called Media Independent Handover Function (MIHF). This is an abstraction layer that provides services to upper layers with a uniform interface regardless of any particular layer 2 technologies. Despite being defined by IEEE, the intent of this upcoming standard is to support both IEEE and non-IEEE technologies, such as 3GPP (UMTS). The MIH Function service consists of the following sub-services:

Media Independent Event Service This consists on an abstract interface for receiving layer 2 events, such as *Link Up*, *Link Down*, or *Link Going Down*, which can then be used by the upper layers to trigger a handover.

Media Independent Command Service This service includes a set o primitives that allow the upper layers to control the link layer in order to support the actual execution of handovers. Examples of commands include *MIH Handover Prepare*, *MIH Handover Commit*, and *MIH Configure*.

Media Independent Information Service This provides a generic framework for querying the network for useful information. Three basic groups of information are defined:

General Network Information Provides information about the network itself, such as a list of Points of Attachment (eg. APs), with respective geographical information, network ID, operator, etc.

Link Layer Information Includes information elements regarding the link layer, such as QoS, channel, frequency, security, etc.

Higher Layer Information This is some sort of Service Discovery framework built into MIH, allowing several network services, such as VoIP, email, VPN, to be registered and announced in this way.

One important aspect to be noted is that MIH does not really support QoS in the traditional sense of the term. While it is possible to obtain "QoS" information from APs, it consists only on signal strength information, not available bandwidth. Moreover, MIH does not include any primitives for reserving QoS resources.

2.7.3 IEEE 802.11e

In traditional IEEE 802.11 [CWKS97] (Wireless LAN), there are mainly two types of MAC protocols available: Distributed Coordination Function (DCF) and Point Coordination Function (PCF). With DCF, access to the medium uses a CSMA/CA scheme, as in 802.3 networks, which means that stations that want to transmit have to listen and wait until the medium becomes free, at which point they start transmitting (after a small random delay). If during transmission a collision is detected, it is aborted and retried at a later time, until it succeeds. Clearly the type of service provided by this MAC is always best effort, and there are bound to happen a lot of collisions, with consequent performance degradation. The Point Coordination Function (PCF) is an alternative MAC protocol for 802.11 that is available only in infrastructure mode. With this MAC protocol, there is a time frame that is delimited by the beacons regularly sent by the AP. This time frame is divided in two periods: Contention Free Period (CFP) and Contention Period. While in the CP access to the medium is distributed, like in DCF, in the CFP the AP periodically polls each station, giving it an opportunity to transmit. Stations are not allowed to transmit inside the CFP unless being polled. Through this polling scheme, QoS support is possible, but unfortunately there is only limited QoS in the PCF MAC protocol, and PCF is not widely deployed in commercial equipments.

The new IEEE 802.11e draft aims to provide QoS extensions [GZ03] for 802.11 networks. Like 802.11, two MAC protocols are defined: Enhanced Distributed Coordination Function (EDCF) and Hybrid Coordination Function (HCF).

EDCF is a distributed MAC similar to DCF. The main difference is that EDCF allows assigning different priorities to stations by manipulation of the values of the *Contention Window*. The Contention Window is the amount of time a station must wait before being allowed to transmit after the medium becoming idle. By assigning a lower contention window value to some stations an not the other, these stations gain a competitive advantage when accessing the medium and have to wait less time, in average, to transmit their frames.

The EDCF offers only relative guarantees. The HCF, on the other hand, uses a polling scheme like PCF, but with a much richer QoS control model. For one thing, each station regularly sends to the AP feedback regarding the size of its own transmission queue, thus the AP may use this information to adjust the priorities when polling each station. In addition, multiple Traffic Classes are defined, and stations are allowed to transmit multiple frames in one burst.

2.7.4 GMPLS

Although MPLS is an architecture that fits well into packet-switched network equipments, there is a whole class of equipments that only deals with bit streams and doesn't even see packet boundaries. Thus, they are unable to even look at an MPLS shim header, much less *swap* MPLS labels. Still, these equipments could also benefit from the management flexibility offered by the MPLS architecture. Generalised MPLS [ASMA04, NR05] is an extension to MPLS that allows just that. Although an equipment may be unable to inspect a traditional MPLS label, GMPLS extends the definition of label by including some types of context information that are implicit in a bitstream, such as time slot number (in time division multiplex systems), or wavelength (optical switching fabrics). Even in packet switching equipments GMPLS may be interesting if the link layer already provides multiple virtual-circuits (eg. ATM, Bluetooth), since it eliminates the need for a redundant shim header.

2.7.5 Generic Link Layer

The Generic Link Layer [BBB+05, GSL+05] (GLL) is an L2 abstraction layer that is being developed in the context of the WWI Ambient Networks project. It presents some similarities to the work described here; in particular, some vague ideas about L2 QoS configuration through a generic interface are mentioned. However, the approach followed is rather different. Whereas the QoSAL tries to be as little intrusive as possible by not even introducing any additional header into the data plane, GLL essentially re-implements many of the L2 features, such as ARQ, flow control, and SAR (Segmentation And Reassembly).

2.7.6 ULLA

The *Unified Link Layer API* (ULLA) [FGI+05] features an abstract API for configuring radio link parameters and receiving generic handover triggers. It is similar to 802.21 in purpose, but is more limited in the sense that it is a host local interface; no protocol is defined for delivering commands or events across L2 network segments. Like 802.21, it does not cover L2 QoS.

2.8 Conclusions

In this chapter some of the most relevant QoS architectures and technologies were presented.

Regarding L3 QoS architectures, it was shown that the IntServ/RSVP QoS model, while providing solid end-to-end QoS guarantees, is not scalable in core networks and does not handle mobility well. DiffServ represents an alternative approach to QoS that does not suffer from the scalability problems of RSVP. However, it relies heavily on over-provisioning and provides only relative guarantees, thus is not very effective in wireless access networks. It was shown that recent L2 technologies, such as UMTS, Bluetooth, and IEEE 802.11e, are increasingly incorporating QoS support.

The IST ARROWS research project successfully demonstrated how to bind a L3 QoS architecture to a QoS-capable L2 technology, but is L3-specific (IntServ/RSVP) and L2-specific (UMTS). MPLS is an interesting architecture that sits between layers 2 and 3 and allows one to establish "virtual circuits", thus making traffic engineering actually realizable in core networks. It also supports QoS, but only in the sense of finding a path that satisfies QoS constraints, not making QoS reservations across a path.

Chapter 3

The QoS Solution in DAIDALOS

3.1 Introduction

The IST research project DAIDALOS (**D**esigning **A**dvanced network **I**nterfaces for the **D**elivery and **A**dministration of **L**ocation independent, **O**ptimised personal **S**ervices) [Dai] provides the motivation and context for the work described in this thesis. The main focus of DAIDALOS is to develop an architecture for fourth generation wireless networks that meets the following goals:

- Should support multiple wireless technologies, such as WLAN/802.11, WMAN/802.16, TD-CDMA, and Bluetooth, instead of designing an architecture ultimately tied to a single technology;
- Users should be shielded from having to manage all the complexity that the heterogeneity of access technologies implies;
- There should be support for third-party services, which communicate with an operator's network infrastructure with a well defined and open interface;
- At the same time, users should not have to manage the complexity of having services from outside an operator. Using external services should "just work", with no manual configuration required;
- Quality of Service should be supported in the operator's base architecture, the service provider, and end-to-end in an integrated fashion, while at the same time transparently to the user;
- Handover between (1) different access technologies, and (2) different domains/operators, must be supported, while preserving QoS as much as possible.

The DAIDALOS project generally follows certain technical design choices that favour reusing and improving existing technologies rather than starting design from scratch, as well giving preference to general solutions, rather than technology specific ones. Therefore, IPv6 is chosen as technology abstraction data plane layer. For mobility support,

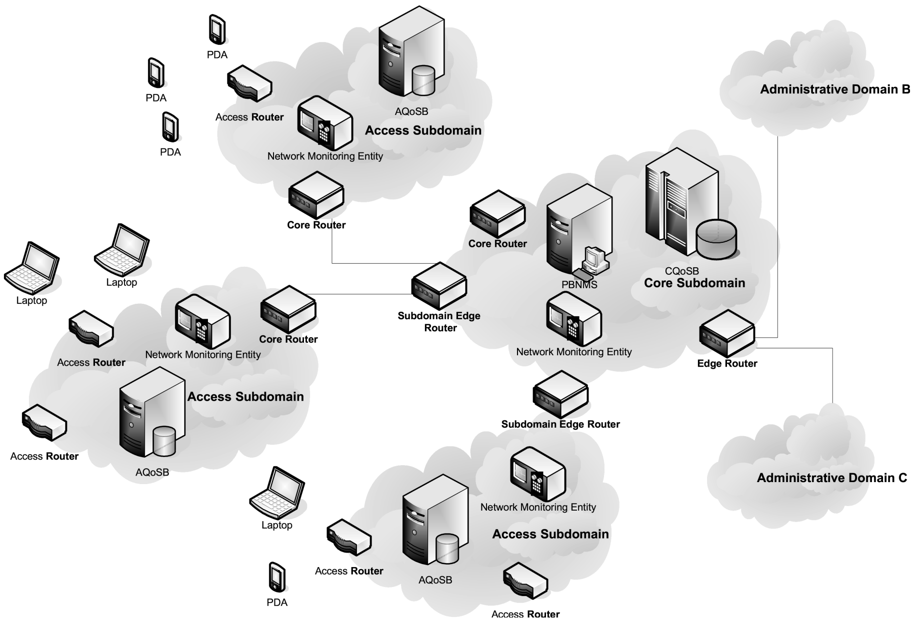


Figure 3.1: Daidalos network architecture (source: DAIDALOS Deliverable D321)

Mobile IPv6 with Fast Handover [KDEM04] is used. Moreover, whenever possible IETF protocols, such as SIP and COPS, are reused.

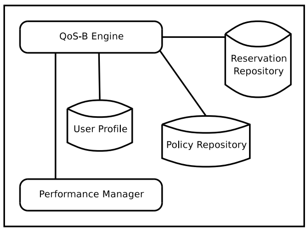
The remainder of this chapter briefly describes the overall DAIDALOS architecture, with special emphasis on the QoS aspects.

3.2 Architecture

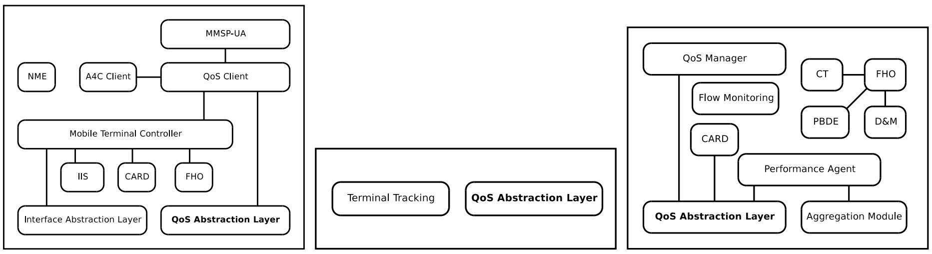
Fig. 3.1 shows an overview of a typical DAIDALOS network, while Fig. 3.2 shows the most relevant modules in this context. Access Points are not represented in Fig. 3.1 for simplification; they are located between terminals and ARs, and are used to provide wireless connection to terminals and connect them to the ARs. The coexistence of several *administrative domains* is contemplated in the architecture. Typically, an administrative domain is controlled by a single network operator, although conceptually a single operator could control several administrative domains. Each administrative domain is divided into one *core subdomain* and several *access subdomains*. The core subdomain interconnects all the access subdomains, through *Subdomain Edge Routers*, and interfaces with external administrative domains, through *Edge Routers*.

3.3 Service and Network Management Architecture

The *Service and Network Management* subsystem corresponds to the DAIDALOS Work Package 3. As the name suggests, the focus of this WP is to develop a flexible architecture



(a) Access QoS Broker



(b) Mobile Terminal (c) Access Point (d) Access Router

Figure 3.2: Detail on the most relevant modules in the Daidalos architecture

that provides the necessary infrastructure for deployment of a varied array of services that represent added value to the end user. At the same time, a robust management architecture is developed, allowing network managers to evaluate the adequateness of the network performance for the services that are being supported at any given time, as well providing the ability to take corrective measures on the same network in case the performance goals are not met at some point in time.

This section will briefly describe the most important components in this architecture in Sections 3.3.1– 3.3.6, as well as shed some light on the QoS reservations strategies that are supported by this platform in Section 3.3.7.

3.3.1 QoS Brokers

The end-to-end QoS model followed in DAIDALOS is based on the concept of *QoS Brokers*. In this architecture, each QoS Broker (AQoSB and CQoSB in Fig. 3.1) is responsible for managing the resources for one subdomain. Thus, each access subdomain is managed by a different QoS Broker, as well as the core subdomain. In order to reserve end-to-end resources, first a QoS reservation request is sent to the Access QoS Broker (see Sec. 3.3.7). It checks that there are enough resources and the access subdomain, by contacting the Access Router's QoS Manager, then propagating the request up through a chain of QoS Brokers that may include the Core QoS Broker, and other QoS Brokers in a different domain, depending on the path for the reservation. The

QoS Brokers communicate using a protocol based on the Common Open Policy Service (COPS) protocol [DBC+00].

The QoS Brokers manage resources using the *Policy Based Network Management* (PBNM) concept. To this end, they receive policies from the Policy Based Network Management System (PBNMS) and cache them in a local *policy repository*. In addition to receiving policies from the PBNMS, the QoS Brokers also generate *alarms* back to the PBNMS to report anomalous situation that may occur, such as prolonged lack of resources, or failures.

The Access QoS Brokers have additional responsibilities. Besides managing QoS resources, they also receive a subset of the users' profiles from the A4C module (not represented in in Fig. 3.1), and it is their responsibility to verify users authorisation for each resource being requested. Moreover, resource checking and authorisation is performed by the Access QoS Broker when users' terminals handover between different Access Routers.

3.3.2 Core Router

The *core routers* in DAIDALOS contain a DiffServ QoS engine, with a COPS based interface to the QoS Broker. The *QoS Manager* is a module in the core router architecture that is responsible for managing the DiffServ resources and interfacing with the QoS Broker. Thus, the QoS Broker delegates the lower-level QoS resource management tasks to the QoS Manager located in each core router. There is also an interface to the Network Monitoring System that is used by the QoS Manager to send queue load information to the Network Monitoring Entity (NME).

3.3.3 Policy Based Network Management System

The Policy Based Network Management System (PBNMS) entity is divided into a Network Server that interacts with the Network Elements, and a Human Server that interacts with the Human Operator. The Network Server holds the central policy repository, which is distributed to the Network Elements (eg. QoS Broker) on startup, and receives alarms from them. The Human Server is an http server providing a user interface to a human operator.

3.3.4 Network Monitoring Entity

The Network Monitoring Entities (NMEs) are spread across the network a strategic locations, and employ both passive and active network monitoring strategies to gather network statistics useful for network management. In active monitoring strategies, test packets or flows are introduced into the network, and measurements are taken upon them to evaluate how are they are affected when travelling across the network. The best example of active monitoring technique is the plain old `ping` utility, which can let us determine packet losses, delays, and number of hops, to reach any destination. As a side effect, active monitoring introduces additional load into the network, therefore the test flows must be small, which unfortunately means that they no longer may be able to simulate with precision the conditions of the real network traffic. Passive monitoring, on the other hand, does not introduce test flows, but merely measures the real traffic, as it travels across a network. Passive monitoring is ideal to monitor services' compliance with previously established Service Level Agreements (SLAs), for instance.

The NMEs communicate with a Central Monitoring System (not represented in in Fig. 3.1) using a XML-RPC interface for configuration, and a Netflow or IPFIX interface for data reporting.

3.3.5 Access Router

The Access Router in DAIDALOS contains an *Advanced Router Mechanism* (ARM) entity, which includes a *QoS Manager*.

The QoS Manager is responsible for some QoS-related tasks, including:

Policing/shaping and scheduling Application flows are shaped/policed and scheduled according to the policy controlled by the QoS Broker, so as to implement a DiffServ functionality;

DSCP remarking Also in accordance with policy indicated by the QoS Broker, DSCP remarking may have to be done in order to implement a complete DiffServ functionality;

QoS context transfer The QoS Manager maintains a state of allocated QoS resources on a per-user basis, and automatically initiates a context transfer of this state when a handover to a new access router takes place;

Load monitoring The QoS Manager constantly monitors the state of the shaping queues, and sends this information to the NME, which then forwards it to the Central Monitoring System (CMS);

Reserve L2 QoS resources Finally, and most importantly in the context of this thesis, the QoS Manager interfaces with the QoS Abstraction Layer in order to reserve L2 QoS resources in the wireless interfaces[1].

In addition to the functionality provided by the QoS Manager, ARM also supports QoS Tran-signalling, multicast, and advanced signal processing (eg. sniffing SIP signalling packets to extract QoS parameters).

3.3.6 QoS Client

Although the DAIDALOS network architecture can cope (with a somewhat limited service level) with *legacy terminals*, which do not support any DAIDALOS-specific QoS protocols, there is a DAIDALOS *QoS Client* module that may run on terminals to provide an enhanced service relative to what can be provided from the network-side alone. Some of the responsibilities assigned to the QoS Client include:

[1] Or wired interfaces connected to the wireless APs

Control QoS Signalling The QoS Client can act on behalf of the application to control signalling of particular flows. Both in-band signalling (DSCP marking) and out-of-band (eg. end-to-end RSVP or COPS message to the QoS Broker) are supported by the QoS Client;

Signalling for Legacy Applications The QoS Client contains a list of well known applications/ports, along with recommended QoS settings, and is able to automatically reserve QoS resources when these legacy applications start sending traffic;

Packet Marking There are two separate reasons to perform packet marking:

1. DSCP marking may be required for the in-band QoS signalling mentioned above;
2. As part of the L2 QoS reservation “contract”, the QoS Abstraction Layer expects that packets that are to be tunnelled through any particular QoSAL connection be marked with the respective *connection identifier* in the IPv6 Flow Label field. While the QoS Manager takes care of setting up packet marking for downlink flows, in case of uplink flows it has to be the QoS Client to do this task.

React to QoS Level Changes The QoS Client can react to QoS degradation notifications (eg. from the QoSAL) and send events to the Multimedia Service Provisioning User Agent (MMSP-UA), which may, for example, start a transcoding process in order to match the application QoS requirements to the new level that can be provided by the network.

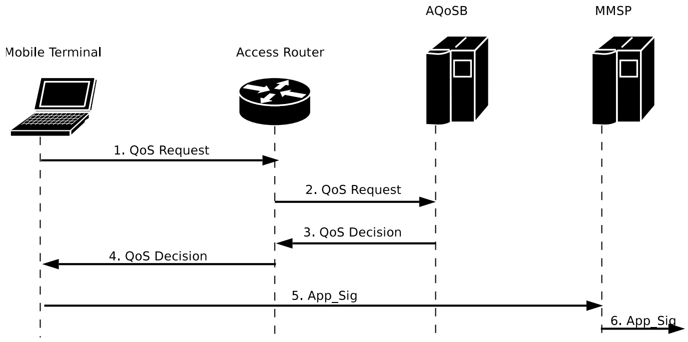
3.3.7 QoS Reservation Strategies

A DAIDALOS access network is sufficiently flexible to support multiple QoS reservation strategies:

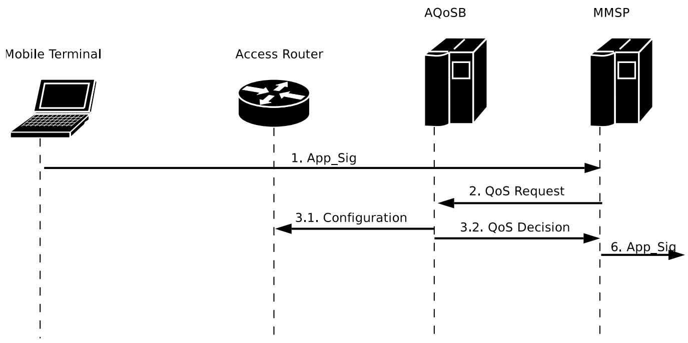
Terminal issues QoS requests In this reservation strategy (Fig. 3.3(a)), it is the terminal that first takes initiative of contacting the QoS Broker via the ARM, asking it to reserve resources for a flow. After A4C verification of the request, the QoS Broker pushes the request up the chain of QoS Brokers and orders the AR to make a reservation for the AN;

Service proxy issues QoS requests In this scenario (Fig. 3.3(b)), the application contacts a service proxy, and exchanges signalling which may or not include QoS parameters. The service proxy then issues a QoS request to the QoS Broker. After A4C verification of the request, the QoS Broker pushes the request up to the core QoS Broker and orders the AR to make a reservation for the AN;

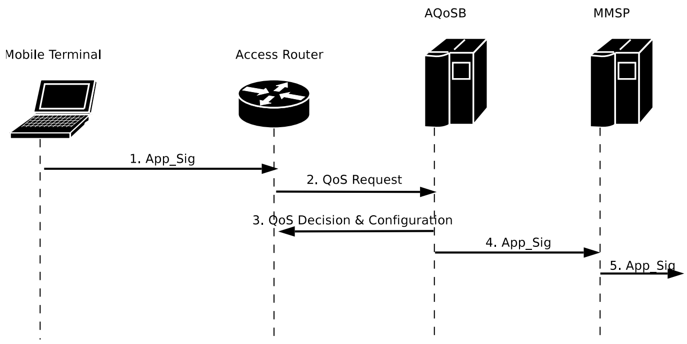
Terminal issues QoS requests through ARM In this scenario (Fig. 3.3(c)), the application performs normal signalling, without caring about DAIDALOS QoS interfaces. However, the AR intercepts the application signalling and extracts



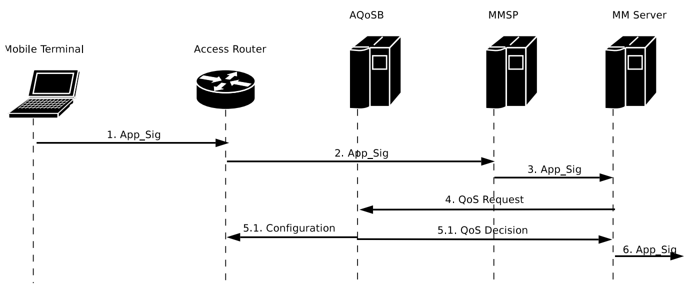
(a) Terminal issues QoS requests



(b) Service proxy issues QoS requests



(c) Terminal issues QoS requests through ARM



(d) Application server issues QoS requests

Figure 3.3: DAIDALOS QoS reservation strategies

the QoS and flow descriptor, thus is able to request reservation of QoS resources through the QoS Broker, as in the other scenarios. In alternative to intercepting application signalling, the AR can also look at the DSCP of incoming packets, previously marked at the terminal, to discover the QoS settings for each flow;

Application server issues QoS request In this instance (Fig. 3.3(d)), the user terminal sends a normal request to the application server as usual, and it is the latter that then requests QoS resources back to the access network.

Which reservation strategy is the best is, probably, a question with no direct answer. While *Terminal issues QoS requests* is certainly the most flexible approach to QoS reservation, since it does not require any single entity to be aware of every application the user may want to use in his terminal, it requires a "DAIDALOS aware" terminal which, although desirable, is not possible in all cases. The *Service proxy issues QoS requests* strategy, on the other hand, does not require the terminal to use any DAIDALOS specific QoS interfaces but, on the other hand, only works for applications using the correct multimedia proxy. There is no such problem with the *Terminal issues QoS requests through ARM* strategy, but unfortunately it will only work for a handful of applications that the network is programmed to handle, as well as subjecting the AR to the intensive task of monitoring the contents of signalling packets. The *Application server issues QoS request* strategy simplifies charging when the application server belongs to a different administrative domain, since the server buys the QoS resources to the access network, and then the user is charged a single time for application service + network QoS.

3.4 Network Integration Architecture

The *Network Integration Architecture* subsystem corresponds to the DAIDALOS Work Package 2. It addresses the infrastructure problems related to the lower layers of the access part of the network, such as mobility support, Layer 2 QoS, broadcast, security, and ad-hoc networking. Besides the WP2 QoS architecture, which is already described in full detail in Chapter 4, in this section only the mobility subsystem will be presented.

3.4.1 Mobile IPv6 Soft-handover

Mobile IPv6 [JPA04] with Fast Handover [KDEM04] extensions, used in DAIDALOS, already represent a great improvement in handover speed relative to traditional MIPv6. Still, there is a short period of time during handover when the terminal is physically (L2) connected to the new AP but is not yet registered with the AR. During this period of time, packets are still being sent through the old AP, thus lost by the terminal. Although it is a short period of time, it can have a noticeable impact on the QoS if for example the user is having an audio conversation.

To work around this limitation, the concept of *soft-handover* is borrowed from UMTS, allowing the network to send flows through both old and new ARs, thus guaranteeing that the terminal will receive at least one copy of each packet regardless of the AR it is registered with. This technique works best for L2 technologies that are able to stay connected to more than one AP/AR at the same time, although it still produces noticeable improvements otherwise. It also works well for terminals with multiple interfaces, even of different technologies.

A key element in the realization of this concept is the Duplication & Merging Agent (D&M), which runs in a router that sits "upstream" to all the ARs that cooperate in soft-handover. During handover, it automatically intercepts all packets destined to the terminal, tags them with a unique sequence number, and then sends one copy tunnelled through both old and new AR. A similar agent also runs in the terminal; it takes care of dis-encapsulating packets from the tunnel and remove any duplicates found, as identified by the sequence numbers. For uplink flows, the reverse operation happens, i.e. the D&M agent in the terminal duplicates and tunnels packets through old and new ARs, and the agent in the core network router eliminates duplicates.

3.4.2 Interface Abstraction Layer

The Interface Abstraction Layer (not to be confused with the QoS Abstraction Layer) is a library/toolkit that resides on the mobile node and provides the ability to enumerate network interfaces and obtain several properties from these interfaces to help with mobility issues; at the same time it maintains an abstract API, i.e. technology independent. The following functionality is made available by the IAL:

- Detection, enumeration, and identification of network interfaces, including technology/product information;
- Provision of triggers when an interface changes status (eg. if an interface disappears as the corresponding PCMCIA card is removed);
- Reporting of information available on interfaces, such as available channels, normalised signal strength/quality values;
- Channel selection on interfaces;
- Ability to query or modify both generic and device-specific options.

