Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TECHNISCHE UNIVERSITÄT
CAROLO-WILHELMINA
ZU BRAUNSCHWEIG

Master's Thesis

Quality of Service for Multiplayer Game Provisioning in Mobile Ad Hoc Networks

Dirk Budke

September 30, 2005

Tutors:	Oliver Wellnitz	wellnitz@ibr.cs.tu-bs.de
	Károly Farkas	farkas@tik.ee.ethz.ch
Supervisors:	Prof. L. Wolf	wolf@ibr.cs.tu-bs.de
	Prof. B. Plattner	plattner@tik.ee.ethz.ch

Erklärung

Ich versichere, die vorliegende Arbeit selbständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Zürich, September 30, 2005 Dirk Budke

Abstract

Real-time applications, especially real-time multiplayer games, are getting popular in mobile ad hoc environments as mobile devices and wireless communication technologies are becoming ubiquitous. However, these applications have strict demands on the underlying network, requiring low latency with a minimum of jitter and packet loss. Therefore, Quality of Service (QoS) support from the network is essential to meet the demands of real-time applications, which is a challenging task in mobile ad hoc networks due to the high level of node mobility, properties of the wireless communication channel and the lack of central coordination. Employing a typical multiplayer game scenario, proactive as well as reactive routing protocols for mobile ad hoc networks have been evaluated in a network simulator. The reactive protocol AODV provides the best overall performance, however, latency and jitter are still too high to meet the demands of multiplayer computer games.

In order to improve the performance of real-time applications, Quality of Service extensions to the AODV routing protocol, the interface queue and the IEEE 802.11 MAC layer have been evaluated in this thesis. Besides common QoS extensions some new ones, such as real-time neighbour aware rate control, hop constrained queueing timeouts and MAC layer based neighbour detection have been proposed. The QoS extensions have been implemented in the network simulator NS-2 and in a test environment running Linux.

By gradually applying RTS/CTS adaption, priority queueing with timeouts and real-time neighbour aware rate control combined with broken link detection and backup routes, the performance of real-time multiplayer games has been improved significantly and is more than five times better than without the QoS extensions. For connections up to three hops the demands of real-time applications like multiplayer games are met in the employed simulation scenario using these QoS extensions.

Kurzfassung

Echtzeitanwendungen und insbesondere Mehrbenutzerspiele mit Echtzeitanforderungen in mobilen ad-hoc Umgebungen gewinnen durch die steigende Bedeutung von mobilen, schnurlosen Kommunikationsgeräten immer mehr an Beliebtheit. Jedoch stellen sie strikte Anforderungen an das zu Grunde liegende Netz, wie geringe Latenz, Jitter und minimaler Paketverlust. Auf Grund der hohen Mobilität der Knoten, spezieller Eigenschaften der schnurlosen Datenübertragung und der dezentralen Administration von mobilen ad-hoc Netzwerken, ist die Bereitstellung der Dienstgüte eine große Herausforderung.

In dieser Arbeit wurden mit Hilfe des Netzwerksimulators NS-2 vier proaktive und reaktive Wegewahlprotokolle in einem typischen Mehrbenutzerspielszenario analysiert und miteinander verglichen. Dabei lieferte das reaktive Protokoll AODV die besten Ergebnisse bezüglich Latenz, Jitter und Paketverlust.

Dennoch sind die erzielten Werte nicht ausreichend für Echtzeitanwendungen. Daher wurden weitere Mechanismen zur Verbesserung der Dienstgüte auf MAC-, Warteschlangen- und Wegewahlebene in den Netzwerksimulator NS-2 integriert und in einer Testumgebung unter Linux implementiert. Mit Hilfe von Prioritätswarteschlangen mit Zeitüberschreitungs- und Mengenflusskontrolle, kombiniert mit dem Abschalten von RTS/CTS und der Bereitstellung von Ersatz-Routen und Verbindungsunterbrechungserkennung auf MAC Ebene, wurde die Leistung erheblich verbessert. Dadurch konnte die Dienstgüte für Echtzeitpakete soweit gesteigert werden, dass sie die Echtzeitanforderung von Mehrbenutzerspielen erfüllt, falls die Daten nicht über mehr als drei Knoten weitergeleitet werden.

Technische Universität Braunschweig
Institut für Betriebssysteme und Rechnerverbund
Prof. Dr. L. Wolf

TU Braunschweig · Institut für Betriebssysteme und Rechnerverbund · Postfach 3329 · 38023 Braunschweig

Mühlenpfordtstr. 23
38106 Braunschweig
Telefon: (05 31) 3 91 – 3283
Telefax: (05 31) 3 91 – 5936
WWW: http://www.ibr.cs.tu-bs.de/

Prof. Dr. L. Wolf

Braunschweig, 01.04.2005

Aufgabenstellung für die Masterarbeit

Quality of Service for Multiplayer Game Provisioning in Mobile Ad Hoc Networks

vergeben an

Herrn cand. inform. Dirk Budke
Matr.-Nr. 2648556, Email: budke@ibr.cs.tu-bs.de

Mobile Ad Hoc Networks are self-organised wireless multi-hop networks comprising various heterogeneous mobile devices that are independent of any existing networking infrastructure and can be set up spontaneously. Due to device mobility the network topology may vary frequently. Additionally, new nodes may emerge and join the ad hoc network whereas existing nodes may vanish at any time. In order to send packets from a source node all along the way passing several intermediate nodes to a destination node a routing protocol is needed that takes device mobility and wireless network conditions into account.

Multiplayer Computer Games enjoy great popularity and with the advent of powerful mobile devices people would like to play games using their mobile devices. Since computer games, in particular real-time games have strict demands on the network, quality of service (QoS) has to be provided to cope with unreliable network connections, low bandwidth, high latency and limited device resources.

Task description

Starting with a collection of networking requirements of common multiplayer games, typical game scenarios should be modelled in the network simulator NS-2 including mobility and communication patterns. Afterwards, these scenarios should be simulated employing standard ad hoc routing protocols as well as enhanced protocols providing for quality of service. This also includes the modification of existing or implementation of new QoS routing protocols. The results of the simulations should be evaluated against the requirements mentioned before. The routing protocols that achieve the best average performance should be implemented in Linux and evaluated in a test environment. A concurrent NS-2 / Linux implementation is preferred if possible. Finally, an interface between the routing protocol and the SIRAMON framework should be specified and implemented to enable SIRAMON using QoS routing and accessing essential networking parameters.

Die **Laufzeit beträgt 6 Monate