3.4.3 Intelligent Interface Selection

4G networks will offer unprecedented flexibility in terms of technology support. That also means increased complexity for the end users, which is precisely one of the problems DAIDALOS tries to address.

In order to relieve the user from manually having to select, in real time, between different APs, different technologies, or even different domains, an *Intelligent Interface Selection* (IIS) module has been designed. It is located at the terminal, and automatically switches between different interfaces, APs, or ARs, on behalf of the user. The selection criteria is based on user preferences which can include, for example, relative importance of QoS over cost, relative preference of network providers, or even enabling the use of multiple interfaces simultaneously for different applications/services. The user preferences may be stored on the Home Agent or on the terminal itself. The interface selection algorithm receives as input parameters such as a list of available APs and ARs, as well as respective characterisation information (eg. available bandwidth, type of technology, network provider, etc.) obtained through the *Candidate Access Router Discovery* (CARD) protocol.

3.4.4 Performance Manager

In DAIDALOS, two different handover styles are supported. On one hand, there is *Mobile Initiated Handover*, in which the mobile terminal takes initiative to execute a handover to a new, better AP/AR, normally triggered by IAL, and a new target is AP/AR selected by IIS. On the other hand, a *Network Initiated Handover* strategy is also supported, where the terminal receives indications from the networking, suggesting that it performs a handover to a different AP or AR, so as to attain a better load balance among an operator's set of APs and ARs. The key to network initiated handover is the Performance Manager (PM) module; the PM functionality is distributed among several different entities deployed in different nodes:

Terminal Tracking Module It is located in the AP; it performs passive scanning of terminals, and sends measurement information from the point of view of the network side back to the AR, along with information regarding neighbouring APs;

Performance Attendant This module is located in the AR, and it contains the following submodules:

- **Aggregation Module** This is the module that is responsible for receiving the information sent by the Terminal Tracking Module from the APs. In fact, it collects information from several APs and sends it to the PM module in the QoS Broker;
- **Policy Based Decision Enforcement** This module receives the output of the PM algorithms from the QoS Broker, then runs any handover decisions through a local policy database. If the policies allow it, then a signal is sent to the handover module to begin a network initiated handover;

Performance Manager The actual PM algorithm is included as a QoS Broker module. It receives input from the various Performance Agents in all ARs and makes handover decisions when appropriate in order to balance the load of terminals among the set of available APs and ARs.

3.4.5 Mobile Terminal Controller

The Mobile Terminal Controller (MTC) module provides central coordination functions among a set different modules in the terminal, including IIS, IAL, and QoS Client. Some of the information that the MTC is able to manage includes:

- The MT's current status, such as interface currently in use, L2 address of the AP it is connected to, IP address of the AR, and list of services running on the terminal with corresponding QoS levels, and current Care of Address (CoA);
- Characteristics of next handover target, including target interface, L2 address of target AP, IP address of target AR, and new/next CoA;
- A list of APs detected by the terminal, together with their L2 addresses, corresponding local interface, technology type, and signal quality;
- A list of handover candidate ARs, along with corresponding IP addresses, APs connected to them and their available bandwidth;

During handover, the MTC is responsible for deselecting the old interface and selecting a new one. It also triggers IIS operation when detecting (via IAL indication) deteriorating signal strength. Finally, it interacts with QoS Client and security modules.

3.5 Conclusions

The DAIDALOS architecture is complex and thorough. Nearly all network aspects of a 4G network are covered by this architecture, from the high-level policy-based network management system, to network monitoring, service provisioning, a flexible QoS system, and advanced mobility support. This still leaves out subsystems not covered here, such as ad-hoc networking, security, pervasive systems, and broadcast.

Within the QoS part of the overall architecture, the QoS Abstraction Layer plays a central role. Every service in DAIDALOS requires one form or another of QoS. Moreover, all user flows invariably involve the access network. Usually, there are plenty of resources available in the wired part of the network, so the QoS bottleneck is in the wireless part. Since the QoSAL controls the admission control in the wireless part, it plays an important role in end-to-end admission control. Moreover, wireless part of the network is the one that experiences the most delay. Therefore, QoS support in the wireless part, as realized by the QoSAL, is the most important in all the end-to-end path.

Chapter 4

The QoS Abstraction Layer

In this chapter, a detailed specification of the module called *QoS Abstraction Layer* will be presented. First, Sect. 4.1 enumerates the functional requirements that have been considered in the design. These requirements in a way are aligned with the objectives and contributions described in in Sec. 1.2 and 1.3. Then, in Sec. 4.2 a detailed description of the solution will be presented, including architecture, modules, interfaces, and protocol. Finally, some conclusions are included in Sec. 4.3.

4.1 QoSAL Requirements

The functional requirements, as presented in this section, are divided in several groups. Sec. 4.1.1 lists the QoS requirements. These are the main requirements that motivated this work in the first place. Sec. 4.1.2 lists the requirements related to mobility support. Additionally, Sec. 4.1.3 lists "autoconfiguration" requirements. Finally, some other requirements are listed in Sec. 4.1.4.

4.1.1 QoS Requirements

The QoS requirements pertain to the set of characteristics that the QoS reservation model must observe.

Reservations The main goal is to allow reservation of QoS resources across an arbitrary wireless network. The motivation for this goal is clear and already explained in this document;

Abstraction The QoS parameters should be simple and generic, and each AP must perform the necessary mapping to its own technology. The reason for this is that the QoSAL should not favour any one specific technology, therefore its QoS parameters must be generic. At the same time, there is no point in trying to require too detailed QoS parameters, since a great loss of QoS resolution is to be expected in any case due to the highly heterogeneous environment the QoSAL is expected to run on.

Flexibility The offered QoS model must be flexible, so that in can be reused in multiple end-to-end QoS architectures. In particular, some effort must be made to make it easily deployed to support the most common QoS models currently in the Internet, such as IntServ [BCS94] and DiffServ [CDWW98], as well as any new one likely to be deployed, such as NSIS [HKL04];

4.1.2 Mobility Requirements

The fact that the QoSAL runs in environments where mobility is a core concern means that it must actively support mobility.

Smooth handover There should be support for *smooth* handover of terminals between APs or ARs. This means that the QoS for flows with QoSAL reservation for any given terminal should not be significantly degraded as the terminal moves from one AP/AR to a new one.

Triggers It should provide network load and QoS degradation indications to trigger handovers. Contrary to many handover trigger implementations, the signal strength alone is not a good indication to trigger handover. What really matters for a terminal is not the signal strength but what level of QoS can be obtained with this signal. Of course, the two are highly correlated, which helps explain why in many instances signal strength is used as a substitute for QoS, but in truth it is the latter that really counts. As an example, consider an IEEE 802.11 terminal whose signal strength at some point decreases to 25%. Whether it should handover or not does not depend on the signal strength alone. If the AP is lightly loaded, then the terminal can switch to a different transmission mode; a slower one, but with more energy per bit, to compensate for the decreased signal power. Sure, that terminal is occupying the wireless medium for much longer, thus wasting more resources, but since the AP was not very loaded in the first place then it does not have much impact. If, on the other hand, the AP was already moderately or highly loaded then the extra time spent by the terminal in the slower transmission mode is prejudicial to the overall resource efficiency of the AP, or perhaps not even possible due to the high load.

4.1.3 Autoconfiguration Requirements

The QoSAL protocol should be able to transparently adapt to complex IEEE 802 style networks. In particular, the following aspects should be considered:

Auto-routing No a priori knowledge of the L2 network topology should be required by either AR or MN. In an IEEE 802 network, it is possible to interconnect equipments (eg. switches) in any way we like without worrying about which ports in which switches a frame needs to go through to reach its destination; we just fill in the destination MAC address and send it, and we *know* the frame will somehow find its way to the destination host. The QoSAL should somehow capture this "auto-routing" concept from IEEE 802 networks, since it brings great improvements in network setup time.

Concatenated networks Multiple concatenated wireless links should be transparently supported, and QoS reserved in all of them. As already explained, and depicted in Fig. 1.1, one of the likely scenarios in 4G networks is the concatenation of multiple wireless hops, such as a long ranged but *fixed* IEEE 802.16 link followed by shorter-ranged but *mobile* IEEE 802.11 or 802.15.1. From the point of view of the IP layer, all these concatenated wireless links are treated just as a simple L2 network segment, without knowing or caring whether the L2 segment is split into multiple segments of heterogeneous wireless technologies. What is true for datagram sending ought to be true also for QoS reservations, otherwise we defeat the builtin autoconfiguration mechanisms in IEEE 802 equipments.

Dynamic adaptation Dynamic changes in the L2 topology shall be supported. Sometimes the topology of a complex L2 network has to change. First of all, there is the problem of terminal mobility. Even terminals are part of the L2 network, and as they move from one AP to another they are implicitly changing the topology. Other topology changes that can happen are related to network management issues, such as taking out an AP that has stopped working, or adding new APs to increase an AR's coverage or bandwidth capacity. These topology changes are only natural, and should continue not to require manual reconfiguration.

Legacy equipments The QoSAL protocol should work transparently in the presence of L2 equipments that are unaware of the QoSAL protocol. For instance, in a concatenated scenario, if one of the APs doesn't support QoSAL, the reservations still should proceed to the next segments, and QoS would still work in these other segments in the path.

4.1.4 Other Requirements

Some additional requirements, which do not fit in any of the above categories, are presented here:

Events Similar in spirit to the *triggers* requirement described in Sec. 4.1.2, the QoSAL should provide events, or indications, although with different goal, which is to allow cross-layer optimisations. As explained in [CRR04], the upper layers, such as TCP, RTP, and above, can draw great benefits from having more detailed and real-time information of events that take place at the link layer. Adjusting the TCP congestion window to the wireless link conditions as they change over time, and changing the encoding parameters of a streaming session to adjust to the ever changing channel state are two of the most compelling examples of the kind of improvements that can be achieved with a proper events interface in place at the link layer;

Robustness In a wireless environment, it is often impossible to guarantee accurate transmission at certain points in time, which means there's a significant probability a transmitted frame will be received with errors. Sometimes, an effect called *fading* occurs, meaning that for a period of time the signal strength decreases to almost nothing[1], which means sometimes a frame cannot be transmitted, even with ARQ. This is a harsh environment, but the QoSAL must cope with it;

Multicast A multicast QoS abstraction should be supported. In 4G networks, as operators strive to offer video streaming of real-time events, such as football matches, it will not be scalable, or cost effective, to use the traditional unicast traffic model, due to the prohibitely high bandwidth load that would be produced on the access network. Thus, multicast support will play an important role in 4G networks, as it allows the "send once, charge many times" philosophy. In fact, multicast is already being introduced into UMTS networks [Iva05]. Thus, the QoSAL should contain primitives to reserve QoS for multicast sessions as well;

Modularity The architecture should be highly modular, allowing support for new technologies to be easily "plugged into" the architecture. The motivation for this requirement is simple to understand. As new modulation and transmission techniques are developed, new wireless technologies are going to be developed, possibly including changing service interfaces, which means that adaptation modules for the QoSAL interface has to be created, allowing the new technologies to be seamlessly integrated in the overall architecture. However, many of the QoSAL requirements are common to all technologies, and their solutions essentially technology independent. Therefore, if the QoSAL had to be fully implemented for each technology, there would be a lot of duplicated work. To avoid this duplication, the QoSAL architecture should clearly isolate the technology independent functionalities, and allow technology dependent modules to "plug in" support for each supported L2 technology.

4.2 QoSAL Specification

In this section, the QoS Abstraction Layer specification is presented. Starting with Sec. 4.2.1, where the overall architecture is described, then moving on to Sec. 4.2.2, which specifies in detail the *service interface*, followed by a specification of the QoSAL protocol in Sec. 4.2.3, and finally the *driver interface* is presented in Sec. 4.2.4.

4.2.1 Architecture

Modules

The architecture that has been designed is summarised in Fig. 4.1. It represents a typical access network, with an AR, an AP, and a MN. The QoS Abstraction Layer (QoSAL)

[1] For example due to specific phase alignments between multiple reflections, which change as a terminal moves

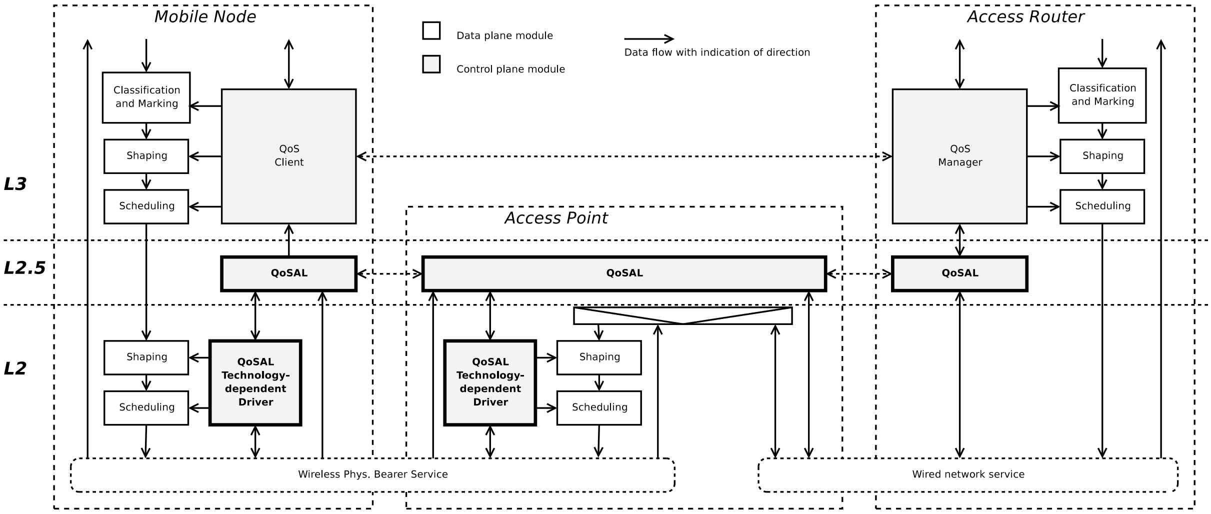


Figure 4.1: QoS Abstraction Layer architecture

runs on all these network elements. Note, however, that the architecture is not limited to a single AP; multiple APs can stand between a MN and an AR. At the AR, the QoSAL accepts service requests (Sec. 4.2.2) from the *QoS Manager*; it is responsible for end-to-end QoS management, but only deals with IP subnetworks. For instance, it is unaware of the APs in Fig. 1.1. The reservation of resources in the L2 network is delegated to the QoSAL.

Control Plane L3-L2 QoS Mapping

The QoSAL instances located in different L2 nodes communicate using a custom protocol that is transported directly over L2 bearers (see Sec. 4.2.3). In response to abstract QoS requests from the QoS Manager, the QoSAL modules running on AP and MN ask the technology-specific *QoSAL Driver* modules, using the driver interface described in Sec. 4.2.4, to implement the request at L2, which usually means to configure data-plane modules (e.g. shaping and scheduling), and prepare them for the new flow. Alternatively, the QoSAL driver could simply translate the abstract request into an L2 primitive. For instance, in an UMTS or GPRS network interface, at MN side, the primitive `SMREG-PDP-ACTIVATE-REQ` can be used to reserve L2 QoS resources.

Conceptually, the QoSAL reservations are like “virtual channels” across a whole L2 network, whose boundaries are determined by a pair of L3 nodes, with an attached QoS service level. These virtual channels are denominated “QoS connections”. As a set of packets enter these connections, as explained in Sec. 4.2.1, they are transported to the other end of the tunnel with the QoS guarantees that have been negotiated by the QoS reservation. For each QoSAL connection there exists an integer number that uniquely identifies it within a *Link Access Network* (LAN). A LAN is composed by the set of L2 nodes, including APs and MNs, reachable by a single network interface of an AR.

Table 4.1: The DAIDALOS QoS classes and their associated parameters

Parameters	**Class 0 Conversational**	**Class 1 Transactional**	**Class 2 Streaming**	**Class 3 Best Effort**
Delay — upper bound on mean delay (end-to-end)	150 ms	400 ms	1 s	unspecified
Delay — upper bound on mean delay (LAN)	40 ms	100 ms	250 ms	unspecified
Packet loss — upper bound on the packet loss probability	1×10^{-3}	1×10^{-3}	1×10^{-3}	unspecified
Designed for:	Interactive voice and video (e.g. audio and video conferencing)	Transaction data, interactive (e.g. Web browsing, telnet, e-commerce)	short transactions, bulk data, video streaming (e.g. video on demand, ftp)	Legacy applications / low cost services

The QoS parameters used in QoSAL are basically the same as the ones used in end-to-end QoS reservations, namely *Class Identifier*, *Reserved Bitrate*, and *TSpec*. The Class Identifier is used to indicate to the QoSAL what type of traffic will be transported over the connection. Table 4.1 lists the available class identifiers in Daidalos. Each class identifier implicitly defines some QoS parameters, such as packet loss probability and delay. Only the LAN-part of the delay applies for the QoSAL connections, and it is about one quarter of the maximum allowed end-to-end delay. The Reserved Bitrate is the most important parameter in the QoS reservation, as it indicates the amount of bandwidth that is to be reserved in the wireless network. Finally, the TSpec parameter has the same meaning as in RSVP/IntServ, and is used as indication to the L2 QoS elements regarding the properties of the flow that is to be transported over the QoSAL connection. The L2 QoS equipments may use this information to scale buffers, for example, or, by the same token, reject reservations that would require buffers too large for them to handle.

Data Plane L3-L2 QoS Mapping

The QoSAL, as protocol, only exists in the control plane of the communications stack. But, in the data plane, there has to be an interface for L3 to indicate to the L2 QoS modules what kind of QoS treatment to apply to each packet. While at the IP layer individual flows may have separate queues where packets are shaped, they are eventually multiplexed into a single queue for transmission by the network card. This is how IP, and in particular the Linux network stack, normally works, since it was not designed with L2 QoS in mind. Before queueing packets for transmission by L2, however, the L3 modules *mark* the IPv6 *Flow Label* field of all packets with a *Connection Identifier*, which is a number used by the QoSAL-aware network interfaces to uniquely identify a reservation/flow (or "connection", using the service interface terminology). The same Flow Label is used throughout the whole L2 network, including AR, AP(s), and MN, to map packets into flows.

Currently, only the IPv6 Flow Label is used to associate packets with QoS connections, although a mapping based on L2 or even new L2.5 headers is under consideration (see Sec. 6.3). Using Flow Label for QoS mapping is a double edged sword. On one hand, it doesn't require any new headers, thus L2 frames can pass through L2 equipments that are not QoSAL-aware, albeit only with best-effort service. Moreover, the Flow Label is a simple 20-bit integer field, making it a relatively simple operation to map packets to flows using this field[2]. On the other hand, this solution does limit the QoSAL to IPv6, which is both conceptually wrong and in practice may constitute a problem, since L2 equipments do not normally have an IP stack, or if they do it is used only for management; in data plane, APs only look at IEEE 802 headers.

So far it has only been explained how packets with a valid, non-zero Flow Label are treated. What about best effort? Well, in fact, in DAIDALOS even the best effort flows need QoS resources reserved. Either applications are DAIDALOS-aware and make the QoS reservation explicitly at the terminal, or they are legacy applications, in which case they just start transmitting right away. However, as soon as these packets reach the AR, they are intercepted by the ARM, and a suitable best effort reservation is triggered. For this to work, however, it is necessary that L2 QoS equipments reserve just a bit of residual bandwidth for unmarked packets, so that they can at least reach the AR and trigger a proper reservation. Moreover, some bandwidth must also be pre-allocated for for signalling. Signalling packets, in Daidalos, are identified by a set of values marked into DSCP field of packets. Table 4.2 summarises the rules for classifying packets in the data plane, and how to treat packets of each type. The mapping rules may appear too complicated, considering the need to inspect both Flow Label and DSCP fields, which might degrade performance. But one has to consider the fact that the majority of packets have a valid QoSAL reservation, and for these packets looking at the Flow Label is enough to determine how to treat them. Only the remaining packets need more careful inspection, but they are so few that this doesn't have any significant impact on the overall performance.

4.2.2 Service Interface

The services offered by the QoS Abstraction Layer to layer 3, i.e. to the QoS Manager in Fig. 4.1, can be classified into five groups: 1) QoS reservation; 2) resource querying; 3) QoS degradation notification; 4) mobility; 5) multicast.

[2] Especially when compared to the alternative of matching by the tuple (source-ipv6, destination-ipv6, protocol, source-port, destination-port), totalling 38 bytes, 4 of which (the ports) are not directly accessible using a constant offset.

Table 4.2: Data plane QoS mapping rules

Flow Label	**DSCP**	**Packet Type / Treatment**
$\neq 0$	Don't care	Application flow with reservation — apply QoS of connection identified by $CnxID = FlowLabel$
$= 0$	$\neq signalling$	Application flow temporarily without reservation — some residual bandwidth reserved for this.
$= 0$	$= signalling$	Signalling packet — some residual bandwidth reserved for this.

Data Types

The description of the parameter types involved follows:

Addr: Can be either an L3Addr or an L2Addr;

L3Addr: An IPv6 address. In fact, this type is just an alias for `struct sockaddr_in6` in the interface implementation;

L2Addr: An IEEE 802 MAC address, in case of a remote AP scenario, or a generic link-layer address (e.g. UMTS IMEI), in case of a single (non-concatenated) locally attached wireless interface. In fact, this type is just an alias for `struct sockaddr_ll` in the interface implementation;

Result: Enumeration value indicating the status of completion of an operation. Can be either ACCEPT or REJECT, and indicates whether the Abstraction Layer accepts or not what is proposed in the primitive;

CnxID This is an unsigned 20-bit integer number that identifies a given abstraction layer connection. It is unique only within a given Link Access Network;

TSpec: This parameter includes a set of attributes that characterise the application flow that is to be transported in the QoSAL connection. Parameters include:

- Peak Bitrate (p)
- Average Bitrate (r)
- Maximum Burst Size (b)
- Maximum Transmission Unit (M)
- Minimum Policed Unit (m)

They have the same meaning as described in [SW97];

RSpec: This parameter includes a set of attributes that define the service that Layer 2 promises to deliver, as long as application traffic does not violate the Tspec. These attributes include Class Identifier and Reserved Bitrate, as described in Sec. 4.2.1. Other parameters, such as BER and delay, are determined implicitly based on the Class Identifier;

Bitrate: Transmitted information per time unit, in bits per second;

ContextInfo: A block of binary data that contains some information not used by the AL but associated with the connection. It is transmitted (piggy-backed in QoSAL Protocol Data Units (PDUs)) from AR to MN, and from there passed to the upper layers;

BER: Bit Error Ratio, defined as the ratio between erroneous bits and total number of bits in a frame/packet; it is represented as a single precision floating-point number. A negative value means that BER information is not available;

Priority: An integer number that represents a priority, from -128 (lowest) to $+127$ (highest);

Boolean: A boolean value, either True or False.

QoS Reservation

The main service offered by the QoS Abstraction Layer consists in the creation of *QoS connections* between the AR and a MN. These QoS connections can be described as virtual channels between the two elements, which offer certain QoS guarantees, such as bitrate and delay. There are primitives to create, modify, and release QoS connections.

Connection activation The primitives `AL-CNX-ACTIVATE-REQ` and `AL-CNX-ACTIVATE-RESP` can be used to negotiate the establishment, or *activation*, of a new QoS connection.

The `AL-CNX-ACTIVATE-REQ` primitive triggers the signalling described in Sec. 4.2.3, and returns a *connection identifier* in an `AL-CNX-ACTIVATE-RESP` primitive if the reservation succeeds. The entity that requested QoS reservation is then responsible to setup the flow marking module that will mark the Flow Label of the packets belonging to a reservation with the associated connection identifier. Moreover, the connection identifier is used for modification and deactivation of QoS connections, and it is included in QoS degradation notifications.

When a connection is activated, the destination end point is notified of this using the primitive `AL-CNX-INDICATION`. The main purpose of this primitive is to notify QoS degradation, and is described in detail further below, but is also reused for notification of connection activation, as exemplified in Fig. 4.2. The ctx_info parameter of this primitive plays an important role here. The same ContextInfo parameter that is supplied by the QoS Manager at the AR is passed into the QoS Client as an `AL-CNX-INDICATION` parameter at the MN. Although an opaque binary data block from the point of view of

Table 4.3: Specification of the `AL-CNX-ACTIVATE-REQ` service interface primitive

Primitive name:	`AL-CNX-ACTIVATE-REQ`
Description:	Requests activation of a new QoS connection.
Direction:	Downcall

Type	**Name**	**Description**
Addr	dest_addr	Destination address of the MN for which we wish to make a reservation. Note: generally, dest_addr should be of type L3Addr (`sockaddr_in6`); however, multicast reservations should be indicated with a dest_addr of type L2Addr, which is actually a `struct sockaddr_ll` with the member `sll_pkttype` set to the value PACKET_MULTICAST. More information about multicast in Sec. 4.2.2.
Tspec	tx_tspec	Traffic Specification for the transmitted flow
Rspec	tx_rspec	Reservation Specification for the transmitted flow
Tspec	rx_tspec	Traffic Specification for the received flow
Rspec	rx_rspec	Reservation Specification for the received flow
Priority	retention_prio	Retention priority; the lower the priority, the more likely it will be for this connection to be selected for QoS degradation when resources become scarce
Boolean	handoff	Can be True to support mobility scenarios, as explained in Sec. 4.2.2, otherwise it should be set to False.
ContextInfo	ctx_info	Context information to attach to the connection

Table 4.4: Specification of the `AL-CNX-ACTIVATE-RESP` service interface primitive

Primitive name:	`AL-CNX-ACTIVATE-RESP`
Description:	Result of a QoS connection activation request.
Direction:	Upcall

Type	**Name**	**Description**
CnxID	cnx_id	Identifier for newly created connection, in case of successful activation
Result	result	Indicates if reservation was successful or not

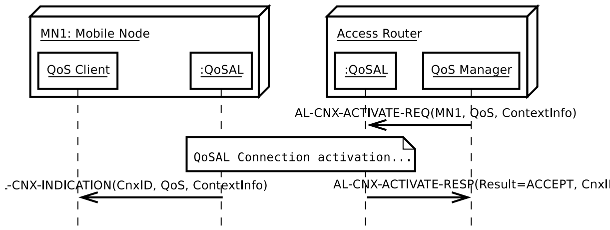


Figure 4.2: Service interface message exchanges during a successful connection activation.

the QoSAL, this data block actually is used to send L3 QoS information, such as DSCP, and source/destination addresses and ports, from the AR to the MN. This information is used in the MN, for example to setup flow label and DSCP flow marking in the uplink direction.

Two more aspects are worth mentioning. The first one is regarding the retention_prio parameter. The existence of this parameter is justified by the frequently changing conditions in wireless networks as terminals move towards or away from the APs. When the QoSAL detects that QoS resources available at some point become insufficient to keep the QoS level promised on the QoSAL connections, it is forced to degrade or tear down some of connections in order to free up resources for the remaining ones. Although the QoS Broker knows best the relative priorities of all connections in accordance with the users' profiles, the variations often happen very fast and local decisions are taken by the QoSAL drivers in each AP, in order to react in time as anomalies are detected. To this end, the retention_prio parameter is used by the QoS Manager when activating a connection to indicate which should be the relative priority of such connection for purposes of QoS degradation. QoSAL connections with lower priorities are more likely to be selected for degradation.

Also in need of some clarification are the QoS parameters tx_tspec, tx_rspec, rx_tspec, and rx_rspec. The meaning of the tx_ and rx_ prefixes are as follows. The tx_ parameters refer to the QoS that is to be reserved for *transmitted* flows, while the rx_ parameters refer to the QoS that is to be reserved for *received* flows. In this context, transmitted/received is always from the point of view of the entity requesting the connection activation. From the point of view of the QoS Manager, *transmitted* means *downlink*, while *received* means *uplink*.

Connection modification The primitives `AL-CNX-MODIFY-REQ` and `AL-CNX-MODIFY-RESP` can be used to modify the QoS parameters for an already established QoS connection. The usage of these primitives is similar to the connection activation scenario, i.e. first the caller sends a `AL-CNX-MODIFY-REQ`, containing the CnxID of the connection that is to be modified, and the new QoS parameters, and the QoSAL calls back later after the modification is complete, sending an `AL-CNX-MODIFY-RESP` containing the CnxID and the result of the modification. It worth noting, however, that if a modification fails it only means that the connection reverts to the same QoS settings before the modification

Table 4.5: Specification of the `AL-CNX-MODIFY-REQ` service interface primitive

Primitive name:	`AL-CNX-MODIFY-REQ`
Description:	Requests modification of the QoS parameters of an existing connection
Direction:	Downcall

Type	**Name**	**Description**
CnxID	cnxid	The connection identifier for the connection
Tspec	tx_tspec	New Traffic Specification for the transmitted flow
Rspec	tx_rspec	New Reservation Specification for the transmitted flow
Tspec	rx_tspec	New Traffic Specification for the received flow
Rspec	rx_rspec	New Reservation Specification for the received flow

Table 4.6: Specification of the `AL-CNX-MODIFY-RESP` service interface primitive

Primitive name:	`AL-CNX-MODIFY-RESP`
Description:	Result of a QoS connection modification request
Direction:	Upcall

Type	**Name**	**Description**
CnxID	cnxid	The connection identifier for the connection for which a modification has been requested
Result	result	Indicates if modification was successful or not

being attempted. A modification to decrease the QoS levels always succeeds[3].

Connection deactivation The primitive `AL-CNX-DEACTIVATE` can be used to deactivate an already established QoS connection, freeing its QoS resources along with it. There is no response primitive, since it always succeeds. When a connection is deactivated, the destination endpoint (i.e. the MN) is notified with an identical `AL-CNX-DEACTIVATE` primitive, as show in Fig. 4.3.

Resource Querying

There is a primitive to request a report of the available (free) resources in the access network, `AL-RESOURCE-QUERY`. Its only parameter, `DestAddr`, identifies the "path" for which resources are to be queried. `DestAddr` can be an L2 address of a MN, or an AP, with the following meaning:

[3]Unless, of course, there's a race condition when a modification is requested and at the same time the wireless conditions change in a way to make both the previous and new QoS levels impossible to maintain.

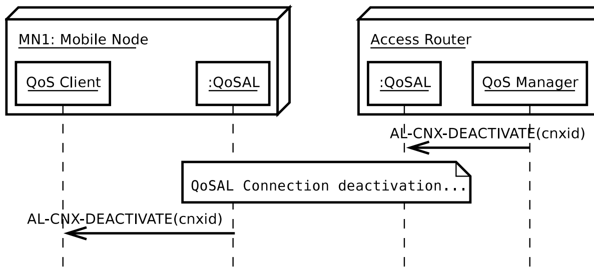


Figure 4.3: Service interface message exchanges during a connection deactivation.

Table 4.7: Specification of the `AL-CNX-DEACTIVATE` service interface primitive

Primitive name:	`AL-CNX-DEACTIVATE`
Description:	Requests deactivation of an existing connection.
Direction:	Downcall/Upcall

Type	**Name**	**Description**
CnxID	cnxid	The connection identifier for the connection

MN Request a single report for all APs in the path towards the given MN;

AP Request a single report for all APs in the path towards the given AP, including the destination AP itself;

The resources are reported via the upcall primitive `AL-RESOURCE-INDICATION` after some delay. Currently, the only reported parameter is:

Bandwidth An *estimation* of free bandwidth in the AP. If the path includes multiple APs, the returned bandwidth is the minimum of all bandwidths of individual APs;

`AL-RESOURCE-INDICATION` can also be sent spontaneously by the QoSAL, either periodically or whenever significant changes in the available resources are detected.

The `AL-RESOURCE-QUERY` primitive is used by the QoS Manager to check L2 admission control before proceeding with end-to-end QoS reservations. Moreover, it is used to provide QoS information to the Performance Manager, thus allowing a more efficient load balancing of mobile terminals among different APs/ARs.

Table 4.8: Specification of the `AL-RESOURCE-QUERY` service interface primitive

Primitive name:	`AL-RESOURCE-QUERY`
Description:	Request information about overall resources available in a given L2 network path.
Direction:	Downcall

Type	**Name**	**Description**
Addr	Target	Address of the target that defines a path. Target can be a MN, an AP, or a broadcast address.

Table 4.9: Specification of the `AL-RESOURCE-INDICATION` service interface primitive

Primitive name:	`AL-RESOURCE-INDICATION`
Description:	Report information about overall resources available in a given L2 network path.
Direction:	Upcall

Type	Name	Description
L2Addr	Target	Target that identifies the path that this report refers to.
Bitrate	Bandwidth	Estimation of available bandwidth

Table 4.10: Specification of the `AL-RESOURCE-DEGRADATION` service interface primitive

Primitive name:	`AL-RESOURCE-DEGRADATION`
Description:	Notification of degradation of overall resources.
Direction:	Upcall

Type	Name	Description
list<(L2Addr, Bitrate)>	resources	List of AP address/bandwidth pairs.

The primitive AL-RESOURCE-DEGRADATION is automatically sent to the Performance Attendant module (a client that connects to a special local socket) whenever the available bandwidth in any one of the APs visible by the AR drops below 25% of the respective technology theoretical maximum. When this primitive is received by the PA, the Performance Manager is notified, and a Network Initiated Handover procedure may be triggered this way.

QoS Degradation Notifications

The primitive `AL-CNX-INDICATION` is issued from the AL at both AR and MN to indicate that the AL was forced to modify the QoS for a specific connection due to changing conditions in the wireless medium. The primitive includes the connection identifier and Rspec as parameters. This primitive is important, for it allows link adaptation by applications, and may serve as trigger for higher-level handover decisions.

During the lifetime of a connection, the QoSAL is allowed to spontaneously modify the reserved bitrate within the bounds of the initial reservation, as shown in Fig. 4.4. In this example, the QoSAL senses a degradation in the signal strength, and temporarily switches the transmission mode to one with lower bitrate but lower error probability. Since the actual bitrate obtainable by the connection has decreased, an `AL-CNX-INDICATION` primitive is issued at MN an AR side to notify the QoS Manager and/or applications of this. When the effect that caused the degradation ceases, the link is again gradually switched to higher bitrate modes, and new QoS levels become possible in the connections; in this case `AL-CNX-INDICATION` primitives are sent again, until the initial QoS level is attained.

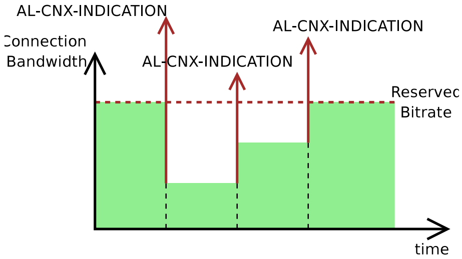


Figure 4.4: Example of QoS degradation notifications

Table 4.11: Specification of the `AL-CNX-INDICATION` service interface primitive

Primitive name:	`AL-CNX-INDICATION`
Description:	Notification of new connection activated or QoS changing on a connection.
Direction:	Upcall

Type	**Name**	**Description**
CnxID	cnx_id	Identifier for connection that has changed
Tspec	tx_tspec	New Traffic Specification for the transmitted flow
Rspec	tx_rspec	New Reservation Specification for the transmitted flow
Tspec	rx_tspec	New Traffic Specification for the received flow
RSpec	rx_rspec	New Reservation Specification for the received flow
BER	ber	Optional BER information
ContextInfo	ctxt_info	Context information attached to the connection

As explained, `AL-CNX-INDICATION` is also reused as a means to notify the QoS Client of new connections, hence the presence of the ContextInfo parameter in this primitive. However, the ContextInfo is always empty except during the activation notification.

Mobility

The service interface of the QoSAL has explicit support for mobility scenarios through the *handoff* parameter of `AL-CNX-ACTIVATE-REQ` and the MN-side primitive `AL-HANDOVER-EXECUTE`.

At the beginning of an impending handover, the QoS Manager in the new AR cannot do a regular connection activation. That is because the MN is not yet connected to the new AR/AP, therefore a message cannot be sent to it, and the QoS Manager does not even have knowledge of which will be the AP the MN will connect through. Thus, as shown in Fig. 4.5, the QoS Manager sends an `AL-CNX-ACTIVATE-REQ` with the *handoff* parameter set to True. This parameter value has the effect of postponing the QoSAL connection activation until the instant MN connects to the new AR/AP. This instant is signalled by the primitive `AL-HANDOVER-EXECUTE` at the MN, which triggers an equivalent PDU that is sent to the AR, causing the QoSAL to proceed with a regular activation.

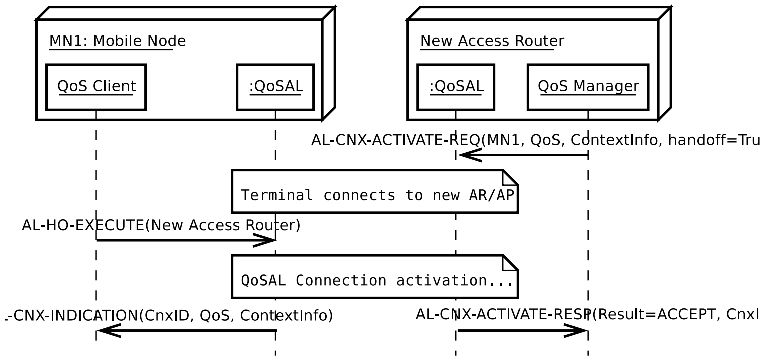


Figure 4.5: Example of a handover using QoSAL interfaces

Table 4.12: Specification of the `AL-HO-EXECUTE` service interface primitive

Primitive name:	`AL-HO-EXECUTE`
Description:	Handover just occurred; proceed with any pending connection activation(s).
Direction:	Downcall (MN side only)

Type	**Name**	**Description**
Addr	dst	Address of the new AR the MN has just connected to

Multicast

The support of multicast QoS will be important in future networks, as multimedia streaming services are likely to gain prominence. Therefore, multicast support had also to be designed into the QoSAL.

There are two stages required to obtain multicast QoS. First, a QoS reservation is performed, using the already described `AL-CNX-ACTIVATE-REQ` primitive, but specifying an L2 multicast address instead of a MN unicast address. As usual, a connection identifier is returned, which identifies the *multicast session*. After a multicast session is created, membership management takes place. The primitive `AL-MULTICAST-JOIN` takes a connection identifier and an L2 address as parameters, and it is used to notify the QoSAL that a new MN is joining the indicated multicast session. Conversely `AL-MULTICAST-LEAVE` is used to indicate that a MN is leaving a multicast session. Both primitives are part of the service provided by the QoSAL to the QoS Manager at AR.

4.2.3 Protocol

After having seen a description of the service offered by the QoSAL to upper layers, it is time to learn how QoSAL protocol conveys the requests from the service interface to remotely located APs, and brings back resource information and QoS degradation notifications from the same remote APs, while at the same time satisfying the requirements listed in Sec. 4.1.

Table 4.13: Specification of the `AL-MULTICAST-JOIN` service interface primitive

Primitive name:	`AL-MULTICAST-JOIN`
Description:	Indicate that a terminal is joining a multicast session
Direction:	Downcall

Type	**Name**	**Description**
Addr	mn_addr	Address of the MN that is joining the multicast session
CnxID	session	Connection identifier of the multicast session

Table 4.14: Specification of the `AL-MULTICAST-LEAVE` service interface primitive

Primitive name:	`AL-MULTICAST-LEAVE`
Description:	Indicate that a terminal is leaving a multicast session
Direction:	Downcall

Type	**Name**	**Description**
Addr	mn_addr	Address of the MN that is leaving the multicast session
CnxID	session	Connection identifier of the multicast session

The QoSAL PDUs are transmitted as IEEE 802 frames[4] with a dedicated protocol type. The decision not to use the IP protocol to transport PDUs is motivated by several factors, such as:

1. The QoSAL is meant to be a Layer 2.5 protocol;

2. A decision of using the IP layer would bring up a difficult decision of which IP version to use, 4 or 6;

3. The IP layer represents an extra overhead in terms of PDU size that simply is not compensated by any significant feature added by it and that is needed by the QoSAL;

4. The QoSAL protocol has to be implemented in all QoS-capable L2 nodes, such as APs, and L2 nodes, by definition, do not necessarily have a third layer.

The primitives of the service interface are closely related to the PDUs. In fact, for most of these primitives, such as `AL-CNX-ACTIVATE-REQ` and `AL-CNX-ACTIVATE-RESP`, there is a PDU corresponding to each primitive. The protocol borrows some ideas from the RSVP protocol [BZB+97], namely in-band signalling and soft-state. In-band signalling means that the PDUs use the same channels and paths as regular application packets. In other words, there is no "management network" overlaid to the access network to control the APs. It is also soft-state because QoS reservations have to be periodically refreshed, else they expire.

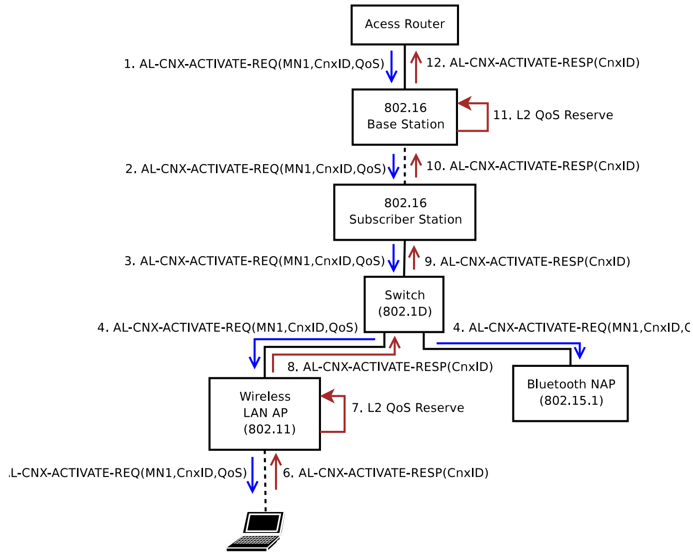


Figure 4.6: QoS connection activation example

Connection Activation, Modification, and Deactivation

Fig. 4.6 illustrates how the `AL-CNX-ACTIVATE-REQ` primitive is realized by the protocol. First, a unique identifier for the QoS connection, CnxID, is derived. Then, an Ethernet frame is sent, with MN1 as destination MAC address. The frame contains an `AL-CNX-ACTIVATE-REQ` PDU, with CnxID and QoS as parameters. It is important to emphasise that by putting the MN address as destination address of the signalling frames, the design goals in Sec. 4.1.3 are met. This comes "for free" with IEEE 802.1D Learning Bridges [Tel98], implemented by Switches and wireless APs. Thus, the request PDU follows the path towards the MN, traversing a series of APs. However, QoSAL-enabled APs automatically recognise the AL PDU by its Ethernet protocol number, and pass it to the QoSAL code for special processing, before allowing it to be forwarded.

No reservation is performed at this point, though. The AP has to wait for an `AL-CNX-ACTIVATE-RESP` primitive, at which point the reservation is committed, in case of a successful response from downstream. The reason why the confirmation is required before committing the reservation has to do with the Learning Bridge. A Learning Bridge is not programmed with any routes; it learns the routes from the traffic it receives, and maintains a cached table of MAC address / output port. This property is both a strength, since it allows it to work with no previous configuration of routes, and a weakness, because it is not guaranteed to always know the route for a particular destination. When a Learning Bridge does not know the route for a particular MAC address, it simply forwards the frame through all output ports, as exemplified in Fig. 4.6. Thus, it is possible that a given QoSAL instance may receive a PDU "by accident", and forcing it to wait for a response from the MN before acting on the request prevents this problem.

[4]Although other L2 networks can be supported with little additional effort

These "mistakes" happen relatively infrequently, but we need to cope with them in the rare cases when they happen.

There is an analogy between `AL-CNX-ACTIVATE-REQ` and RSVP's *Path* messages, and between `AL-CNX-ACTIVATE-RESP` and RSVP's *Resv*. Like RSVP, the AL protocol is soft-state, which means that AL QoS connections must be periodically refreshed, or else they expire. When a QoSAL connection needs to be refreshed, a new `AL-CNX-ACTIVATE-REQ` PDU is sent from the AR, with the same parameters as in the initial activation. This time, however, the QoSAL instances located in intermediate APs receive the PDU but recognise the CnxID as belonging to an already known connection, and they "refresh" the connection state by restarting an internal timeout that would cause that connection to expire after some time.

At the same time, the QoSAL compares the QoS parameters in the `AL-CNX-ACTIVATE-REQ` PDU with the parameters in the local connection state. If the parameters are different, a QoS modification primitive is sent to the driver(s) as soon as a corresponding `AL-CNX-ACTIVATE-RESP` arrives. In this way, there is no need to have `AL-CNX-MODIFY-REQ` and `AL-CNX-MODIFY-RESP` PDUs, therefore significantly simplifying the protocol.

Figure 4.7 might help understand better how the QoS protocol works with respect to QoSAL connections. Each connection is represented by an object modelled by a state machine represented in this diagram. It follows UML 2.0 [PP05] *Protocol State Machine* notation, which is very similar to SDL. The "connection" object is initially created when a `AL-CNX-ACTIVATE-REQ` PDU is received containing an unknown CnxID. This is represented by the transition from the filled circle near the bottom-right corner of the diagram. When the object is created it registers itself in a global CnxID→Connection table, so that any future "connection bound" messages (`AL-CNX-`*Something*) are directed to this object. Upon initialisation, a timer is created, and when it expires the connection object is destroyed. However, each time a `AL-CNX-ACTIVATE-RESP` is received the timer is restarted. Thus, as long as `AL-CNX-ACTIVATE-RESP` PDUs are received regularly, the connection stays alive. A special object (not represented in the diagram) called "connection initiator" is present in the QoSAL in the AR, which assumes the responsibility of sending `AL-CNX-ACTIVATE-REQ` PDUs at regular intervals, which are forwarded down the L2 tree, causing `AL-CNX-ACTIVATE-RESP`'s to be transmitted following the reverse path, and refresh the connection states in each L2 node. This is how the "soft-state" property of the protocol is realized. Also represented in the state machine is the way that changing QoS parameters in a `AL-CNX-ACTIVATE-REQ` trigger a modification of the local QoS resources. Finally, a `AL-CNX-DEACTIVATE` primitive unconditionally causes the connection object be destroyed.

Resource Query

Regarding the resource query primitive described in Sec. 4.2.2, the two reservation styles are simply mapped into MN, or AP addresses. Whatever the addressing scheme used, a `AL-RESOURCE-QUERY` PDU is sent from the AR towards the requested destination. When this PDU reaches either the destination or an edge AP, the forwarding of this message stops, and a response is sent back to original requester. The response consists

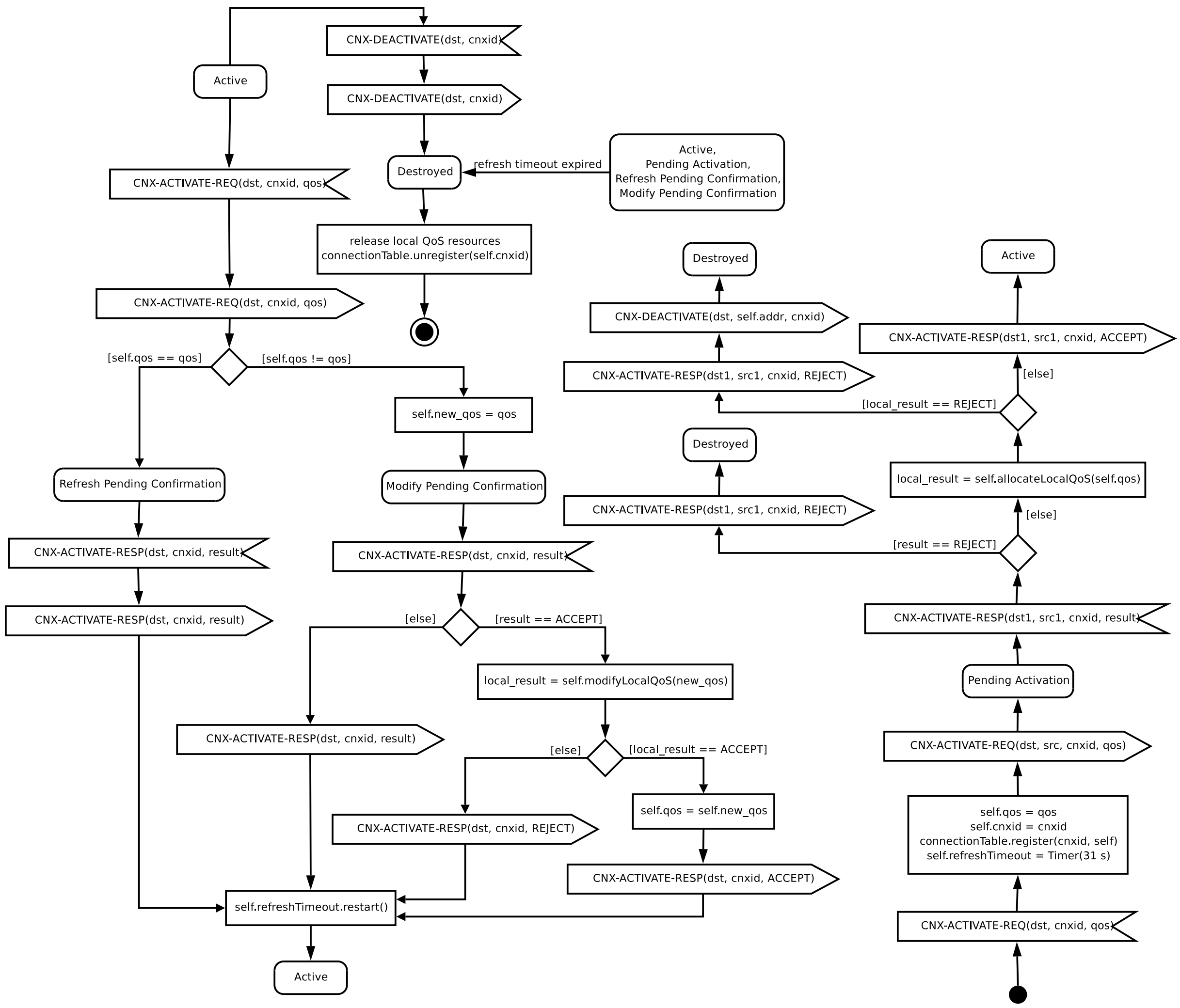


Figure 4.7: Protocol state machine of a QoSAL connection object (UML 2.0 notation)

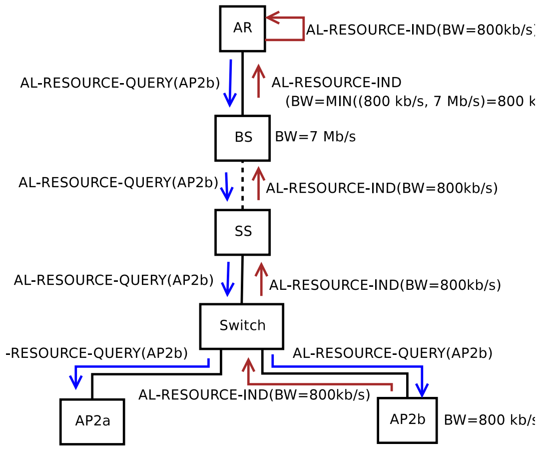


Figure 4.8: Example of a resource query with an AP as target.

of a `AL-RESOURCE-INDICATION` PDU containing the minimum bandwidth available. The minimum bandwidth is initialised to a special *infinite* value, and as the message passes through the APs, each AP combines its own available bandwidth with the value found in the message. The overall available bandwidth reported to the AR is, therefore, equal to the minimum of all the bandwidths in each AP. For example, in Fig. 4.8 the IEEE 802.16 Base Station (BS) has 7 Mbit/s of available bandwidth, but only 800 kbit/s are free in AP2b, and that is the minimum value that is reported to the AR.

Announcement/Discovery

One of the requirements, in particular from the Performance Agent module, is the ability of an AR to automatically discover APs reachable from any of the interfaces. This is so that the AR can query each AP that is discovered, to receive bandwidth reports. Also because the PA doesn't *know* which APs are available.

There is a PDU called `AL-ANNOUNCE` that is used by QoSAL instances running in APs to announce their presence. These messages are periodically broadcast by APs and received by ARs. This PDU contains as parameter a list of *intermediate nodes*, in addition to the usual source and destination addresses. The intermediate nodes list contains the L2 addresses of all QoSAL nodes besides the one that created the message, and it is used to discover the full L2 topology in the case of concatenated APs. This way, the AR, who receives these announcements, discovers not only the existence of APs, but also a list of intermediate APs in the path to each edge AP, and through which local interface it is reachable.

The announcement messages are periodically broadcast by APs, and a L2 topology database is kept at ARs. The information for each AP is soft-state, and expires some time after announcements ceasing to be received. This way, the AR also finds out when an AP is disconnected.

Table 4.15: List of PDUs used in the QoSAL protocol

PDU	Values
`AL-CNX-ACTIVATE-REQ`	dst, src, cnxid, tx_tspec, tx_rspec, rx_tspec, rx_rspec, retention_prio, ctx_info
`AL-CNX-ACTIVATE-RESP`	dst, src, cnx_id, result
`AL-CNX-DEACTIVATE`	dst, src, cnx_id
`AL-CNX-INDICATION`	dst, src, cnxid, tx_tspec, tx_rspec, rx_tspec, rx_rspec, ber, ctx_info
`AL-RESOURCE-QUERY`	dst, src
`AL-RESOURCE-INDICATION`	dst, src, bandwidth
`AL-ANNOUNCE`	dst, src, aplist

Summary

Table 4.15 summarises the PDUs that are used in the QoSAL protocol. The parameters *dst* and *src* are always present, since they are the destination and source (MAC) addresses of the message. At this point, there are no PDU equivalents for `AL-MULTICAST-JOIN` and `AL-MULTICAST-LEAVE`, meaning that multicast group membership is not (yet) implemented for concatenated networks.

4.2.4 Driver Interface

Registration

The interface between Abstraction Layer and driver is generic and independent of location, be it MN, AP, or AR. The Abstraction Layer is a process that runs in each node. A QoSAL driver is another process that may run in any node, and register interest in implementing QoS for a given network interface.

The primitive `AL-DRIVER-REGISTER` must be issued by the driver immediately after connecting to the QoSAL. It declares intention of the driver to handle QoS for a specific network interface. A driver may wish to handle several network interfaces. To do so, it must create several connections (sockets) to the QoSAL, one for each network device. Each connection represents one driver instance from the point of view of the QoSAL, even though they may all come from the same process. Each driver instance handles exactly one network interface.

QoS Reservation

Connection activation The primitives `AL-CNX-ACTIVATE-REQ` and `AL-CNX-ACTIVATE-RESP` are the driver equivalents to the corresponding service interface primitives, which can be used to negotiate the establishment, or *activation*, of a new QoS connection.

Table 4.16: Specification of the `AL-DRIVER-REGISTER` driver interface primitive

Primitive name:	`AL-DRIVER-REGISTER`
Description:	Register driver with the main QoSAL process.
Direction:	Upcall

Type	**Name**	**Description**
String	netdevice	Network device identifier that the driver wishes to handle
Bitrate	bwmax	Maximum theoretical bandwidth supported by the technology

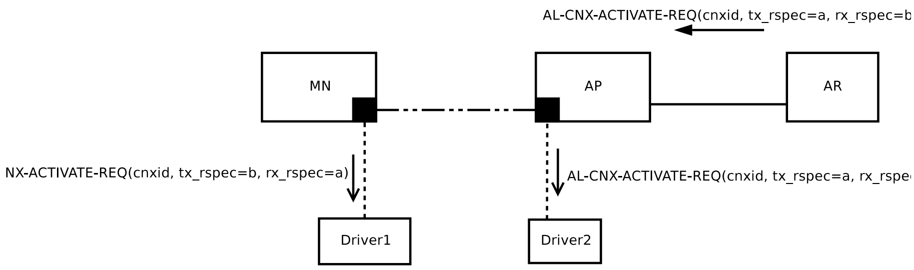


Figure 4.9: Example of how tx/rx QoS parameters can change in the driver interface.

These primitives are similar to the service interface primitives, except for a few minor differences. For instance, only L2 addresses are supported here, not L3 ones. Moreover, there are two kinds of connection identifiers at stake here. There is a *global identifier*, which is the one used in the service interface and marked into packets' Flow Label. Then there is a *local identifier*, which is returned by the driver and used to uniquely identify a QoSAL connection in the QoSAL–Driver interface, but has no global meaning. Also the ContextInfo parameter is not present in the driver interface, since it is not needed by the driver. Finally, it should be noted that the tx/rx parameters retain their meaning from the service interface, although depending on which side of a wireless link a driver is registered to handle tx may be considered uplink or downlink. For example, in Fig. 4.9 Driver1 receives the tx/rx parameters swapped relative to the parameters requested in the service interface at the AR; from the point of view of the AR *transmission* means *downlink* and *reception* means *uplink*, while from the point of view of Driver1 *downlink* traffic is *received* by the network interface it registers to handle, while *uplink* traffic is *transmitted* through this same interface, hence the need to swap the values.

Connection modification The primitives `AL-CNX-MODIFY-REQ` and `AL-CNX-MODIFY-RESP` can be used to modify the QoS parameters for an already establishment QoS connection, just like in the service interface.

Connection deactivation The primitive `AL-CNX-DEACTIVATE` can be used to deactivate an already established QoS connection, freeing its QoS resources along with it.

Table 4.17: Specification of the AL-CNX-ACTIVATE-REQ driver interface primitive

Primitive name:	AL-CNX-ACTIVATE-REQ
Description:	Requests activation of a new QoS connection
Direction:	Downcall

Type	**Name**	**Description**
L2Addr	dest_addr	Destination address of the destination MN for the reservation. This parameter may be safely ignored if the driver doesn't require it. Multicast reservations are indicated with a dest_addr of type L2Addr, which is actually a `struct sockaddr_ll` with the member `sll_pkttype` set to the value PACKET_MULTICAST.
CnxID	cnxid	Global connection identifier. The driver must configure L2 to map packets into the connection being activated by comparing the Flow Label IPv6 header to the global connection identifier.
Tspec	tx_tspec	Traffic Specification for the transmitted flow
Tspec	tx_tspec	Traffic Specification for the transmitted flow
Rspec	tx_rspec	Reservation Specification for the transmitted flow
Tspec	rx_tspec	Traffic Specification for the received flow
Rspec	rx_rspec	Reservation Specification for the received flow
Priority	retention_prio	Retention priority; the lower the priority, the more likely it will be for this connection to be selected for QoS degradation when resources become scarce

Table 4.18: Specification of the `AL-CNX-ACTIVATE-RESP` driver interface primitive

Primitive name:	`AL-CNX-ACTIVATE-RESP`
Description:	Result of a QoS connection activation request.
Direction:	Upcall

Type	**Name**	**Description**
Result	result	Indicates if reservation was successful or not
CnxID	cnx_id	Identifier for newly created connection. This identifier is generally different from the global connection identifier. It only has to uniquely identify a QoS connection for in the specific AL-Driver interface.

Table 4.19: Specification of the `AL-CNX-MODIFY-REQ` driver interface primitive

Primitive name:	`AL-CNX-MODIFY-REQ`
Description:	Requests modification of the QoS parameters of an existing connection.
Direction:	Downcall

Type	**Name**	**Description**
CnxID	cnxid	The (local) connection identifier for the connection
Tspec	tx_tspec	New Traffic Specification for the transmitted flow
Rspec	tx_rspec	New Reservation Specification for the transmitted flow
Tspec	rx_tspec	New Traffic Specification for the received flow
Rspec	rx_rspec	New Reservation Specification for the received flow

Table 4.20: Specification of the `AL-CNX-MODIFY-RESP` driver interface primitive

Primitive name:	`AL-CNX-MODIFY-RESP`
Description:	Result of a QoS connection modification request.
Direction:	Upcall

Type	**Name**	**Description**
CnxID	cnxid	The (local) connection identifier for the connection for which a modification has been requested
Result	result	Indicates if modification was successful or not

Table 4.21: Specification of the `AL-CNX-DEACTIVATE` driver interface primitive

Primitive name:	`AL-CNX-DEACTIVATE`
Description:	Requests deactivation of an existing connection.
Direction:	Downcall/Upcall

Type	Name	Description
CnxID	cnxid	The (local) connection identifier for the connection

Table 4.22: Specification of the `AL-RESOURCE-QUERY` driver interface primitive

Primitive name:	`AL-RESOURCE-QUERY`
Description:	Request information about overall resources available in the network interface.
Direction:	Downcall

Type	Name	Description
—	—	—

There is no response primitive, since it always succeeds.

Resource Querying

The primitive `AL-RESOURCE-QUERY` is used to request a report of the available (free) resources in the link attached to the network interface handled by the driver. It takes no parameters, but has the following side-effects:

1. Causes the driver to send back an `AL-RESOURCE-INDICATION` as soon as possible;
2. The driver assumes responsibility of, from there on, spontaneously sending `AL-RESOURCE-INDICATION` when changes in the available resources take place.

It should be noted that a driver only has to report *local* resources in the L2 segment directly controlled by it. It is the QoSAL that computes the overall resources by combining the reports from individual drivers.

QoS Degradation Notifications

The driver primitive `AL-CNX-INDICATION` is directly responsible for the primitive with the same name in the service interface. It is spontaneously sent by the driver when changes

Table 4.23: Specification of the `AL-RESOURCE-INDICATION` driver interface primitive

Primitive name:	`AL-RESOURCE-INDICATION`
Description:	Report information about overall resources available in the network interface.
Direction:	Upcall

Type	Name	Description
Bitrate	Bandwidth	Estimation of available bandwidth

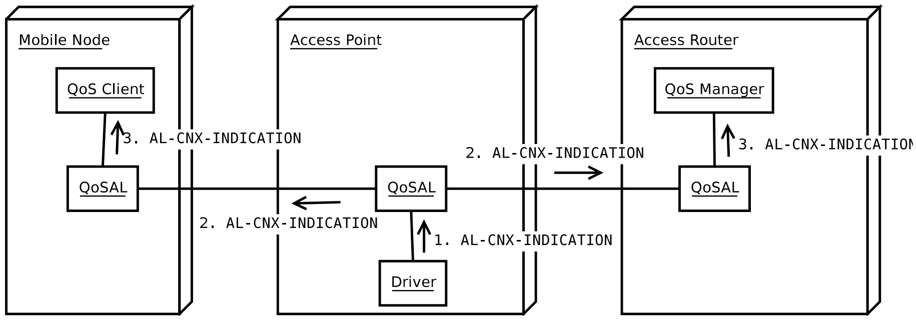


Figure 4.10: Example of an AL-CNX-INDICATION primitive being emitted by the driver.

Table 4.24: Specification of the `AL-CNX-INDICATION` driver interface primitive

Primitive name:	`AL-CNX-INDICATION`
Description:	Notification of changing QoS on a connection.
Direction:	Upcall

Type	**Name**	**Description**
CnxID	cnx_id	Identifier (local) for the connection that has changed
Tspec	tx_tspec	New Traffic Specification for the transmitted flow
Rspec	tx_rspec	New Reservation Specification for the transmitted flow
Tspec	rx_tspec	New Traffic Specification for the received flow
RSpec	rx_rspec	New Reservation Specification for the received flow
BER	ber	Optional BER information

in the wireless medium force the QoS contract to be modified. The new QoS values possible to attain are therefore advertised using this primitive. These advertisements are propagated in both directions from the driver: uplink towards the AR/QoS Manager, and downlink towards the MN/QoS Client, as shown in Fig. 4.10.

Multicast

The primitives `AL-MULTICAST-JOIN` and `AL-MULTICAST-LEAVE` match the similarly named service interface primitives, allowing the QoSAL to inform the driver about multicast membership events.

4.2.5 Some notes on event compression

The Abstraction Layer defines two upcall primitives to deliver resource notifications:

1. `AL-CNX-INDICATION` indicates when the resources allocated to a QoS connection can no longer be provided; It contains, among other data, the new bandwidth available to the connection;

Table 4.25: Specification of the `AL-MULTICAST-JOIN` driver interface primitive

Primitive name:	`AL-MULTICAST-JOIN`
Description:	Indicate that a terminal is joining a multicast session
Direction:	Downcall

Type	Name	Description
L2Addr	mn_addr	Address of the MN that is joining the multicast session
CnxID	session	Connection identifier of the multicast session

Table 4.26: Specification of the `AL-MULTICAST-LEAVE` driver interface primitive

Primitive name:	`AL-MULTICAST-LEAVE`
Description:	Indicate that a terminal is leaving a multicast session
Direction:	Downcall

Type	Name	Description
L2Addr	mn_addr	Address of the MN that is leaving the multicast session
CnxID	session	Connection identifier of the multicast session

2. `AL-RESOURCE-INDICATION` provides an indication of the available resources in the LAN.

These notifications have to be transmitted from APs to AR. For performance reasons, it is important to avoid sending too many notifications with minor updates which the AR does not care about. This section describes some procedures proposed for a QoSAL implementation[5] to "compress" (suppress would be a more accurate term) notifications before sending them. As a result, fewer notifications will actually be transmitted by the Abstraction Layer than the ones reported by the Drivers.

General Considerations

It should be noted that compression of notifications only applies to unsolicited notifications, sent asynchronously from the APs. The message `AL-RESOURCE-INDICATION` should always be replied to, although compression could take place at the AR to avoid sending this message without need.

An object class is defined inside the QoSAL, called `EventCompressor`. Any notifications from the driver have to be authorised by the event compressor, who will decide if the notification is to be suppressed, or allowed to be transmitted. The event compressor will make this decision based on: 1) the specific values in the notification; 2) the values of the last notification of the same kind; 3) timing considerations. Basically, the QoSAL is allowed to send notifications when both of the following conditions are met:

1. The hysteresis check on the values of the new indication passes (see below for more details);

[5] Note that this was actually implemented for the QoSAL prototype described in Chapter 5.

2. Sufficient time has passed since the last notification.

The above rules apply to both `AL-CNX-INDICATION` and `AL-RESOURCE-INDICATION`.

The Hysteresis Check

The hysteresis check is a stateful operation that allows us to determine if the variation in a value is sufficiently significant, thus ignoring variations small enough that can be neglected for higher-level purposes. At the same time, it is a very simple algorithm, which has low computing overhead. First, a list of thresholds is defined. The state of the hysteresis check is composed of the two closest of these thresholds to the current value, one above and another below. Whenever a new value is reported, one of three things can happen:

1. The value crosses the upper threshold: in this case, the change is considered significant, thus reported; also, the two thresholds are updated;

2. The value crosses the lower threshold: in this case, the change is considered significant, thus reported; also, the two thresholds are updated;

3. The value stays within the two thresholds: in this case, the change is ignored.

4.3 Conclusions

The QoS Abstraction Layer described here satisfies the requirements listed in the beginning of this chapter.

Regarding the QoS requirements, it does indeed allow QoS reservations, using abstract QoS parameters. It is adaptable to multiple L3 QoS models. For example, it is both possible to map individual RSVP/IntServ flows into separate QoSAL connections, or just as easily create a single QoSAL connection for a whole DSCP aggregate and mark all packets with the same DSCP with a single connection identifier.

On the mobility requirements, the QoSAL provides two types of triggers. One is `AL-RESOURCE-INDICATION` and `AL-RESOURCE-DEGRADATION`, which report changes in the overall free bandwidth, allowing mobility protocols to adjust accordingly. The other is `AL-CNX-INDICATION`, which provides notifications when individual connections are degraded. Finally, in order to prepare for an impending terminal handover, the primitive `AL-CNX-ACTIVATE-REQ` has an additional parameter which can be used to accept the QoS parameters but delay activation until the terminal signals its own arrival via the `AL-HO-EXECUTE` primitive.

The auto-configuration requirements are realized by a combination of several factors. On one hand, a protocol was developed to communicate the service primitives across a network to remote L2 equipments. On the other hand, the protocol was designed in a way that it always uses the terminal MAC address as destination, forcing the QoSAL instances running in intermediate L2 nodes to intercept these messages, even though the destination MAC address is different from these intermediate L2 nodes. This has the effect of forcing the messages to discover the correct path to the terminal, taking profit of the automatic routing provided by “learning bridges”. Also, because learning bridges occasionally send copies of received messages through the wrong path the protocol has to ensure that no action is ever taken without waiting for a corresponding confirmation message from the terminal. Finally, there is an `AL-ANNOUNCE` primitive that is used to enable APs to make themselves known to their corresponding AR.

The other requirements are also fulfilled. For example the already mentioned primitive `AL-CNX-INDICATION` realizes the “events” interface to help in cross-layer issues. Moreover, the QoSAL protocol’s soft-state nature ensures that it can recover from consecutive transmission errors. The introduction of the `AL-MULTICAST-JOIN` and `AL-MULTICAST-LEAVE` primitives, along with multicast address support in `AL-CNX-ACTIVATE-REQ`, ensures that multicast is supported by the QoSAL. Unfortunately, multicast support in the protocol could not be implemented in time. Finally, a good degree of modularity has been achieved by splitting the QoSAL functionality in two parts: the QoSAL per se and the QoSAL drivers, which significantly simplifies driver development by freeing it from implementing technology-independent functionality. Examples of QoSAL drivers for some wireless technologies can be found in [CGN+05].

Chapter 5

Proof-of-Concept

5.1 Introduction

In the interest of confirming the validity of the proposed architecture, a prototype implementation of the QoSAL has been developed, and some testing was done on this implementation. The tests presented in this chapter aim at validating the basic functionality of the prototype.

The remaining of this chapter starts in Sec. 5.2 by describing the development environment, followed by an overview of the architecture of the implementation and testbed in Sec. 5.3. Then, the functional tests are detailed in Sec. 5.4, including their results. Finally, some conclusions are drawn in Sec. 5.5 regarding the whole validation experience.

5.2 Development Environment

The main development environment for the prototype implementation was GNU/Linux. It runs on kernel version 2.6.12, or $\geq$ 2.6.8.1 with a patch to add kernel support for the `--ulog` option of `ebtables`. The officially supported distribution is Mandrake 10.0, but other distributions were also heavily tested, such as Slackware 8.1 (with a custom kernel) and Ubuntu 5.04.

The QoSAL code has been written in the Python programming language.[1], with just a small amount of C code. Although Python is an interpreted language, which has some impact on performance, the choice is justified by 1) speed of development, 2) code quality (number of bugs, stability), 3) readability, and 4) maintainability of the code. Of course, in a fully deployed product C would be a better choice, due to memory and processing power constraints of APs. But, this being a prototype, the advantages of Python outlined above clearly outweigh any other concern. The other key software components that QoSAL uses are Linux's *ethernet bridging*[2] and *ethernet bridging tables*[3], whose purpose is described in Sec. 5.3. Finally, other software components that were used in

[1] `http://www.python.org/`
[2] `http://bridge.sourceforge.net/`
[3] `http://ebtables.sourceforge.net/`

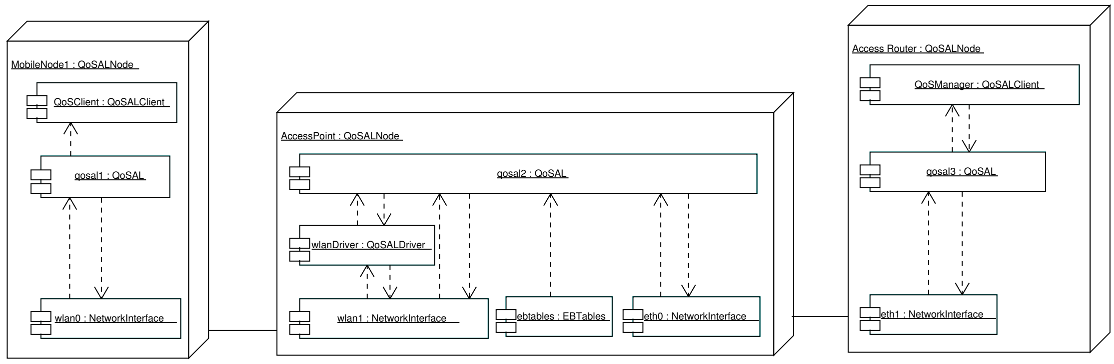


Figure 5.1: Sample QoSAL deployment diagram

development include: (1) GNU Automake, Autoconf, Libtool, for the build system, (2) GCC 3.3, and (3) GNU Emacs editor.

5.3 Architecture

The typical deployment scenario of the QoSAL, MN–AP–AR, and its related components, is depicted in Fig. 5.1. Similarities to the scenario shown in Fig. 4.1 are evident. In the AR, the QoS Manager is represented as a component, which runs in its own process and communicates to the QoSAL component, another process, via a local socket, used to send and receive back primitives. The QoSAL uses the local network interface `eth1`, represented as another component (but one that lives in kernelspace) to send and receive PDUs, by means of a Linux *packet socket*. Quoting from the packet(7) manual page:

> Packet sockets are used to receive or send raw packets at the device driver (OSI Layer 2) level. They allow the user to implement protocol modules in user space on top of the physical layer.

In the MN there is a similar layout of components, with another instance of the QoSAL, the QoS Client fulfilling the role of a QoSAL client, albeit a mostly passive one, and a wireless network interface `wlan0`.

The AccessPoint node is a bit more interesting. It is a simple PC with two network interfaces, one wireless (`wlan1`) and one 802.3 (`eth0`), with *ethernet bridging* taking place between them. To activate ethernet bridging between the two interfaces, a special `br0` *bridge* interface is created, and then the two physical interfaces are attached to it, like this:

```
brctl addbr br0
brctl addif br0 eth1
brctl addif br0 eth2
```

```
ifconfig br0 up
```

It is also possible to activate the Spanning Tree Protocol on this “software bridge”. Under controlled environments, we are certain not to produce cycles, so it is usually not necessary.

Unfortunately, the Linux packet sockets cannot be used to capture packets that are only being “bridged” in this way. If there is an 802.3 frame sent by the AR with destination address of the MN, the packet socket in the AP will not see it. To work around the problem, the ebtables `--ulog` option has been used like this:

```
ebtables -A FORWARD -p 0x1234 --ulog-nlgroup 1 -j DROP
```

The above rule instructs ebtables to queue to userspace any frames with ethertype (L2 protocol) 0x1234, which is the number used by the QoSAL prototype, instead of letting them be forwarded. The complete frames are sent to userspace through a *multicast netlink socket*, along with the following information:

version Version of `ebt_ulog` implementation/protocol used.

indev Name of the network interface from which the frame arrived;

outdev Name of the network interface through which the frame would be transmitted if not intercepted;

physindev Like *indev*, except that the *physical* interface is used instead of bridge virtual interface;

physoutdev Like *outdev*, except that the *physical* interface is used instead of bridge virtual interface;

stamp Time stamp of the time when the packet was queued into userspace;

mark The `ebtables` and `iptables` can contain a *mark* target, which assigns a specified value to a special field in the `sk_buf` structure that encapsulates all packets in kernelspace. This mark value is sent to userpace by `ebt_ulog` as the *mark* field;

hook Number that identifies the ebtables *hook* (equivalent to chain in iptables) that intercepted the packet. Possible values include `NF_BR_PRE_ROUTING`, `NF_BR_LOCAL_IN`, `NF_BR_FORWARD`, `NF_BR_LOCAL_OUT`, `NF_BR_POST_ROUTING`, and `NF_BR_BROUTING`;

data_len, data The complete packet, including Ethernet header and perhaps the VLAN header appended.

Of these fields, only *physindev*, *physoutdev*, and *frame data* are actually used by the QoSAL. Finally, the AccessPoint also has a *QoSAL Driver*, which handles QoS on the wireless interface `wlan1`.

The QoSAL component that is instantiated in all the nodes could be modelled by the class diagram in Fig. 5.2, with some simplifications. The design focuses on modular and clean design, splitting the work among a handful of interconnected objects. Here’s a brief summary of the responsibilities attached to each class:

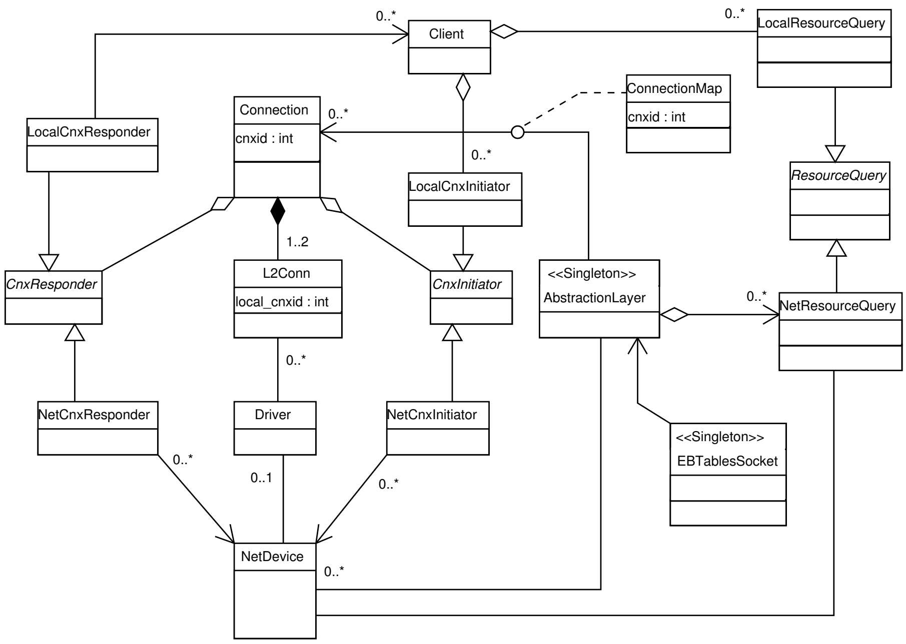


Figure 5.2: Simplified QoSAL class diagram

AbstractionLayer This singleton [GHJV95] class is the central coordinator for the whole system, interconnecting several other objects. For instance, it acts as main message dispatcher, when messages are received from network interfaces or ebtables. It checks whether a message is associated with an existing connection, in which case the message is dispatched to the connection object (through the *iniator* object, as explained below) for further processing. If, on the other hand, the message is connection bound but there is no connection object matching the CnxID, a new connection object is created. Finally, resource query messages are handled similarly. For each unique entity that has sent a resource query, an object is created to handle them, as explained below;

NetDevice This class represents a UNIX network interface, including all its relevant attributes such as name (eg. `eth0`), interface index, or L2 address. It also contains a `PF_PACKET` socket bound to the interface and the QoSAL ethertype. When data arrives in this socket, the QoSAL protocol message is decoded and the resulting message object sent to the AbstractionLayer singleton for further processing. In addition, the class contains a method which receives a message object, encodes it to the QoSAL binary protocol, and sends it through the packet socket;

Driver When a QoSAL Driver component registers itself with the QoSAL, it does so by connecting to a local socket to which the QoSAL is listening, then sending an `AL-DRIVER-REGISTER` message, thus indicating which network interface it wants to handle. At this point, the QoSAL creates a new Driver instance, which contains the socket connected to the driver and is responsible for handling all communications with the driver, including message serialisation. Moreover, the Driver instance registers itself with the corresponding NetDevice instance, which means that a link to the Driver is attached in the NetDevice, allowing the QoSAL to contact the QoSAL driver attached to any given network interface;

Connection This class represents a QoSAL connection. Besides implementing the state machine of Fig. 4.7, it manages the QoS resources on a per-interface/driver basis, which are represented as L2Connection objects. Between zero and two L2Connection objects can be managed by a Connection instance, depending on the specific configuration. For instance, on the example configuration in Fig. 5.1, Connection objects in the AP would have only one L2Connection per Connection, but if there was a QoSAL Driver also handling the `eth0` interface then each Connection would contain two L2Connections.

L2Connection As mentioned above, an L2Connection represents a single QoS reservation, like a Connection, but on a per-interface level.

CnxInitiator Each Connection instance contains a reference to a *connection initiator* object, which presents an abstract interface between the Connection and an entity that requests, or owns, a Connection. A connection initiator sends messages to a Connection object, and can receive back messages to send to the requesting entity. Two subclasses LocalCnxInitiator and NetCnxInitiator implementing this interface:

- **LocalCnxInitiator** puts a Client object in charge of a locally created Connection, and assumes responsibility of periodically refreshing the connection until the client releases it or disconnects from the QoSAL. Messages from the client are forwarded to the connection, while messages sent back from the connection are sent to the correct client;
- **NetCnxInitiator** works on behalf of a remote client, sending connection PDUs to a Connection as they arrive from a network interface. It can also send back response messages through the same network interface.

CnxResponder Similarly to CnxInitiator, CnxResponder represents an abstract frontier between a connection and an entity that receives, or accepts, a connection. The two subclasses that implement this abstract interface are:

- **LocalCnxResponder** This object "reflects" `AL-CNX-ACTIVATE-REQ` PDUs back to the initiator as a `AL-CNX-ACTIVATE-RESP`, thereby terminating the connection and accepting activation requests. At the same time, any local clients connected to the QoSAL are notified of new connections being activated, with a `AL-CNX-INDICATION` upcall primitive, as explained in Chapter 4.
- **NetCnxResponder** This object is used in the responder role when a connection activation is received containing a destination address that does not match

any local interface. It merely retransmits the received messages through the outgoing interface.

Client Represents a client connection to the QoSAL. A Client instance "owns" a LocalCnxInitiator object per connection. The Client class decodes messages from clients and forwards equivalent messages to the local initiator, and conversely receives messages from the initiator and encodes them to send to the clients through a socket.

ResourceQuery This class abstractly represents an entity that has requested a resource query. The two subclasses implement this interface are:

LocalResourceQuery Presents a local client as a resource query entity, meaning that the corresponding `AL-RESOURCE-INDICATION` PDU will be converted to a client interface primitive and sent to all the clients that requested it.

NetResourceQuery Works on behalf of a remote client, meaning that `AL-RESOURCE-INDICATION` PDUs are forwarded through a network interface.

Perhaps an example can illustrate better how the different objects interact. The object diagram in Fig. 5.3 shows the sequence of messages exchanged the system represented by Fig. 5.1 when a connection is being "refreshed". The sequence starts by a LocalCnxInitiator at the AR sending an `AL-CNX-ACTIVATE-REQ` message to the Connection object, triggered by a timer. The Connection, in turn, updates its state machine using this message, then forwards it to the responder, which then sends it to the associated NetDevice, where it is serialized and transmitted through `eth1` as an 802.3 frame. This same frame arrives at the AccessPoint through the network interface `eth0`. However, as already explained, the packet socket listening on `eth0` is unable to receive frames whose destination MAC addresses do not match the address of `eth0`. However, ebtables is able to intercept it, and send it to userspace, to be received by the ebtables object. The message is decoded and, like all arbitrary messages arriving from the network, is routed through the AbstractionLayer singleton, which is able to determine from the message type and CnxID which Connection object it belongs to. The message is then sent to the Connection as if arriving from its "initiator". As in the AR, the Connection updates the state machine and forwards the message to the responder, which in turn encodes it and sends through the socket attached to the `wlan1` network interface. At the MN, the message arrives at the packet socket listening to `wlan0`, since the destination MAC of the message matches this interface. As before, the AbstractionLayer singleton is used to find the corresponding Connection, which receives the `AL-CNX-ACTIVATE-REQ`, updates the state machine, then forwards it to the responder. This time, however, the CnxResponder is actually a LocalCnxResponder. It receives the `AL-CNX-ACTIVATE-REQ` and replies with a `AL-CNX-ACTIVATE-RESP` back to the address that was the source address of the `AL-CNX-ACTIVATE-REQ`. The return path of `AL-CNX-ACTIVATE-RESP` is more or less the reverse of the path taken by `AL-CNX-ACTIVATE-REQ`, but this time the message travels from responder→Connection→initiator, with the Connection state machine being updated in

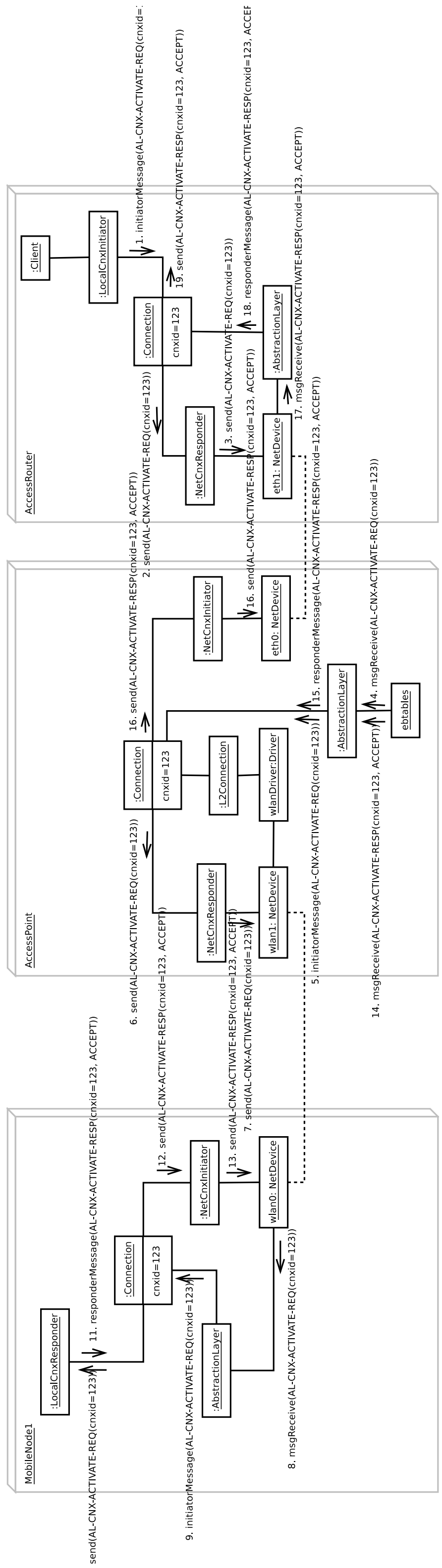


Figure 5.3: Sample QoSAL object diagram showing a connection being refreshed

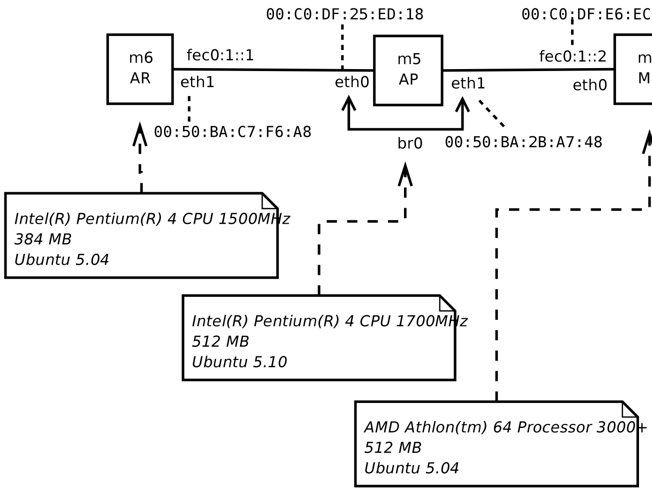


Figure 5.4: Test scenario

each node, and finally arriving back to where it all started, in the LocalCnxInitiator at the AR.

It should be more clear by now why this *CnxResponder* ⇔ *Connection* ⇔ *CnxInitiator* architectural split was devised. The reader should notice that the bulk of the connection state machine is factored out into the Connection class, making it independent of location, type of entity that created the connection (local or remote), and type of entity that accepted the connection (local or remote). The initiator and responder objects, on the other hand, have very little functionality, merely providing some "glue" between a Connection and the rest of the system.

5.4 Tests

The purpose of these tests is to evaluate the correctness of the generic part of QoS Abstraction Layer, without considering technology-specific modules. Thus, a testbed was assembled, composed of a Mobile Node, an Access Point, and an Access Router, as depicted in Fig. 5.4. On each of the nodes in this figure, a QoSAL instance was running, along with a "dummy" QoS driver for each interface: one dummy driver in the AR, two in the AP, and another in the MN. Each dummy driver is statically configured with a bandwidth of 1 Mbit/s, but that value can be changed in runtime by human intervention for testing purposes. During each test, packets were captured in the 'eth1' interface of the AR. The names m6, m5, and m2 denote the host names of each machine and are not relevant for the tests. In the AP, the two interfaces eth0 and eth1 are grouped together to form an ethernet bridge, br0.

Essentially three groups of tests were performed. In Sec. 5.4.1, the aspects related with QoS reservations were tested, including connection activation, modification, and deactivation, as well as resource query and indication. In Sec. 5.4.2, we test the ability of the QoSAL at AR to receive messages from the QoSAL at AP announcing its presence, and asynchronous QoS degradation notifications. Finally, Sec. 5.4.3 includes a test involving an hypothetical mobility scenario, with handover preparation and execution.

5.4.1 QoS resource query, connection activation, modification, and deactivation

The primitives `AL-RESOURCE-QUERY/INDICATION`, `AL-CNX-ACTIVATE-REQ/RESP`, `AL-CNX-MODIFY-REQ/RESP`, and `AL-CNX-DEACTIVATE` have been tested together, in a single test run, the reason being that these primitives are interdependent. It is not possible to modify or deactivate a connection without first activating it. Also, as connections are being activated, modified, and deactivate the available QoS resources in each driver keep changing, thus generate resource indications.

In the following test, there is one "dummy client" that connects to the QoSAL in the MN, pretending to be the QoS Client. Another dummy client, connecting to QoSAL in the AR, plays the role of QoS Manager, and performs the following series of operations (Fig. 5.5):

1. Sends a AL-RESOURCE-QUERY, with the AP as destination MAC address;
2. Waits for a AL-RESOURCE-INDICATION;
3. Sends, three times, AL-CNX-ACTIVATE-REQ, with the MN IPv6 address. The objective of this test is to evaluate how the QoSAL can cope with parallel requests;
4. Waits for three AL-CNX-ACTIVATE-RESP, then takes note of the returned Cnx-IDs;
5. After waiting for a keypress, the test program proceeds to modify the QoS for all connections, by decreasing their tx_rspec.R and rx_rspec.R values by 25 %, then sending out all AL-CNX-MODIFY-REQs in parallel;
6. The program then waits for the three expected AL-CNX-MODIFY-RESPs;
7. After waiting for a key press, the connections are then released by sending three AL-CNX-DEACTIVATEs in parallel;
8. Finally, the test program keeps listening for messages for some more time, in order to receive any final resource indications.

The resulting log files can be found in Sec. A.1. The logs show the connections being successfully activated, as well as subsequent modification and deactivation. The results are, thus, correct. The following additional observations can be drawn from these logs:

- Between the client sending a RESOURCE-QUERY and a the arrival of a RESOURCE-INDICATION reply, approximately 18 ms elapsed;

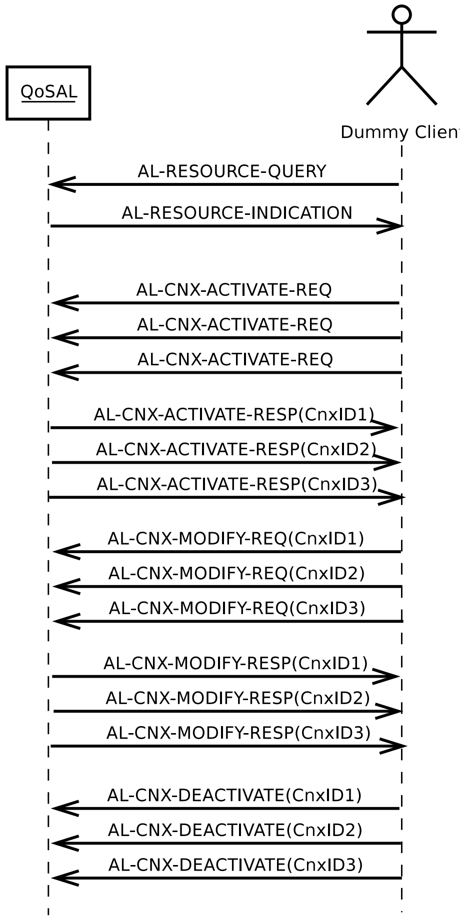


Figure 5.5: The test “QoS resource query, connection activation, modification, and deactivation”

- Activating a single connection took 64 ms. For the three connections together, it took 99 ms, meaning an average of 33 ms per connection. Obviously, the first connection took longer than the following ones, which is a matter that deserves further investigation. Perhaps it is due to an additional delay in the first connection to perform the L3/L2 address mapping (neighbour discovery);

- Modifying the three connections took 75 ms, or 15 ms per connection;

- The logs show that all the relevant dummy drivers are being consulted for resource queries and QoS reservations;

- The QoSAL nodes exchange `AL-CNX-ACTIVATE-REQ/RESP` PDUs to implement connection modification. This is not an error; it is how the protocol works. By detecting a `AL-CNX-ACTIVATE-REQ` for a connection that already exists but with different QoS values, the respective connection is automatically modified, by sending `AL-CNX-MODIFY-REQ` to the respective drivers;

- Regarding resource queries, we observed several interesting results, namely:

  – The first resource indication correctly reports the 1 Mbit/s bandwidth that is statically configured in the dummy drivers;

  – As connections are being activated, modified, and deactivated, successive resource indications arrive, reflecting the amount of available bandwidth at each instant, as consequence of the status of QoS reservations;

  – The resource indications from the AP are being received in duplicated. This is due to both dummy drivers in the AP independently sending resource indications to the QoSAL, and it not being smart enough to merge the two indications into a single message to send to the AR. Future work will try to solve this issue.

- We can also observe, in the MN, the dummy client pretending to be the QoS Client receiving notifications for connections as they are activated or destroyed:

  – When a new connection is activated, it receives a AL-CNX-INDICATION primitive;

  – When a connection is deactivated, a CNX-ACTIVATE-RESP is received, with the corresponding CnxID, and a “result” REJECT.

Regarding the packet capture in the AR interface, the screenshot in Fig. 5.6 shows the open source network analyser Ethereal, extended with a packet dissector plugin for the QoSAL protocol. We can see from the screenshot that it takes about 42 ms between the first `AL-CNX-ACTIVATE-REQ` being sent and the first `AL-CNX-ACTIVATE-RESP` received. For the three connections, the total time is 66 ms, or 22 ms per connection.

Following up on a suspicion (actually also some profiling) that all the detailed logging functions in the QoSAL are slowing it down considerably, another run was made with

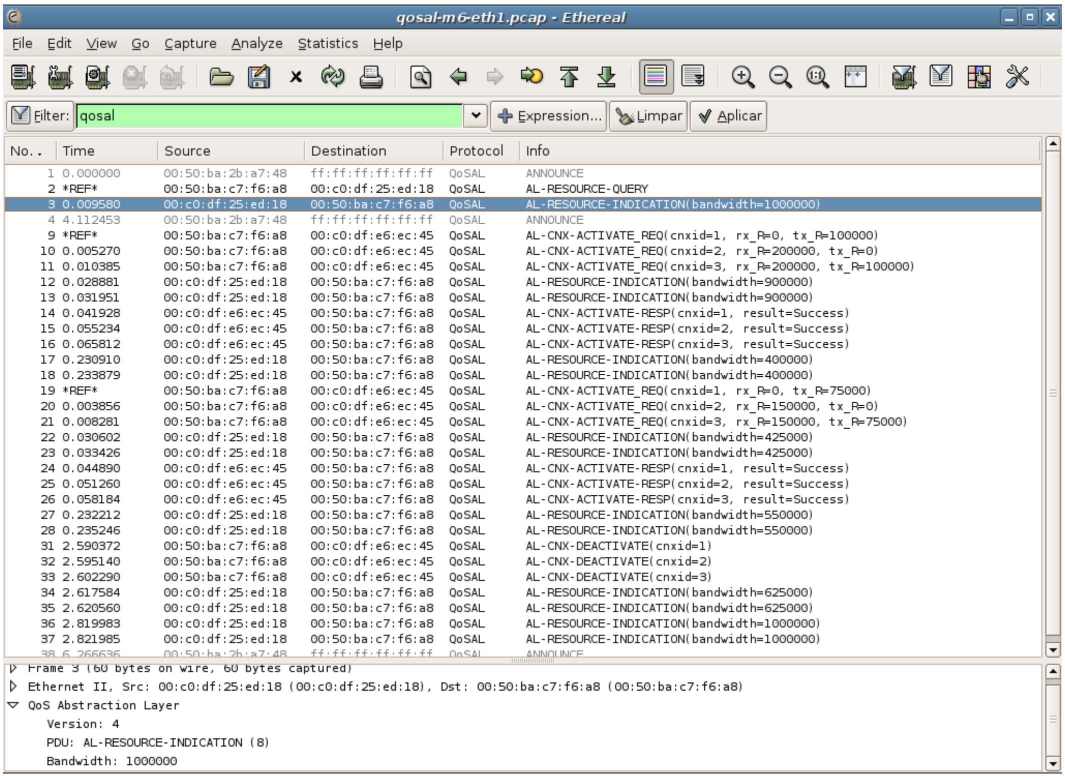


Figure 5.6: Packet capture for the "QoS resource query, connection activation, modification, and deactivation" test.

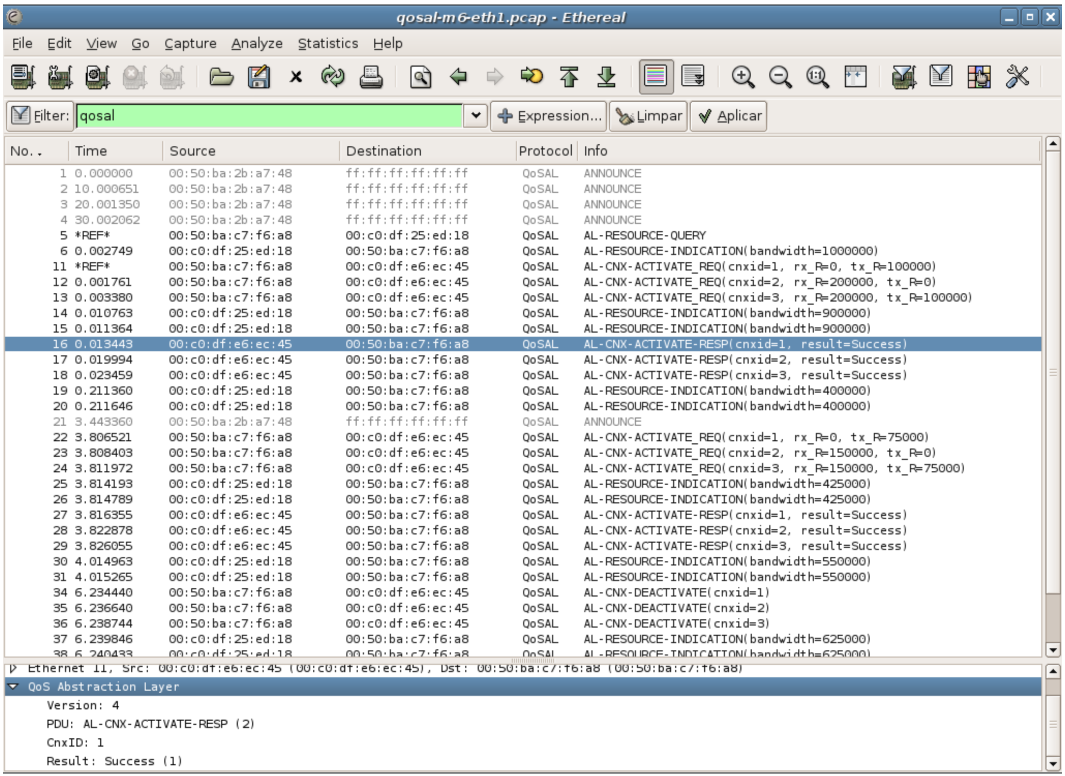


Figure 5.7: Packet capture for the "QoS resource query, connection activation, modification, and deactivation" test with logging disabled.

all logging disabled, eliminating around 90 % of the overhead associated with logging, and only the tcpdump process capturing packets to a file for subsequent analysis. The Ethereal screenshot in Fig. 5.7 reflects this second test condition. The results this time show 13 ms for the first connection, and 23 ms for the three connections, or 7.7 ms per connection. That is, with logging active the connections activation time increased 66/23 = 2.87 times. Therefore, an overhead of almost 3 times should be considering when viewing the results of these tests.

5.4.2 Resource degradation and autodiscovery of APs

The objective of this test is to check that the functionality needed by the Performance Agent, a key component to implement Network Initiated Handover, is implemented correctly. The test consists on the following steps (Fig. 5.8:

1. A fake PA module connects to the QoSAL;

2. Meanwhile, the QoSAL@AP broadcasts an announcement (ANNOUNCE) message, which is received by the QoSAL@AR;

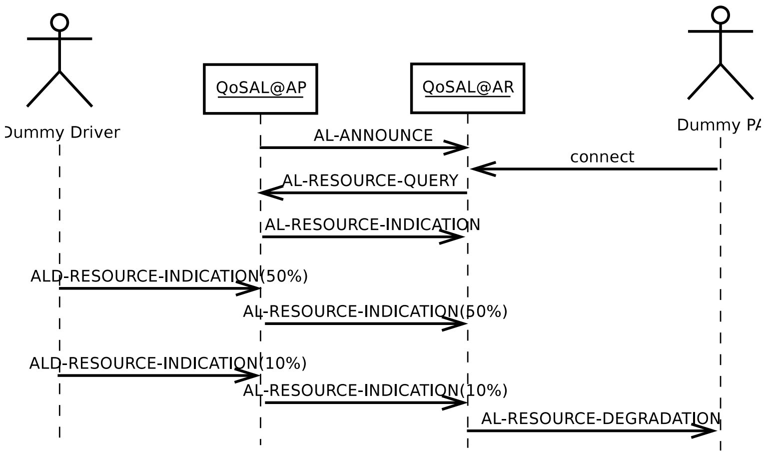


Figure 5.8: The test “Resource degradation and autodiscovery of APs”

3. The human operator manually changes the available bandwidth to 0.5 Mb/s, causing:

   (a) A resource indication sent to the QoSAL@AR;

   (b) **No** QoS degradation being sent to the PA, since it is above the QoS degradation threshold (= 25 % of theoretical maximum bandwidth of the technology);

4. The human operator manually changes the available bandwidth to 0.1 Mbit/s, causing:

   (a) A resource indication sent to the QoSAL@AR;

   (b) A QoS degradation being being sent to the PA, since it is below the QoS degradation threshold (0.25 Mbit/s);

The log files in Sec. A.2 indicate that a RESOURCE-DEGRADATION primitive is sent to the PA only when the free bandwidth changes to 10%, but not when it reaches 50%, which means it is working correctly. The screenshot in Fig. 5.9 shows what happens from the protocol point of view.

5.4.3 Handover preparation and execution

The objective of this test is to check the working conditions of the primitives and parameters used to support mobility scenarios.

One of the problems of handover is that the QoSAL in the new AR may receive connection activation requests before the mobile terminal physically connecting to it. Under normal conditions, the QoSAL would immediately try to activate a connection, which would fail. Therefore, the parameter “handoff” in the `AL-CNX-ACTIVATE-REQ` primitive must be set to true when making a reservation, instructing the QoSAL to delay the

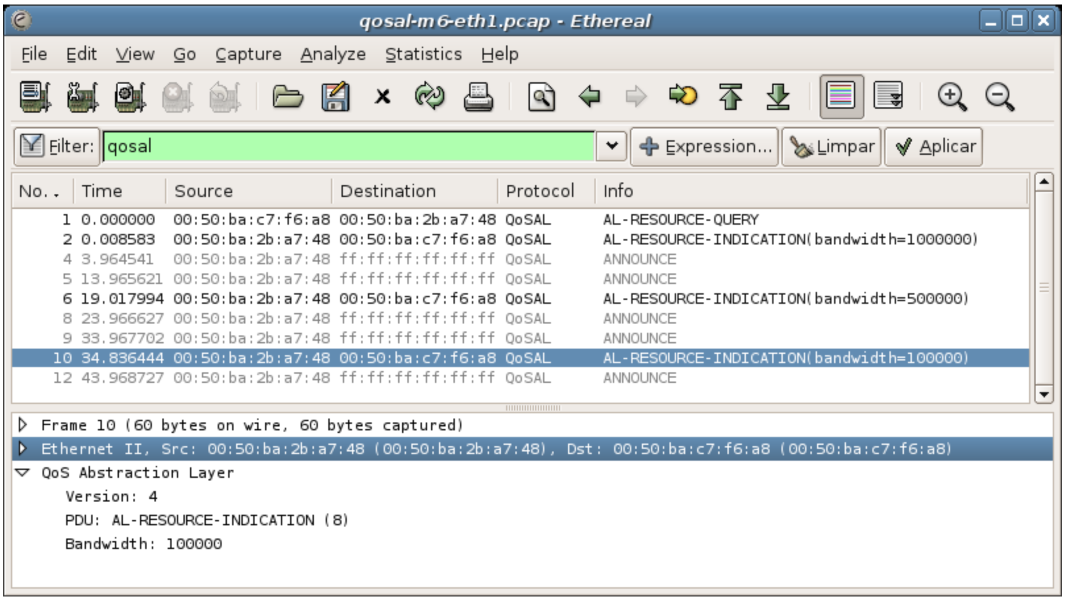


Figure 5.9: Packet capture for the "resource degradation and autodiscovery" test.

connection activation until a `AL-HO-EXECUTE` message is sent by the terminal to advertise its arrival to the new AR. In the MN, the `AL-HO-EXECUTE` is sent by the QoSAL when it receives an equally named trigger from the QoS Client, which in turn is triggered by a message sent to it by the MTC.

This test comprised the following steps (Fig. 5.10):

1. A dummy client (QoS Manager) at the AR connects to QoSAL, and requests activation of a connection with the parameter handoff=True;

2. A dummy client (QoS Client) at the MN sends to the QoSAL a HO-EXECUTE message, containing the new AR IPv6 address as parameter;

3. The QoSAL@MN sends a AL-HO-EXECUTE to the QoSAL@AR;

4. The connection is activated;

5. An activation response is finally sent to the client that requested activation at the AR.

The log files of this test can be found in Sec. A.3, and the corresponding Ethereal capture is in Fig. 5.11. We could observe from the logs that the connection activation is delayed until the `HO-EXECUTE` primitive arrives from the MN, and thus conclude that the test passed successfully.

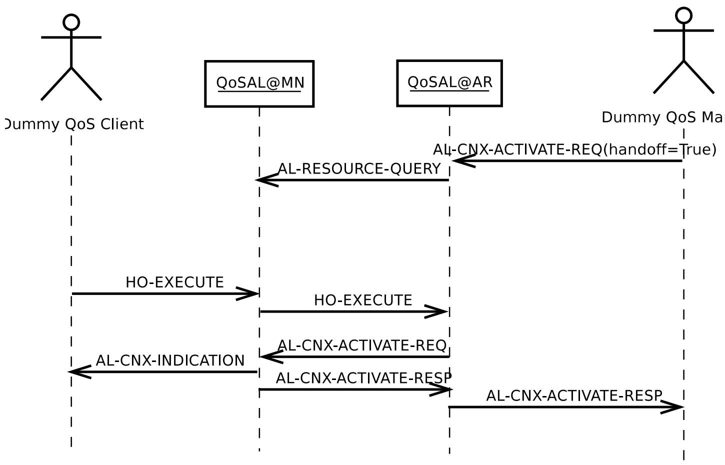


Figure 5.10: The test "Handover preparation and execution"

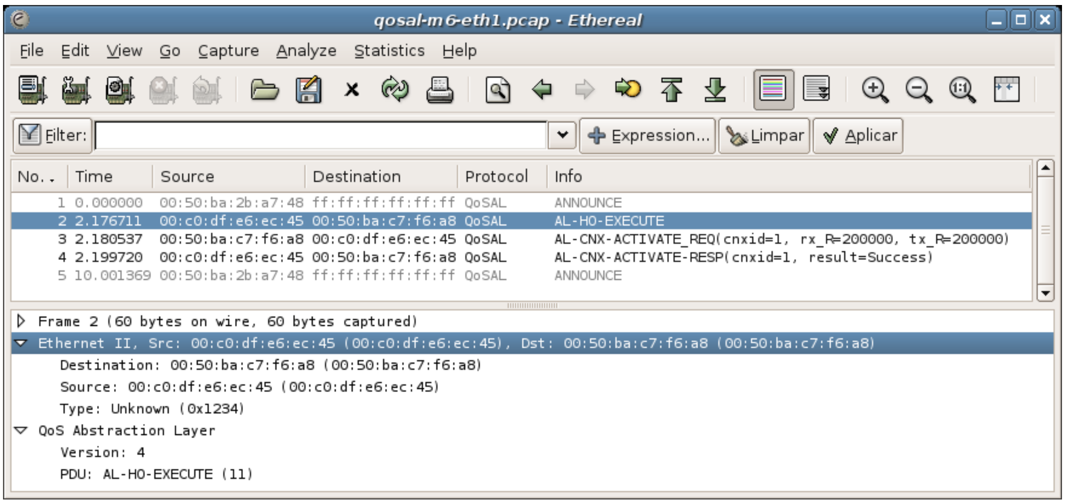


Figure 5.11: Packet capture for the "handover preparation and execution" test.

5.5 Conclusion

A prototype for the QoSAL design and protocol was described in this chapter. The focus of this prototype has been on developing a clean, object-oriented, readable, and modular implementation architecture. We believe this goal was successfully achieved, allowing the prototype to be easily and quickly extended to test new ideas. This software module is also being used in the IST DAIDALOS [Dai] research project, giving it opportunity to be tested with realistic usage patterns.

A few tests that exercised the majority of the functionality offered by the QoSAL, including connection activation, modification, deactivation, resource query/indication, and handover were conducted, with satisfactory results. The QoSAL behaved correctly and as expected[4].

These tests also point the way for future improvements. For instance, in the first test three connections are always managed in parallel: activated, modified, deactivated. This is not an uncommon scenario in real world usage pattern[5]. An interesting optimisation would consist in having the QoS send all related requests in a single PDU, instead of the three separate PDUs that are used now.

---

[4]Of course, the implementation had already been subjected to an extensive period of debugging, hence the correctness of the results.

[5]Consider, for example, an audio-video telephony or streaming application; normally the audio and video flows are reserved at the same time.

Chapter 6

Conclusions

In this thesis a detailed description of the QoS Abstraction Layer was presented. It is both a Layer 2.5 signalling protocol for reserving QoS resources across a wireless access network, and an abstract interface for such reservations independently of the L2 technologies involved. It leverages the Learning Bridge property of IEEE 802 wireless APs to provide automatic discovery of the path for the reservations towards a given terminal, and works with no prior knowledge of the topology of the access network, which may even include multiple concatenated APs. Also provided is the ability to query available resources in the network, receive QoS degradation notifications, and make multicast reservations, as well as active support for integration with IP mobility.

In the course of the research for this solution, several L2 technologies were considered, such as IEEE 802.1D, Bluetooth, and UMTS, all of which, with special emphasis on the latter, helped shape the QoSAL model. Likewise, L3 QoS models like IntServ/RSVP, DiffServ, and MPLS were factored into the design process, yielding an abstract QoS model that is reasonably flexible to be used in conjunction with most L3 QoS models. Moreover, requirements particularly relevant to 4G networks, such mobility and robustness against transmission errors, were also considered in the design. In order to solve scenarios involving multiple concatenated heterogeneous L2 technologies, a reservation protocol was developed in complement of the abstract interface. Finally, a prototype implementation was developed, as proof-of-concept, and tested, providing valuable feedback on the overall architecture.

6.1 Original Contributions

The original contributions of the work associated with this thesis are:

1. An abstract QoS service interface that includes the common aspects of the QoS interfaces of the most important L2 wireless technologies, as well as 4G network requirements. It contains primitives to reserve QoS resources, query the available resources in an access network, and receive notifications of QoS degradation. It works with abstract QoS parameters, which are translated into technology specific

parameters by dedicated modules. In addition, special primitives and parameters provide mobility integration support, and multicast QoS.

2. An associated signalling protocol for reserving QoS in remote APs. It is able to work across a multi-hop L2 network, and automatically discovers the path to a terminal in an access network of multiple APs interconnected in a tree-like topology. It reserves QoS in the correct set of APs, while avoiding the problems associated with the forwarding algorigthm of IEEE 802.1D bridges, and wireless links errors.

6.2 Results

A protype has been developed which implements an *abstract interface* and delegates the technology-dependent QoS realization task to QoSAL drivers, through the *driver interface*. This prototype provided valuable feedback into the design. Most of this feedback has been transparently taken into account and already used to improve the QoSAL specification/design. For example, during the initial design phase, there was a possibilty of a client sending multiple concurrent `AL-CNX-ACTIVATE-REQ` messages without waiting for the corresponding `AL-CNX-ACTIVATE-RESP`s. Since the activation request has no CnxID (because it is the QoSAL that allocates CnxIDs), there is no way to associate subsequent responses to their respective requests. Imposing the serialization of connection activations limits seriously the system performance when there is a batch of connection activations to be performed at the same time. The solution adopted consists in reordering `AL-CNX-ACTIVATE-RESP`s so that they can match the order of the corresponding `AL-CNX-ACTIVATE-REQ`s. In this way, the client is able to perform multiple parallel requests, because it knows that responses will arrive in the same order.

The QoSAL prototype also implementats the QoSAL protocol, which enables the corresponding state machine to be immediately tested and evaluated. Moreover, this allowed us to test more complex scenarios, such as QoS reservations in remote APs. It has been possible to observe some robustness properties confered by the soft-state nature of the protocol.

6.3 Future Work

Future work to be performed based on in this QoS Abstraction Layer includes:

More thorough validation (1) Stress-testing, or formal protocol validation, to discover problems in the protocol, such as bad states and race conditions; (2) Test the effect of transmission errors on the robustness and performance of the protocol; (3) Performance tests, to measure the scalability and network load overhead caused by the protocol;

Coalescing PDUs The design and implementation of an optimization consisting on coalescing multiple similar requests in a single message or frame would provide benefits, in terms of time and overhead bandwidth, particularly in handover scenarios.

Security and compression Integration of security and header compression features into the base QoSAL architecture would be worthy additions.

Better handover integration The process of reserving QoS resources in L2 may involve some calculations that take time, to more or less degree depending on the technology. This preparation time is particularly pernicious during handover, for during that time the flows will not enjoy any QoS guarantees, being relegated to mere best effort treatment. A possible solution to this problem would require the mobility subsystem to notify the QoSAL, in advance, which is the target AR and AP to which a terminal will handover, so that QoS resources can start being prepared with time.

MPLS The current QoSAL architecture depends on packets having the Flow Label field marked in order to identify the QoS reservation each belongs to. This creates an unfortunate dependency on IPv6 and breaks layering rules by making a 2.5 layer (QoSAL) depend on L3. A possible solution for this dependency problem would be for the QoSAL to insert its own user plane header containing the connection identifier for each packet. The information that would be required in this header would be similar to the information contained in MPLS. So, instead of inventing yet another header, we would simply insert an MPLS header, with $Label = CnxID$.

Cross-layer An architecture like the QoSAL is adequate for the role of "link context" identified in [CRR04], making cross-layer optimizations, such as intelligent power control based on delay requirements, become a possibility.

Appendix A

Log Files

A.1 QoS resource query, connection activation, modification, and deactivation

A.1.1 Access Router

```
1130258140.909747 client INFO    From client 10: RESOURCE-QUERY(
   target=<client.L2Addr 00:C0:DF:25:ED:18>)
1130258140.912198 proto  INFO    Sending through eth1: RESOURCE-
   QUERY(dst=<L2Addr/Ethernet 00:C0:DF:25:ED:18>, src=<L2Addr/
   Ethernet 00:50:BA:C7:F6:A8>)
1130258140.923231 proto  INFO    Recv from eth1: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:C0:DF:25:ED:18>, BW=1e+06, MAX=1e+06)
1130258140.924967 driver INFO    To driver eth1: RESOURCE-QUERY()
1130258140.927100 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=1e+06)
1130258140.928078 client INFO    To client 10: RESOURCE-INDICATION
   (target=<client.L2Addr (interface 0)/00:C0:DF:25:ED:18>, BW=1e
   +06)
1130258145.026157 proto  INFO    Recv from eth1: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
1130258145.835185 client INFO    From client 10: CNX-ACTIVATE-REQ(
   dst=<L3Addr fec0:1::2>, tx=(TSpec(100000, 12000, 100000, 12000,
    512), RSpec(0, 100000)), rx=(TSpec(0, 12000, 0, 12000, 512),
   RSpec(0, 0)), prio=0, handoff=0, data_len=0)
1130258145.855346 proto  INFO    Sending through eth1: CNX-
   ACTIVATE-REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<
   L2Addr/Ethernet 00:50:BA:C7:F6:A8>, CnxID=1, tx=(TSpec(100000,
   12000, 100000, 12000, 512), RSpec(0, 100000)), rx=(TSpec(0,
   12000, 0, 12000, 512), RSpec(0, 0)), prio=0, data_len=0)
1130258145.858218 client INFO    From client 10: CNX-ACTIVATE-REQ(
   dst=<L3Addr fec0:1::2>, tx=(TSpec(0, 12000, 0, 12000, 512),
```

```
    RSpec(0, 0)), rx=(TSpec(200000, 12000, 200000, 12000, 512),
    RSpec(0, 200000)), prio=0, handoff=0, data_len=0)
1130258145.860835 proto  INFO    Sending through eth1: CNX-
    ACTIVATE-REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<
    L2Addr/Ethernet 00:50:BA:C7:F6:A8>, CnxID=2, tx=(TSpec(0,
    12000, 0, 12000, 512), RSpec(0, 0)), rx=(TSpec(200000, 12000,
    200000, 12000, 512), RSpec(0, 200000)), prio=0, data_len=0)
1130258145.863402 client INFO    From client 10: CNX-ACTIVATE-REQ(
    dst=<L3Addr fec0:1::2>, tx=(TSpec(100000, 12000, 100000, 12000,
     512), RSpec(0, 100000)), rx=(TSpec(200000, 12000, 200000,
    12000, 512), RSpec(0, 200000)), prio=0, handoff=0, data_len=0)
1130258145.865955 proto  INFO    Sending through eth1: CNX-
    ACTIVATE-REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<
    L2Addr/Ethernet 00:50:BA:C7:F6:A8>, CnxID=3, tx=(TSpec(100000,
    12000, 100000, 12000, 512), RSpec(0, 100000)), rx=(TSpec
    (200000, 12000, 200000, 12000, 512), RSpec(0, 200000)), prio=0,
     data_len=0)
1130258145.886512 proto  INFO    Recv from eth1: RESOURCE-
    INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
    /Ethernet 00:C0:DF:25:ED:18>, BW=900000, MAX=1e+06)
1130258145.888405 driver INFO    To driver eth1: RESOURCE-QUERY()
1130258145.890845 proto  INFO    Recv from eth1: RESOURCE-
    INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
    /Ethernet 00:C0:DF:25:ED:18>, BW=900000, MAX=1e+06)
1130258145.892778 driver INFO    From driver eth1: RESOURCE-
    INDICATION(BW=1e+06)
1130258145.894166 client INFO    To client 10: RESOURCE-INDICATION
    (target=<client.L2Addr (interface 0)/00:C0:DF:25:ED:18>, BW
    =900000)
1130258145.899585 proto  INFO    Recv from eth1: CNX-ACTIVATE-RESP
    (dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr/Ethernet
    00:C0:DF:E6:EC:45>, CnxID=1, result=ACCEPT)
1130258145.901657 driver INFO    To driver eth1: CNX-ACTIVATE-REQ(
    global_cnxid=1, tx=(TSpec(100000, 12000, 100000, 12000, 512),
    RSpec(0, 100000)), rx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0,
     0)), prio=0)
1130258145.905342 driver INFO    From driver eth1: RESOURCE-
    INDICATION(BW=900000)
1130258145.906775 client INFO    To client 10: RESOURCE-INDICATION
    (target=<client.L2Addr (interface 0)/00:C0:DF:25:ED:18>, BW
    =900000)
1130258145.910843 driver INFO    From driver eth1: CNX-ACTIVATE-
    RESP(1, ACCEPT)
1130258145.912281 client INFO    To client 10: CNX-ACTIVATE-RESP
    (1, ACCEPT)
1130258145.915065 proto  INFO    Recv from eth1: CNX-ACTIVATE-RESP
    (dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr/Ethernet
    00:C0:DF:E6:EC:45>, CnxID=2, result=ACCEPT)
```

```
1130258145.917108 driver INFO    To driver eth1: CNX-ACTIVATE-REQ(
    global_cnxid=2, tx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0, 0)
    ), rx=(TSpec(200000, 12000, 200000, 12000, 512), RSpec(0,
    200000)), prio=0)
1130258145.920588 driver INFO    From driver eth1: RESOURCE-
    INDICATION(BW=700000)
1130258145.922101 driver INFO    From driver eth1: CNX-ACTIVATE-
    RESP(2, ACCEPT)
1130258145.923480 client INFO    To client 10: CNX-ACTIVATE-RESP
    (2, ACCEPT)
1130258145.926111 proto  INFO    Recv from eth1: CNX-ACTIVATE-RESP
    (dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr/Ethernet
    00:C0:DF:E6:EC:45>, CnxID=3, result=ACCEPT)
1130258145.928088 driver INFO    To driver eth1: CNX-ACTIVATE-REQ(
    global_cnxid=3, tx=(TSpec(100000, 12000, 100000, 12000, 512),
    RSpec(0, 100000)), rx=(TSpec(200000, 12000, 200000, 12000, 512)
    , RSpec(0, 200000)), prio=0)
1130258145.931685 driver INFO    From driver eth1: RESOURCE-
    INDICATION(BW=400000)
1130258145.933198 driver INFO    From driver eth1: CNX-ACTIVATE-
    RESP(3, ACCEPT)
1130258145.934603 client INFO    To client 10: CNX-ACTIVATE-RESP
    (3, ACCEPT)
1130258146.088611 proto  INFO    Recv from eth1: RESOURCE-
    INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
    /Ethernet 00:C0:DF:25:ED:18>, BW=400000, MAX=1e+06)
1130258146.090357 driver INFO    To driver eth1: RESOURCE-QUERY()
1130258146.091501 proto  INFO    Recv from eth1: RESOURCE-
    INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
    /Ethernet 00:C0:DF:25:ED:18>, BW=400000, MAX=1e+06)
1130258146.094384 driver INFO    From driver eth1: RESOURCE-
    INDICATION(BW=400000)
1130258146.095752 client INFO    To client 10: RESOURCE-INDICATION
    (target=<client.L2Addr (interface 0)/00:C0:DF:25:ED:18>, BW
    =400000)
1130258148.755270 client INFO    From client 10: CNX-MODIFY-REQ(
    CnxID=1, tx=(TSpec(75000, 12000, 75000, 12000, 512), RSpec(0,
    75000)), rx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0, 0)))
1130258148.757090 proto  INFO    Sending through eth1: CNX-
    ACTIVATE-REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<
    L2Addr/Ethernet 00:50:BA:C7:F6:A8>, CnxID=1, tx=(TSpec(75000,
    12000, 75000, 12000, 512), RSpec(0, 75000)), rx=(TSpec(0,
    12000, 0, 12000, 512), RSpec(0, 0)), prio=0, data_len=0)
1130258148.761324 client INFO    From client 10: CNX-MODIFY-REQ(
    CnxID=2, tx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0, 0)), rx=(
    TSpec(150000, 12000, 150000, 12000, 512), RSpec(0, 150000)))
1130258148.763124 proto  INFO    Sending through eth1: CNX-
    ACTIVATE-REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<
    L2Addr/Ethernet 00:50:BA:C7:F6:A8>, CnxID=2, tx=(TSpec(0,
```

```
    12000, 0, 12000, 512), RSpec(0, 0)), rx=(TSpec(150000, 12000,
    150000, 12000, 512), RSpec(0, 150000)), prio=0, data_len=0)
1130258148.765159 client INFO    From client 10: CNX-MODIFY-REQ(
    CnxID=3, tx=(TSpec(75000, 12000, 75000, 12000, 512), RSpec(0,
    75000)), rx=(TSpec(150000, 12000, 150000, 12000, 512), RSpec(0,
     150000)))
1130258148.767066 proto  INFO    Sending through eth1: CNX-
    ACTIVATE-REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<
    L2Addr/Ethernet 00:50:BA:C7:F6:A8>, CnxID=3, tx=(TSpec(75000,
    12000, 75000, 12000, 512), RSpec(0, 75000)), rx=(TSpec(150000,
    12000, 150000, 12000, 512), RSpec(0, 150000)), prio=0, data_len
    =0)
1130258148.791649 proto  INFO    Recv from eth1: RESOURCE-
    INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
    /Ethernet 00:C0:DF:25:ED:18>, BW=425000, MAX=1e+06)
1130258148.793044 driver INFO    To driver eth1: RESOURCE-QUERY()
1130258148.794808 proto  INFO    Recv from eth1: RESOURCE-
    INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
    /Ethernet 00:C0:DF:25:ED:18>, BW=425000, MAX=1e+06)
1130258148.797316 driver INFO    From driver eth1: RESOURCE-
    INDICATION(BW=400000)
1130258148.798291 client INFO    To client 10: RESOURCE-INDICATION
    (target=<client.L2Addr (interface 0)/00:C0:DF:25:ED:18>, BW
    =400000)
1130258148.805953 proto  INFO    Recv from eth1: CNX-ACTIVATE-RESP
    (dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr/Ethernet
    00:C0:DF:E6:EC:45>, CnxID=1, result=ACCEPT)
1130258148.807321 driver INFO    To driver eth1: CNX-MODIFY-REQ(
    CnxID=1, tx=(TSpec(75000, 12000, 75000, 12000, 512), RSpec(0,
    75000)), rx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0, 0)))
1130258148.810915 driver INFO    From driver eth1: RESOURCE-
    INDICATION(BW=425000)
1130258148.812109 driver INFO    From driver eth1: CNX-MODIFY-RESP
    (1, ACCEPT)
1130258148.813070 client INFO    To client 10: CNX-MODIFY-RESP(1,
    ACCEPT)
1130258148.814160 proto  INFO    Recv from eth1: CNX-ACTIVATE-RESP
    (dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr/Ethernet
    00:C0:DF:E6:EC:45>, CnxID=2, result=ACCEPT)
1130258148.816722 driver INFO    To driver eth1: CNX-MODIFY-REQ(
    CnxID=2, tx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0, 0)), rx=(
    TSpec(150000, 12000, 150000, 12000, 512), RSpec(0, 150000)))
1130258148.819840 proto  INFO    Recv from eth1: CNX-ACTIVATE-RESP
    (dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr/Ethernet
    00:C0:DF:E6:EC:45>, CnxID=3, result=ACCEPT)
1130258148.821181 driver INFO    To driver eth1: CNX-MODIFY-REQ(
    CnxID=3, tx=(TSpec(75000, 12000, 75000, 12000, 512), RSpec(0,
    75000)), rx=(TSpec(150000, 12000, 150000, 12000, 512), RSpec(0,
     150000)))
```

```
1130258148.824784 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=475000)
1130258148.825946 driver INFO    From driver eth1: CNX-MODIFY-RESP
   (2, ACCEPT)
1130258148.826873 client INFO    To client 10: CNX-MODIFY-RESP(2,
   ACCEPT)
1130258148.827948 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=550000)
1130258148.830212 driver INFO    From driver eth1: CNX-MODIFY-RESP
   (3, ACCEPT)
1130258148.831245 client INFO    To client 10: CNX-MODIFY-RESP(3,
   ACCEPT)
1130258148.993305 proto  INFO    Recv from eth1: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:C0:DF:25:ED:18>, BW=550000, MAX=1e+06)
1130258148.995028 driver INFO    To driver eth1: RESOURCE-QUERY()
1130258148.997765 proto  INFO    Recv from eth1: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:C0:DF:25:ED:18>, BW=550000, MAX=1e+06)
1130258148.999770 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=550000)
1130258149.001095 client INFO    To client 10: RESOURCE-INDICATION
   (target=<client.L2Addr (interface 0)/00:C0:DF:25:ED:18>, BW
   =550000)
1130258151.346851 client INFO    From client 10: CNX-DEACTIVATE(1)
1130258151.347706 root   INFO    Connection 1 deactivated
1130258151.348825 proto  INFO    Sending through eth1: CNX-
   DEACTIVATE(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr
   /Ethernet 00:50:BA:C7:F6:A8>, CnxID=1)
1130258151.351341 driver INFO    To driver eth1: CNX-DEACTIVATE(1)
1130258151.352648 client INFO    From client 10: CNX-DEACTIVATE(2)
1130258151.353452 root   INFO    Connection 2 deactivated
1130258151.354565 proto  INFO    Sending through eth1: CNX-
   DEACTIVATE(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr
   /Ethernet 00:50:BA:C7:F6:A8>, CnxID=2)
1130258151.356082 driver INFO    To driver eth1: CNX-DEACTIVATE(2)
1130258151.357397 client INFO    From client 10: CNX-DEACTIVATE(3)
1130258151.358190 root   INFO    Connection 3 deactivated
1130258151.361493 proto  INFO    Sending through eth1: CNX-
   DEACTIVATE(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr
   /Ethernet 00:50:BA:C7:F6:A8>, CnxID=3)
1130258151.363307 driver INFO    To driver eth1: CNX-DEACTIVATE(3)
1130258151.366015 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=625000)
1130258151.367467 client INFO    To client 10: RESOURCE-INDICATION
   (target=<client.L2Addr (interface 0)/00:C0:DF:25:ED:18>, BW
   =550000)
1130258151.371779 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=775000)
```

```
1130258151.373329 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=1e+06)
1130258151.378587 proto  INFO    Recv from eth1: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:C0:DF:25:ED:18>, BW=625000, MAX=1e+06)
1130258151.380335 driver INFO    To driver eth1: RESOURCE-QUERY()
1130258151.382726 proto  INFO    Recv from eth1: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:C0:DF:25:ED:18>, BW=625000, MAX=1e+06)
1130258151.384687 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=1e+06)
1130258151.385954 client INFO    To client 10: RESOURCE-INDICATION
   (target=<client.L2Addr (interface 0)/00:C0:DF:25:ED:18>, BW
   =625000)
1130258151.580969 proto  INFO    Recv from eth1: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:C0:DF:25:ED:18>, BW=1e+06, MAX=1e+06)
1130258151.582325 driver INFO    To driver eth1: RESOURCE-QUERY()
1130258151.583406 proto  INFO    Recv from eth1: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:C0:DF:25:ED:18>, BW=1e+06, MAX=1e+06)
1130258151.586226 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=1e+06)
1130258151.587508 client INFO    To client 10: RESOURCE-INDICATION
   (target=<client.L2Addr (interface 0)/00:C0:DF:25:ED:18>, BW=1e
   +06)
1130258155.027741 proto  INFO    Recv from eth1: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
```

A.1.2 Access Point

```
1130258127.605269 driver INFO    From driver ???: REGISTER(ifname=
   eth1, bwmax=1e+06)
1130258127.606040 driver INFO    Driver registered to handle
   netdevice eth1
1130258135.001641 proto  INFO    Sending through br0: ANNOUNCE(dst
   =<L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet
   00:50:BA:2B:A7:48>, nodelist=[])
1130258140.892208 proto  INFO    Recv from ebtables(eth0 -> ):
   RESOURCE-QUERY(dst=<L2Addr/Ethernet 00:C0:DF:25:ED:18>, src=<
   L2Addr/Ethernet 00:50:BA:C7:F6:A8>)
1130258140.893419 driver INFO    To driver eth0: RESOURCE-QUERY()
1130258140.894520 driver INFO    To driver eth1: RESOURCE-QUERY()
1130258140.897410 driver INFO    From driver eth0: RESOURCE-
   INDICATION(BW=1e+06)
1130258140.898797 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=1e+06)
```

```
1130258140.899986 proto  INFO    Sending through br0: RESOURCE-
    INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
    /Ethernet 00:C0:DF:25:ED:18>, BW=1e+06, MAX=1e+06)
1130258145.003036 proto  INFO    Sending through br0: ANNOUNCE(dst
    =<L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet
    00:50:BA:2B:A7:48>, nodelist=[])
1130258145.836190 proto  INFO    Recv from ebtables(eth0 -> eth1):
     CNX-ACTIVATE-REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src
    =<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, CnxID=1, tx=(TSpec
    (100000, 12000, 100000, 12000, 512), RSpec(0, 100000)), rx=(
    TSpec(0, 12000, 0, 12000, 512), RSpec(0, 0)), prio=0, data_len
    =0)
1130258145.837966 proto  INFO    Sending through br0: CNX-ACTIVATE
    -REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr/
    Ethernet 00:50:BA:C7:F6:A8>, CnxID=1, tx=(TSpec(100000, 12000,
    100000, 12000, 512), RSpec(0, 100000)), rx=(TSpec(0, 12000, 0,
    12000, 512), RSpec(0, 0)), prio=0, data_len=0)
1130258145.841422 proto  INFO    Recv from ebtables(eth0 -> eth1):
     CNX-ACTIVATE-REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src
    =<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, CnxID=2, tx=(TSpec(0,
    12000, 0, 12000, 512), RSpec(0, 0)), rx=(TSpec(200000, 12000,
    200000, 12000, 512), RSpec(0, 200000)), prio=0, data_len=0)
1130258145.843358 proto  INFO    Sending through br0: CNX-ACTIVATE
    -REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr/
    Ethernet 00:50:BA:C7:F6:A8>, CnxID=2, tx=(TSpec(0, 12000, 0,
    12000, 512), RSpec(0, 0)), rx=(TSpec(200000, 12000, 200000,
    12000, 512), RSpec(0, 200000)), prio=0, data_len=0)
1130258145.846511 proto  INFO    Recv from ebtables(eth0 -> eth1):
     CNX-ACTIVATE-REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src
    =<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, CnxID=3, tx=(TSpec
    (100000, 12000, 100000, 12000, 512), RSpec(0, 100000)), rx=(
    TSpec(200000, 12000, 200000, 12000, 512), RSpec(0, 200000)),
    prio=0, data_len=0)
1130258145.848478 proto  INFO    Sending through br0: CNX-ACTIVATE
    -REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr/
    Ethernet 00:50:BA:C7:F6:A8>, CnxID=3, tx=(TSpec(100000, 12000,
    100000, 12000, 512), RSpec(0, 100000)), rx=(TSpec(200000,
    12000, 200000, 12000, 512), RSpec(0, 200000)), prio=0, data_len
    =0)
1130258145.850593 proto  INFO    Recv from ebtables(eth1 -> eth0):
     CNX-ACTIVATE-RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src
    =<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, CnxID=1, result=ACCEPT)
1130258145.852514 driver INFO    To driver eth1: CNX-ACTIVATE-REQ(
    global_cnxid=1, tx=(TSpec(100000, 12000, 100000, 12000, 512),
    RSpec(0, 100000)), rx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0,
     0)), prio=0)
1130258145.855991 driver INFO    To driver eth0: CNX-ACTIVATE-REQ(
    global_cnxid=1, tx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0, 0)
    ), rx=(TSpec(100000, 12000, 100000, 12000, 512), RSpec(0,
```

```
    100000)), prio=0)
1130258145.859438 proto  INFO    Recv from ebtables(eth1 -> eth0):
     CNX-ACTIVATE-RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src
    =<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, CnxID=2, result=ACCEPT)
1130258145.861855 driver INFO    From driver eth0: RESOURCE-
    INDICATION(BW=900000)
1130258145.863212 proto  INFO    Sending through br0: RESOURCE-
    INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
    /Ethernet 00:C0:DF:25:ED:18>, BW=900000, MAX=1e+06)
1130258145.864974 driver INFO    From driver eth1: RESOURCE-
    INDICATION(BW=900000)
1130258145.866284 proto  INFO    Sending through br0: RESOURCE-
    INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
    /Ethernet 00:C0:DF:25:ED:18>, BW=900000, MAX=1e+06)
1130258145.868106 proto  INFO    Recv from ebtables(eth1 -> eth0):
     CNX-ACTIVATE-RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src
    =<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, CnxID=3, result=ACCEPT)
1130258145.870528 driver INFO    From driver eth0: CNX-ACTIVATE-
    RESP(1, ACCEPT)
1130258145.871693 driver INFO    To driver eth0: CNX-ACTIVATE-REQ(
    global_cnxid=2, tx=(TSpec(200000, 12000, 200000, 12000, 512),
    RSpec(0, 200000)), rx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0,
     0)), prio=0)
1130258145.874974 driver INFO    From driver eth1: CNX-ACTIVATE-
    RESP(1, ACCEPT)
1130258145.876282 proto  INFO    Sending through br0: CNX-ACTIVATE
    -RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr/
    Ethernet 00:C0:DF:E6:EC:45>, CnxID=1, result=ACCEPT)
1130258145.877811 driver INFO    To driver eth1: CNX-ACTIVATE-REQ(
    global_cnxid=2, tx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0, 0)
    ), rx=(TSpec(200000, 12000, 200000, 12000, 512), RSpec(0,
    200000)), prio=0)
1130258145.881355 driver INFO    From driver eth0: RESOURCE-
    INDICATION(BW=700000)
1130258145.882615 driver INFO    From driver eth1: RESOURCE-
    INDICATION(BW=700000)
1130258145.883942 driver INFO    From driver eth0: CNX-ACTIVATE-
    RESP(2, ACCEPT)
1130258145.884982 driver INFO    To driver eth0: CNX-ACTIVATE-REQ(
    global_cnxid=3, tx=(TSpec(200000, 12000, 200000, 12000, 512),
    RSpec(0, 200000)), rx=(TSpec(100000, 12000, 100000, 12000, 512)
    , RSpec(0, 100000)), prio=0)
1130258145.888329 driver INFO    From driver eth1: CNX-ACTIVATE-
    RESP(2, ACCEPT)
1130258145.889544 proto  INFO    Sending through br0: CNX-ACTIVATE
    -RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr/
    Ethernet 00:C0:DF:E6:EC:45>, CnxID=2, result=ACCEPT)
1130258145.891123 driver INFO    To driver eth1: CNX-ACTIVATE-REQ(
    global_cnxid=3, tx=(TSpec(100000, 12000, 100000, 12000, 512),
```

```
   RSpec(0, 100000)), rx=(TSpec(200000, 12000, 200000, 12000, 512)
   , RSpec(0, 200000)), prio=0)
1130258145.894827 driver INFO    From driver eth0: RESOURCE-
   INDICATION(BW=400000)
1130258145.896183 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=400000)
1130258145.897531 driver INFO    From driver eth0: CNX-ACTIVATE-
   RESP(3, ACCEPT)
1130258145.898964 driver INFO    From driver eth1: CNX-ACTIVATE-
   RESP(3, ACCEPT)
1130258145.900175 proto  INFO    Sending through br0: CNX-ACTIVATE
   -RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr/
   Ethernet 00:C0:DF:E6:EC:45>, CnxID=3, result=ACCEPT)
1130258146.065385 proto  INFO    Sending through br0: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:C0:DF:25:ED:18>, BW=400000, MAX=1e+06)
1130258146.068450 proto  INFO    Sending through br0: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:C0:DF:25:ED:18>, BW=400000, MAX=1e+06)
1130258148.739483 proto  INFO    Recv from ebtables(eth0 -> eth1):
    CNX-ACTIVATE-REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src
   =<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, CnxID=1, tx=(TSpec(75000,
    12000, 75000, 12000, 512), RSpec(0, 75000)), rx=(TSpec(0,
   12000, 0, 12000, 512), RSpec(0, 0)), prio=0, data_len=0)
1130258148.741100 proto  INFO    Sending through br0: CNX-ACTIVATE
   -REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr/
   Ethernet 00:50:BA:C7:F6:A8>, CnxID=1, tx=(TSpec(75000, 12000,
   75000, 12000, 512), RSpec(0, 75000)), rx=(TSpec(0, 12000, 0,
   12000, 512), RSpec(0, 0)), prio=0, data_len=0)
1130258148.743234 proto  INFO    Recv from ebtables(eth0 -> eth1):
    CNX-ACTIVATE-REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src
   =<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, CnxID=2, tx=(TSpec(0,
   12000, 0, 12000, 512), RSpec(0, 0)), rx=(TSpec(150000, 12000,
   150000, 12000, 512), RSpec(0, 150000)), prio=0, data_len=0)
1130258148.744809 proto  INFO    Sending through br0: CNX-ACTIVATE
   -REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr/
   Ethernet 00:50:BA:C7:F6:A8>, CnxID=2, tx=(TSpec(0, 12000, 0,
   12000, 512), RSpec(0, 0)), rx=(TSpec(150000, 12000, 150000,
   12000, 512), RSpec(0, 150000)), prio=0, data_len=0)
1130258148.747901 proto  INFO    Recv from ebtables(eth0 -> eth1):
    CNX-ACTIVATE-REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src
   =<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, CnxID=3, tx=(TSpec(75000,
    12000, 75000, 12000, 512), RSpec(0, 75000)), rx=(TSpec(150000,
    12000, 150000, 12000, 512), RSpec(0, 150000)), prio=0,
   data_len=0)
1130258148.749800 proto  INFO    Sending through br0: CNX-ACTIVATE
   -REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr/
   Ethernet 00:50:BA:C7:F6:A8>, CnxID=3, tx=(TSpec(75000, 12000,
   75000, 12000, 512), RSpec(0, 75000)), rx=(TSpec(150000, 12000,
```

```
   150000, 12000, 512), RSpec(0, 150000)), prio=0, data_len=0)
1130258148.751795 proto  INFO    Recv from ebtables(eth1 -> eth0):
    CNX-ACTIVATE-RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src
   =<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, CnxID=1, result=ACCEPT)
1130258148.753437 driver INFO    To driver eth0: CNX-MODIFY-REQ(
   CnxID=1, tx=(TSpec(75000, 12000, 75000, 12000, 512), RSpec(0,
   75000)), rx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0, 0)))
1130258148.756292 driver INFO    To driver eth1: CNX-MODIFY-REQ(
   CnxID=1, tx=(TSpec(75000, 12000, 75000, 12000, 512), RSpec(0,
   75000)), rx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0, 0)))
1130258148.759582 proto  INFO    Recv from ebtables(eth1 -> eth0):
    CNX-ACTIVATE-RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src
   =<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, CnxID=2, result=ACCEPT)
1130258148.761183 driver INFO    To driver eth0: CNX-MODIFY-REQ(
   CnxID=2, tx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0, 0)), rx=(
   TSpec(150000, 12000, 150000, 12000, 512), RSpec(0, 150000)))
1130258148.764045 driver INFO    To driver eth1: CNX-MODIFY-REQ(
   CnxID=2, tx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0, 0)), rx=(
   TSpec(150000, 12000, 150000, 12000, 512), RSpec(0, 150000)))
1130258148.767103 driver INFO    From driver eth0: RESOURCE-
   INDICATION(BW=425000)
1130258148.768294 proto  INFO    Sending through br0: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:C0:DF:25:ED:18>, BW=425000, MAX=1e+06)
1130258148.769975 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=425000)
1130258148.771136 proto  INFO    Sending through br0: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:C0:DF:25:ED:18>, BW=425000, MAX=1e+06)
1130258148.772854 proto  INFO    Recv from ebtables(eth1 -> eth0):
    CNX-ACTIVATE-RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src
   =<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, CnxID=3, result=ACCEPT)
1130258148.774318 driver INFO    To driver eth0: CNX-MODIFY-REQ(
   CnxID=3, tx=(TSpec(75000, 12000, 75000, 12000, 512), RSpec(0,
   75000)), rx=(TSpec(150000, 12000, 150000, 12000, 512), RSpec(0,
    150000)))
1130258148.777088 driver INFO    To driver eth1: CNX-MODIFY-REQ(
   CnxID=3, tx=(TSpec(75000, 12000, 75000, 12000, 512), RSpec(0,
   75000)), rx=(TSpec(150000, 12000, 150000, 12000, 512), RSpec(0,
    150000)))
1130258148.780248 driver INFO    From driver eth0: CNX-MODIFY-RESP
   (1, ACCEPT)
1130258148.781500 driver INFO    From driver eth1: CNX-MODIFY-RESP
   (1, ACCEPT)
1130258148.782597 proto  INFO    Sending through br0: CNX-ACTIVATE
   -RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr/
   Ethernet 00:C0:DF:E6:EC:45>, CnxID=1, result=ACCEPT)
1130258148.784261 driver INFO    From driver eth0: RESOURCE-
   INDICATION(BW=475000)
```

```
1130258148.785430 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=475000)
1130258148.786661 driver INFO    From driver eth0: CNX-MODIFY-RESP
   (2, ACCEPT)
1130258148.787913 driver INFO    From driver eth1: CNX-MODIFY-RESP
   (2, ACCEPT)
1130258148.789015 proto  INFO    Sending through br0: CNX-ACTIVATE
   -RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr/
   Ethernet 00:C0:DF:E6:EC:45>, CnxID=2, result=ACCEPT)
1130258148.790634 driver INFO    From driver eth0: RESOURCE-
   INDICATION(BW=550000)
1130258148.791808 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=550000)
1130258148.793032 driver INFO    From driver eth0: CNX-MODIFY-RESP
   (3, ACCEPT)
1130258148.794454 driver INFO    From driver eth1: CNX-MODIFY-RESP
   (3, ACCEPT)
1130258148.795845 proto  INFO    Sending through br0: CNX-ACTIVATE
   -RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr/
   Ethernet 00:C0:DF:E6:EC:45>, CnxID=3, result=ACCEPT)
1130258148.970069 proto  INFO    Sending through br0: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:C0:DF:25:ED:18>, BW=550000, MAX=1e+06)
1130258148.973114 proto  INFO    Sending through br0: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:C0:DF:25:ED:18>, BW=550000, MAX=1e+06)
1130258151.329726 proto  INFO    Recv from ebtables(eth0 -> eth1):
    CNX-DEACTIVATE(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<
   L2Addr/Ethernet 00:50:BA:C7:F6:A8>, CnxID=1)
1130258151.330851 root   INFO    Connection 1 deactivated
1130258151.331508 proto  INFO    Sending through br0: CNX-
   DEACTIVATE(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr
   /Ethernet 00:50:BA:C7:F6:A8>, CnxID=1)
1130258151.332950 driver INFO    To driver eth0: CNX-DEACTIVATE(1)
1130258151.333901 driver INFO    To driver eth1: CNX-DEACTIVATE(1)
1130258151.335072 proto  INFO    Recv from ebtables(eth0 -> eth1):
    CNX-DEACTIVATE(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<
   L2Addr/Ethernet 00:50:BA:C7:F6:A8>, CnxID=2)
1130258151.336128 root   INFO    Connection 2 deactivated
1130258151.336777 proto  INFO    Sending through br0: CNX-
   DEACTIVATE(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr
   /Ethernet 00:50:BA:C7:F6:A8>, CnxID=2)
1130258151.340448 driver INFO    To driver eth0: CNX-DEACTIVATE(2)
1130258151.342690 driver INFO    To driver eth1: CNX-DEACTIVATE(2)
1130258151.345130 proto  INFO    Recv from ebtables(eth0 -> eth1):
    CNX-DEACTIVATE(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<
   L2Addr/Ethernet 00:50:BA:C7:F6:A8>, CnxID=3)
1130258151.346577 root   INFO    Connection 3 deactivated
```

```
1130258151.347606 proto  INFO    Sending through br0: CNX-
   DEACTIVATE(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr
   /Ethernet 00:50:BA:C7:F6:A8>, CnxID=3)
1130258151.349171 driver INFO    To driver eth0: CNX-DEACTIVATE(3)
1130258151.351371 driver INFO    To driver eth1: CNX-DEACTIVATE(3)
1130258151.353821 driver INFO    From driver eth0: RESOURCE-
   INDICATION(BW=625000)
1130258151.355124 proto  INFO    Sending through br0: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:C0:DF:25:ED:18>, BW=625000, MAX=1e+06)
1130258151.356883 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=625000)
1130258151.358118 proto  INFO    Sending through br0: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:C0:DF:25:ED:18>, BW=625000, MAX=1e+06)
1130258151.359924 driver INFO    From driver eth0: RESOURCE-
   INDICATION(BW=775000)
1130258151.361105 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=775000)
1130258151.362448 driver INFO    From driver eth0: RESOURCE-
   INDICATION(BW=1e+06)
1130258151.363616 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=1e+06)
1130258151.557796 proto  INFO    Sending through br0: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:C0:DF:25:ED:18>, BW=1e+06, MAX=1e+06)
1130258151.559839 proto  INFO    Sending through br0: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:C0:DF:25:ED:18>, BW=1e+06, MAX=1e+06)
1130258155.004385 proto  INFO    Sending through br0: ANNOUNCE(dst
   =<L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet
   00:50:BA:2B:A7:48>, nodelist=[])
```

A.1.3 Mobile Node

```
1130258130.577453 driver INFO    From driver ???: REGISTER(ifname=
   eth0, bwmax=1e+06)
1130258130.578025 driver INFO    Driver registered to handle
   netdevice eth0
1130258134.999591 proto  INFO    Recv from eth0: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
1130258145.000912 proto  INFO    Recv from eth0: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
1130258145.836564 proto  INFO    Recv from eth0: CNX-ACTIVATE-REQ(
   dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr/Ethernet
   00:50:BA:C7:F6:A8>, CnxID=1, tx=(TSpec(100000, 12000, 100000,
```

```
   12000, 512), RSpec(0, 100000)), rx=(TSpec(0, 12000, 0, 12000,
   512), RSpec(0, 0)), prio=0, data_len=0)
1130258145.838077 driver INFO    To driver eth0: CNX-ACTIVATE-REQ(
   global_cnxid=1, tx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0, 0)
   ), rx=(TSpec(100000, 12000, 100000, 12000, 512), RSpec(0,
   100000)), prio=0)
1130258145.838740 client INFO    To client 8: CNX-INDICATION(CnxID
   =1, tx=(TSpec(100000, 12000, 100000, 12000, 512), RSpec(0,
   100000)), rx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0, 0)), ber
   =-1.000000e+00, data_len=0)
1130258145.840663 driver INFO    From driver eth0: RESOURCE-
   INDICATION(BW=900000)
1130258145.841316 driver INFO    From driver eth0: CNX-ACTIVATE-
   RESP(1, ACCEPT)
1130258145.841977 proto  INFO    Sending through eth0: CNX-
   ACTIVATE-RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<
   L2Addr/Ethernet 00:C0:DF:E6:EC:45>, CnxID=1, result=ACCEPT)
1130258145.842937 proto  INFO    Recv from eth0: CNX-ACTIVATE-REQ(
   dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr/Ethernet
   00:50:BA:C7:F6:A8>, CnxID=2, tx=(TSpec(0, 12000, 0, 12000, 512)
   , RSpec(0, 0)), rx=(TSpec(200000, 12000, 200000, 12000, 512),
   RSpec(0, 200000)), prio=0, data_len=0)
1130258145.844201 driver INFO    To driver eth0: CNX-ACTIVATE-REQ(
   global_cnxid=2, tx=(TSpec(200000, 12000, 200000, 12000, 512),
   RSpec(0, 200000)), rx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0,
    0)), prio=0)
1130258145.845810 client INFO    To client 8: CNX-INDICATION(CnxID
   =2, tx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0, 0)), rx=(TSpec
   (200000, 12000, 200000, 12000, 512), RSpec(0, 200000)), ber
   =-1.000000e+00, data_len=0)
1130258145.846877 driver INFO    From driver eth0: RESOURCE-
   INDICATION(BW=700000)
1130258145.847605 proto  INFO    Recv from eth0: CNX-ACTIVATE-REQ(
   dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr/Ethernet
   00:50:BA:C7:F6:A8>, CnxID=3, tx=(TSpec(100000, 12000, 100000,
   12000, 512), RSpec(0, 100000)), rx=(TSpec(200000, 12000,
   200000, 12000, 512), RSpec(0, 200000)), prio=0, data_len=0)
1130258145.848945 client INFO    To client 8: CNX-INDICATION(CnxID
   =3, tx=(TSpec(100000, 12000, 100000, 12000, 512), RSpec(0,
   100000)), rx=(TSpec(200000, 12000, 200000, 12000, 512), RSpec
   (0, 200000)), ber=-1.000000e+00, data_len=0)
1130258145.849912 driver INFO    From driver eth0: CNX-ACTIVATE-
   RESP(2, ACCEPT)
1130258145.850432 proto  INFO    Sending through eth0: CNX-
   ACTIVATE-RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<
   L2Addr/Ethernet 00:C0:DF:E6:EC:45>, CnxID=2, result=ACCEPT)
1130258145.851271 driver INFO    To driver eth0: CNX-ACTIVATE-REQ(
   global_cnxid=3, tx=(TSpec(200000, 12000, 200000, 12000, 512),
   RSpec(0, 200000)), rx=(TSpec(100000, 12000, 100000, 12000, 512)
```

```
   , RSpec(0, 100000)), prio=0)
1130258145.852915 driver INFO    From driver eth0: RESOURCE-
   INDICATION(BW=400000)
1130258145.853525 driver INFO    From driver eth0: CNX-ACTIVATE-
   RESP(3, ACCEPT)
1130258145.854063 proto  INFO    Sending through eth0: CNX-
   ACTIVATE-RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<
   L2Addr/Ethernet 00:C0:DF:E6:EC:45>, CnxID=3, result=ACCEPT)
1130258148.739625 proto  INFO    Recv from eth0: CNX-ACTIVATE-REQ(
   dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr/Ethernet
   00:50:BA:C7:F6:A8>, CnxID=1, tx=(TSpec(75000, 12000, 75000,
   12000, 512), RSpec(0, 75000)), rx=(TSpec(0, 12000, 0, 12000,
   512), RSpec(0, 0)), prio=0, data_len=0)
1130258148.740712 driver INFO    To driver eth0: CNX-MODIFY-REQ(
   CnxID=1, tx=(TSpec(75000, 12000, 75000, 12000, 512), RSpec(0,
   75000)), rx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0, 0)))
1130258148.742394 driver INFO    From driver eth0: RESOURCE-
   INDICATION(BW=425000)
1130258148.743197 proto  INFO    Recv from eth0: CNX-ACTIVATE-REQ(
   dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr/Ethernet
   00:50:BA:C7:F6:A8>, CnxID=2, tx=(TSpec(0, 12000, 0, 12000, 512)
   , RSpec(0, 0)), rx=(TSpec(150000, 12000, 150000, 12000, 512),
   RSpec(0, 150000)), prio=0, data_len=0)
1130258148.744326 driver INFO    To driver eth0: CNX-MODIFY-REQ(
   CnxID=2, tx=(TSpec(0, 12000, 0, 12000, 512), RSpec(0, 0)), rx=(
   TSpec(150000, 12000, 150000, 12000, 512), RSpec(0, 150000)))
1130258148.745884 driver INFO    From driver eth0: CNX-MODIFY-RESP
   (1, ACCEPT)
1130258148.746437 proto  INFO    Sending through eth0: CNX-
   ACTIVATE-RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<
   L2Addr/Ethernet 00:C0:DF:E6:EC:45>, CnxID=1, result=ACCEPT)
1130258148.747334 driver INFO    From driver eth0: RESOURCE-
   INDICATION(BW=475000)
1130258148.747887 driver INFO    From driver eth0: CNX-MODIFY-RESP
   (2, ACCEPT)
1130258148.748495 proto  INFO    Sending through eth0: CNX-
   ACTIVATE-RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<
   L2Addr/Ethernet 00:C0:DF:E6:EC:45>, CnxID=2, result=ACCEPT)
1130258148.749393 proto  INFO    Recv from eth0: CNX-ACTIVATE-REQ(
   dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr/Ethernet
   00:50:BA:C7:F6:A8>, CnxID=3, tx=(TSpec(75000, 12000, 75000,
   12000, 512), RSpec(0, 75000)), rx=(TSpec(150000, 12000, 150000,
    12000, 512), RSpec(0, 150000)), prio=0, data_len=0)
1130258148.750485 driver INFO    To driver eth0: CNX-MODIFY-REQ(
   CnxID=3, tx=(TSpec(75000, 12000, 75000, 12000, 512), RSpec(0,
   75000)), rx=(TSpec(150000, 12000, 150000, 12000, 512), RSpec(0,
    150000)))
1130258148.752056 driver INFO    From driver eth0: RESOURCE-
   INDICATION(BW=550000)
```

```
1130258148.752652 driver INFO    From driver eth0: CNX-MODIFY-RESP
   (3, ACCEPT)
1130258148.753194 proto  INFO    Sending through eth0: CNX-
   ACTIVATE-RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<
   L2Addr/Ethernet 00:C0:DF:E6:EC:45>, CnxID=3, result=ACCEPT)
1130258151.329277 proto  INFO    Recv from eth0: CNX-DEACTIVATE(
   dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr/Ethernet
   00:50:BA:C7:F6:A8>, CnxID=1)
1130258151.329942 root   INFO    Connection 1 deactivated
1130258151.330289 client INFO    To client 8: CNX-ACTIVATE-RESP(1,
    REJECT)
1130258151.330694 driver INFO    To driver eth0: CNX-DEACTIVATE(1)
1130258151.331950 driver INFO    From driver eth0: RESOURCE-
   INDICATION(BW=625000)
1130258151.336900 proto  INFO    Recv from eth0: CNX-DEACTIVATE(
   dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr/Ethernet
   00:50:BA:C7:F6:A8>, CnxID=2)
1130258151.337462 root   INFO    Connection 2 deactivated
1130258151.337815 client INFO    To client 8: CNX-ACTIVATE-RESP(2,
    REJECT)
1130258151.338198 driver INFO    To driver eth0: CNX-DEACTIVATE(2)
1130258151.339168 driver INFO    From driver eth0: RESOURCE-
   INDICATION(BW=775000)
1130258151.345652 proto  INFO    Recv from eth0: CNX-DEACTIVATE(
   dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr/Ethernet
   00:50:BA:C7:F6:A8>, CnxID=3)
1130258151.346233 root   INFO    Connection 3 deactivated
1130258151.346573 client INFO    To client 8: CNX-ACTIVATE-RESP(3,
    REJECT)
1130258151.346967 driver INFO    To driver eth0: CNX-DEACTIVATE(3)
1130258151.347942 driver INFO    From driver eth0: RESOURCE-
   INDICATION(BW=1e+06)
1130258155.002274 proto  INFO    Recv from eth0: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
```

A.2 Resource degradation and autodiscovery of APs

A.2.1 Access Router

```
1130349947.328791 client INFO    From PA client 10: <client.
   ResourceQueryList object at 0xb79d81ec>
1130349947.329707 client INFO    From client 10: <client.
   ResourceQueryList object at 0xb79d81ec>
1130349947.331304 proto  INFO    Sending through eth1: RESOURCE-
   QUERY(dst=<L2Addr/Ethernet 00:50:BA:2B:A7:48>, src=<L2Addr/
   Ethernet 00:50:BA:C7:F6:A8>)
```

```
1130349947.341828 proto  INFO    Recv from eth1: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:50:BA:2B:A7:48>, BW=1e+06, MAX=1e+06)
1130349947.343481 driver INFO    To driver eth1: RESOURCE-QUERY()
1130349947.345702 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=1e+06)
1130349951.297905 proto  INFO    Recv from eth1: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
1130349961.298932 proto  INFO    Recv from eth1: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
1130349966.351331 proto  INFO    Recv from eth1: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:50:BA:2B:A7:48>, BW=500000, MAX=1e+06)
1130349966.352706 driver INFO    To driver eth1: RESOURCE-QUERY()
1130349966.354713 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=1e+06)
1130349971.299941 proto  INFO    Recv from eth1: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
1130349981.301000 proto  INFO    Recv from eth1: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
1130349982.169728 proto  INFO    Recv from eth1: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:50:BA:2B:A7:48>, BW=100000, MAX=1e+06)
1130349982.171063 driver INFO    To driver eth1: RESOURCE-QUERY()
1130349982.173061 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=1e+06)
1130349982.174104 client INFO    To client 10: RESOURCE-
   DEGRADATION([(<client.L2Addr eth1/00:50:BA:2B:A7:48>, 100000)])
1130349991.302011 proto  INFO    Recv from eth1: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
```

A.2.2 Access Point

```
1130349919.295589 driver INFO    From driver ???: REGISTER(ifname=
   eth1, bwmax=1e+06)
1130349919.296338 driver INFO    Driver registered to handle
   netdevice eth1
1130349921.667103 proto  INFO    Sending through br0: ANNOUNCE(dst
   =<L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet
   00:50:BA:2B:A7:48>, nodelist=[])
1130349931.668457 proto  INFO    Sending through br0: ANNOUNCE(dst
   =<L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet
   00:50:BA:2B:A7:48>, nodelist=[])
```

```
1130349941.670329 proto  INFO    Sending through br0: ANNOUNCE(dst
   =<L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet
   00:50:BA:2B:A7:48>, nodelist=[])
1130349947.708034 proto  INFO    Recv from ebtables(eth0 -> ):
   RESOURCE-QUERY(dst=<L2Addr/Ethernet 00:50:BA:2B:A7:48>, src=<
   L2Addr/Ethernet 00:50:BA:C7:F6:A8>)
1130349947.709248 driver INFO    To driver eth0: RESOURCE-QUERY()
1130349947.709922 driver INFO    To driver eth1: RESOURCE-QUERY()
1130349947.712791 driver INFO    From driver eth0: RESOURCE-
   INDICATION(BW=1e+06)
1130349947.714147 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=1e+06)
1130349947.715324 proto  INFO    Sending through br0: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:50:BA:2B:A7:48>, BW=1e+06, MAX=1e+06)
1130349951.671705 proto  INFO    Sending through br0: ANNOUNCE(dst
   =<L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet
   00:50:BA:2B:A7:48>, nodelist=[])
1130349961.673097 proto  INFO    Sending through br0: ANNOUNCE(dst
   =<L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet
   00:50:BA:2B:A7:48>, nodelist=[])
1130349966.724712 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=500000)
1130349966.725699 proto  INFO    Sending through br0: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:50:BA:2B:A7:48>, BW=500000, MAX=1e+06)
1130349971.674472 proto  INFO    Sending through br0: ANNOUNCE(dst
   =<L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet
   00:50:BA:2B:A7:48>, nodelist=[])
1130349981.675874 proto  INFO    Sending through br0: ANNOUNCE(dst
   =<L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet
   00:50:BA:2B:A7:48>, nodelist=[])
1130349982.543773 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=100000)
1130349982.544738 proto  INFO    Sending through br0: RESOURCE-
   INDICATION(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr
   /Ethernet 00:50:BA:2B:A7:48>, BW=100000, MAX=1e+06)
1130349991.677262 proto  INFO    Sending through br0: ANNOUNCE(dst
   =<L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet
   00:50:BA:2B:A7:48>, nodelist=[])
```

A.2.3 Mobile Node

```
1130349924.322551 driver INFO    From driver ???: REGISTER(ifname=
   eth0, bwmax=1e+06)
1130349924.323138 driver INFO    Driver registered to handle
   netdevice eth0
```

```
1130349931.017902 proto  INFO    Recv from eth0: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
1130349941.017292 proto  INFO    Recv from eth0: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
1130349951.016103 proto  INFO    Recv from eth0: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
1130349961.014949 proto  INFO    Recv from eth0: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
1130349971.013677 proto  INFO    Recv from eth0: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
1130349981.012424 proto  INFO    Recv from eth0: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
1130349991.011125 proto  INFO    Recv from eth0: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
```

A.3 Handover preparation and execution

A.3.1 Access Router

```
1130324044.661775 driver INFO    From driver ???: REGISTER(ifname=
   eth1, bwmax=1e+06)
1130324044.662649 driver INFO    Driver registered to handle
   netdevice eth1
1130324045.158308 proto  INFO    Recv from eth1: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
1130324055.160153 proto  INFO    Recv from eth1: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
1130324057.336959 proto  INFO    Recv from eth1: HO-EXECUTE(dst=<
   L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr/Ethernet 00:C0:
   DF:E6:EC:45>, serving_node=<L2Addr/None >)
1130324057.339015 proto  INFO    Sending through eth1: CNX-
   ACTIVATE-REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<
   L2Addr/Ethernet 00:50:BA:C7:F6:A8>, CnxID=1, tx=(TSpec(200000,
   12000, 200000, 12000, 512), RSpec(1, 200000)), rx=(TSpec
   (200000, 12000, 200000, 12000, 512), RSpec(1, 200000)), prio=0,
    data_len=0)
1130324057.359920 proto  INFO    Recv from eth1: CNX-ACTIVATE-RESP
   (dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr/Ethernet
```

```
    00:C0:DF:E6:EC:45>, CnxID=1, result=ACCEPT)
1130324057.361610 driver INFO    To driver eth1: CNX-ACTIVATE-REQ(
    global_cnxid=1, tx=(TSpec(200000, 12000, 200000, 12000, 512),
    RSpec(1, 200000)), rx=(TSpec(200000, 12000, 200000, 12000, 512)
    , RSpec(1, 200000)), prio=0)
1130324057.364859 driver INFO    From driver eth1: RESOURCE-
    INDICATION(BW=600000)
1130324057.365958 driver INFO    From driver eth1: CNX-ACTIVATE-
    RESP(1, ACCEPT)
1130324057.366898 client INFO    To client 8: CNX-ACTIVATE-RESP(1,
     ACCEPT)
1130324065.161555 proto  INFO    Recv from eth1: ANNOUNCE(dst=<
    L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
    BA:2B:A7:48>, nodelist=[])
```

A.3.2 Access Point

```
1130324046.463978 driver INFO    From driver ???: REGISTER(ifname=
    eth1, bwmax=1e+06)
1130324046.465049 driver INFO    Driver registered to handle
    netdevice eth1
1130324055.159415 proto  INFO    Sending through br0: ANNOUNCE(dst
    =<L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet
    00:50:BA:2B:A7:48>, nodelist=[])
1130324057.334855 proto  INFO    Recv from ebtables(eth1 -> eth0):
     HO-EXECUTE(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<
    L2Addr/Ethernet 00:C0:DF:E6:EC:45>, serving_node=<L2Addr/None
    >)
1130324057.336051 proto  INFO    Sending through br0: HO-EXECUTE(
    dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr/Ethernet
    00:C0:DF:E6:EC:45>, serving_node=<L2Addr/None >)
1130324057.341551 proto  INFO    Recv from ebtables(eth0 -> eth1):
     CNX-ACTIVATE-REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src
    =<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, CnxID=1, tx=(TSpec
    (200000, 12000, 200000, 12000, 512), RSpec(1, 200000)), rx=(
    TSpec(200000, 12000, 200000, 12000, 512), RSpec(1, 200000)),
    prio=0, data_len=0)
1130324057.343324 proto  INFO    Sending through br0: CNX-ACTIVATE
    -REQ(dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr/
    Ethernet 00:50:BA:C7:F6:A8>, CnxID=1, tx=(TSpec(200000, 12000,
    200000, 12000, 512), RSpec(1, 200000)), rx=(TSpec(200000,
    12000, 200000, 12000, 512), RSpec(1, 200000)), prio=0, data_len
    =0)
1130324057.347247 proto  INFO    Recv from ebtables(eth1 -> eth0):
     CNX-ACTIVATE-RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src
    =<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, CnxID=1, result=ACCEPT)
1130324057.348832 driver INFO    To driver eth1: CNX-ACTIVATE-REQ(
    global_cnxid=1, tx=(TSpec(200000, 12000, 200000, 12000, 512),
```

```
   RSpec(1, 200000)), rx=(TSpec(200000, 12000, 200000, 12000, 512)
   , RSpec(1, 200000)), prio=0)
1130324057.350268 driver INFO    To driver eth0: CNX-ACTIVATE-REQ(
   global_cnxid=1, tx=(TSpec(200000, 12000, 200000, 12000, 512),
   RSpec(1, 200000)), rx=(TSpec(200000, 12000, 200000, 12000, 512)
   , RSpec(1, 200000)), prio=0)
1130324057.354741 driver INFO    From driver eth0: RESOURCE-
   INDICATION(BW=600000)
1130324057.355688 driver INFO    From driver eth1: RESOURCE-
   INDICATION(BW=600000)
1130324057.357358 driver INFO    From driver eth0: CNX-ACTIVATE-
   RESP(1, ACCEPT)
1130324057.358384 driver INFO    From driver eth1: CNX-ACTIVATE-
   RESP(1, ACCEPT)
1130324057.359270 proto  INFO    Sending through br0: CNX-ACTIVATE
   -RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr/
   Ethernet 00:C0:DF:E6:EC:45>, CnxID=1, result=ACCEPT)
1130324065.160805 proto  INFO    Sending through br0: ANNOUNCE(dst
   =<L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet
   00:50:BA:2B:A7:48>, nodelist=[])
```

A.3.3 Mobile Node

```
1130324057.334997 client INFO    From client 10: HO-EXECUTE(<
   L3Addr fec0:1::1>)
1130324057.335951 proto  INFO    Sending through eth0: HO-EXECUTE(
   dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<L2Addr/Ethernet
   00:C0:DF:E6:EC:45>, serving_node=<L2Addr/None >)
1130324057.347255 proto  INFO    Recv from eth0: CNX-ACTIVATE-REQ(
   dst=<L2Addr/Ethernet 00:C0:DF:E6:EC:45>, src=<L2Addr/Ethernet
   00:50:BA:C7:F6:A8>, CnxID=1, tx=(TSpec(200000, 12000, 200000,
   12000, 512), RSpec(1, 200000)), rx=(TSpec(200000, 12000,
   200000, 12000, 512), RSpec(1, 200000)), prio=0, data_len=0)
1130324057.348432 proto  INFO    Sending through eth0: CNX-
   ACTIVATE-RESP(dst=<L2Addr/Ethernet 00:50:BA:C7:F6:A8>, src=<
   L2Addr/Ethernet 00:C0:DF:E6:EC:45>, CnxID=1, result=ACCEPT)
1130324057.349230 client INFO    To client 8: CNX-INDICATION(CnxID
   =1, tx=(TSpec(200000, 12000, 200000, 12000, 512), RSpec(1,
   200000)), rx=(TSpec(200000, 12000, 200000, 12000, 512), RSpec
   (1, 200000)), ber=-1.000000e+00, data_len=0)
1130324057.349854 client INFO    To client 10: CNX-INDICATION(
   CnxID=1, tx=(TSpec(200000, 12000, 200000, 12000, 512), RSpec(1,
    200000)), rx=(TSpec(200000, 12000, 200000, 12000, 512), RSpec
   (1, 200000)), ber=-1.000000e+00, data_len=0)
1130324065.164580 proto  INFO    Recv from eth0: ANNOUNCE(dst=<
   L2Addr/Ethernet FF:FF:FF:FF:FF:FF>, src=<L2Addr/Ethernet 00:50:
   BA:2B:A7:48>, nodelist=[])
```

Appendix B

List of Abbreviations

3GPP 3rd Generation Partnership Program

4GW 4th Generation

4GW 4th Generation Wireless

ACL Asynchronous Connection-oriented Logical

AL Abstraction Layer

AN Access Network

AP Access Point

API Application Programming Interface

APN Access Point Name

AR Access Router

ARM Advanced Routing Mechanism

ARQ Automatic Repeat reQuest

ARROWS Advanced Radio Resource Management On Wireless Services

ATM Asynchronous Transfer Mode

BE Best Effort

BER Bit Error Rate, Bit Error Ratio

BNEP Bluetooth Network Encapsulation Protocol

BS Base Station

CAN Content Adaptation Node

CARD Candidate Access Router Discovery

CFP Contention Free Period

CL Controlled Load

CMS Central Monitoring System

CN Core Network

GARP Generic Attribute Registration Protocol

GCC GNU Compiler Collection

GLL Generic Link Layer

GMRP GARP Multicast Registration Protocol

COPS Common Open Policy Service

CP Contention Period

CRC Cyclic Redundancy Check

CSMA Carrier Sense Multiple Access

DAIDALOS Designing Advanced network Interfaces for the Delivery and Administration of Location independent, Optimised personal Services

DCF Distributed Coordination Function

DSBM Distributed Subnet Bandwidth Manager

DSCP Differentiated Services CodePoint

EDCF Enhanced Distributed Coordination Function

FEC Forward Error Correction

GGSN Gateway GPRS Support Node

GMPLS Generalised Multi Protocol Label Switching

GNU GNU's Not Unix

GPRS General Packet Radio Service

GSM Global System for Mobile Communications

HCF Hybrid Coordination Function

IAL Interface Abstraction Layer

IEEE Institute of Electrical and Electronics Engineers

IETF Internet Engineering TaskForce

IIS Intelligent Interface Selection

IP Internet Protocol

IPFIX Internet Protocol Flow Information Export

IST Information Society Technologies

L2CAP Logical Link Control and Adaptation Protocol

LAN Local Area Network

LDP Label Distribution Protocol

LSP Label Switched Path

MAC Media Access Control

MBMS Multimedia Broadcast/Multicast Service

MIH Media Independent Handover

MIHF Media Independent Handover Function

MIP Mobile IP

MN Mobile Node

MPLS Multi Protocol Label Switching

MT Mobile Terminal

MTC Mobile Terminal Controller

MTU Maximum Transmission Unit

NME Network Monitoring Entity

NSAPI Network Service Access Point Identifier

NSIS Next Steps In Signalling

OSI Open Systems Interconnect

PA Performance Attendant

PBNM Policy Based Network Management

PBNMS Policy Based Network Monitoring System...

PCF Point Coordination Function

PDP Packet Data Protocol

PDU Protocol Data Unit

PM Performance Manager

QoS Quality of Service

QoSAL QoS Abstraction Layer

RAB Radio Access Bearer

RAN Radio Access Network

RFC Request For Comments

RNC Radio Network Controller

RRM Radio Resource Manager

RSVP Resource ReSerVation Protocol

RTP Real Time Protocol

SAP Service Access Point

SAR Segmentation And Reassembly

SBM Subnet Bandwidth Manager

SCO Synchronous Connection Oriented

SDL Specification and Description Language

SDU Service Data Unit

SGSN Serving GPRS Support Node

SIG Special Intererst Group

SIP Session Initiation Protocol

SLA Service Level Agreement

SLS Service Level Specification

TBF Token Bucket Filter

TCP Transmission Control Protocol

TCS Traffic Conditioning Specification

TE Terminal Equipment, Traffic Engineering

TFT Traffic Flow Template

TOS Type Of Service

UML Unified Modelling Language

UMTS Universal Mobile Telecommunications System

VLAN Virtual LAN

VPN Virtual Private Network

WLAN Wireless Local Area Network

WMAN Wireless Metropolitan Area Network

WP Work Package

Bibliography

[3GP05] 3GPP. Multimedia Broadcast/Multicast Service; Architecture and Functional Description. TS 23.246, March 2005.

[ABG+01] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow. RSVP-TE: Extensions to RSVP for LSP Tunnels. RFC 3209, IETF, December 2001.

[ADF+01] L. Andersson, P. Doolan, N. Feldman, A. Fredette, and B. Thomas. LDP Specification. RFC 3036, IETF, January 2001.

[ASMA04] Peter Ashwood-Smith, Eric Mannie, and Daniel O. Awduche. Generalized Multi-Protocol Label Switching (GMPLS) Architecture. RFC 3945, IETF, October 2004.

[BBB+05] F. Berggren, A. Bria, L. Badia, I. Karla, R. Litjens, P. Magnusson, F. Meago, H. Tang, and R. Veronesi. Multi-Radio Resource Management for Ambient Networks. In *Proceedings of the 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications, Berlin, Germany*, September 2005.

[BBC+98] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475, IETF, December 1998.

[BCS94] R. Braden, D. Clark, and S. Shenker. Integrated services in the Internet architecture: an overview. RFC 1633, IETF, June 1994.

[BFY+00] Y. Bernet, P. Ford, R. Yavatkar, F. Baker, L. Zhang, M. Speer, R. Braden, B. Davie, J. Wroclawski, and E. Felstaine. A Framework for Integrated Services Operation over Diffserv Networks. RFC 2998, IETF, November 2000.

[BZB+97] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (RSVP) – version 1 functional specification. RFC 2205, IETF, September 1997.

[CDWW98] Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. RFC 2475, IETF, December 1998.

[CGN+05] Gustavo Carneiro, Carlos García, Pedro Neves, Zhikui Chen, Michelle Wetterwald, Manuel Ricardo, P. Serrano, Susana Sargento, and Albert Banchs. The DAIDALOS Architecture for QoS over Heterogeneous Wireless Networks. In *Proc. of the 14th IST Summit*, June 2005.

[Cra98] M. Crawford. Transmission of IPv6 Packets over Ethernet Networks. RFC 2464, IETF, December 1998.

[CRR04] Gustavo Carneiro, José Ruela, and Manuel Ricardo. Cross-layer design in 4G wireless terminals. *Wireless Communications, IEEE*, 11(2):7 – 13, April 2004.

[CWKS97] B. P. Crow, I. Widjaja, L. G. Kim, and P. T. Sakai. IEEE 802.11 Wireless Local Area Networks. *Communications Magazine, IEEE*, 35(9):116–126, September 1997.

[Dai] Daidalos IST Project. Designing Advanced Interfaces for the Delivery and Administration of Location independent Optimised personal Services. (FP6-2002-IST-1-506997).

[DBC+00] D. Durham, J. Boyle, R. Cohen, S. Herzog, R. Rajan, and A. Sastry. The COPS (Common Open Policy Service) Protocol. RFC 2748, IETF, January 2000.

[DH98] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, IETF, December 1998.

[FGI+05] T. Farnham, A. Gefflaut, A. Ibing, P. Mähönen, D. Melpignano, J. Riihijärvi, and M. Sooriyabandara. Toward Open and Unified Link-Layer API. In *Proceedings of IST Mobile and Wireless Summit 2005*, June 2005.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[GSL+05] Jens Gebert, Rolf Sigle, Johan Lundsjö, Per Magnusson, Mikael Prytz, Joachim Sachs, Eftychia Alexandri, Jan Markendahl, Fredrik Berggren, Ljupco Jorguseski, Remco Litjens, Reza Karimi, Haitao Tang, Francesco Meago, Catarina Cedervall, Peter Karlsson, George Koudouridis, Ralf Jennen, Ramón Agüero, and Leonardo Badia. WWI Ambient Networks, D 2.2 MRA Architecture, 17 January 2005.

[GZ03] Daqing Gu and Jinyun Zhang. QoS enhancement in IEEE 802.11 wireless local area networks. *Communications Magazine, IEEE*, 41(6):120–124, June 2003.

[HD03] R. Hinden and S. Deering. Internet Protocol Version 6 (IPv6) Addressing Architecture. RFC 3513, IETF, April 2003.

[HKL04] R. Hancock, G. Karagiannis, and Loughney. Next Steps in Signaling: Framework. draft-ietf-nsis-fw-07, November 2004.

[Iva05] Tonci Ivankovic. Support of multimedia broadcast/multicast service in UMTS networks. In *8th International Conference on Telecommunications*, volume 1, pages 91–98, June 2005.

[JPA04] D. Johnson, C. Perkins, and J. Arkko. Mobility Support in IPv6. RFC 3775, IETF, June 2004.

[KDEM04] Rajeev Koodli, Gopal Dommety, and Karim El Malki. Fast Handovers for Mobile IPv6. draft-ietf-mipshop-fast-mipv6-03.txt, 25 October 2004.

[KJS] Anis Koubaa, Aref Jarraya, and Ye-Qiong Song. SBM protocol for providing real-time qos in ethernet LANs.

[LC03] Shing-Jiuan Leu and Ruay-Shiung Chang. Integrated service mobile internet: Rsvp over mobile ipv4&6. *Mobile Networks and Applications*, 8(6):635–642, 2003.

[LLM+98] David C. Lee, Daniel L. Lough, Scott F. Midkiff, Nathaniel J. Davis IV, and Philip E. Benchoff. The Next Generation of the Internet: Aspects of the Internet Protocol Version 6. *Network, IEEE*, January 1998.

[MW05] P. McDermott-Wells. What is Bluetooth? *Potentials, IEEE*, 23(5):33–35, January 2005.

[NR05] T.D. Nadeau and H. Rakotoranto. GMPLS operations and management: today's challenges and solutions for tomorrow. *Communications Magazine, IEEE*, 43(7):68–74, July 2005.

[PG93] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Trans. Netw.*, 1(3):344–357, 1993.

[PP05] Dan Pilone and Neil Pitman. *UML 2.0 in a Nutshell.* O'Reilly, June 2005.

[RDCR02] Manuel Ricardo, Jaime Dias, Gustavo Carneiro, and José Ruela. Support of IP QoS over UMTS Networks. In *Proceedings of the 13th IEEE International Symposium On Personal, Indoor and Mobile Radio Communications*, September 2002.

[RVC01] Eric C. Rosen, Arun Viswanathan, and Ross Callon. Multiprotocol Label Switching Architecture. RFC 3031, IETF, January 2001.

[SPG97] S. Shenker, C. Partridge, and R. Guerin. Specification of Guaranteed Quality of Service. RFC 2212, IETF, September 1997.

[Sta96] William Stallings. IPv6: The New Internet Protocol. *Communications Magazine, IEEE*, July 1996.

[SW97] S. Shenker and J. Wroclawski. General characterization parameters for integrated service network elements. RFC 2215, IETF, September 1997.

[Tel98] Information Technology Telecommunications. Iso/iec final dis 15802-3 (ieee p802.1d/d17), 25 May 1998.

[VCL04] R. Vida, L. Costa, and LIP6. Multicast Listener Discovery Version 2 (MLDv2) for IPv6. RFC 3810, IETF, June 2004.

[Wro97] J. Wroclawski. Specification of the Controlled-Load Network Element Service. RFC 2211, IETF, September 1997.

[YHB+00] R. Yavatkar, D. Hoffman, Y. Bernet, F. Baker, and M. Speer. SBM (Subnet Bandwidth Manager): A Protocol for RSVP-based Admission Control over IEEE 802-style networks. RFC 2814, IETF, May 2000.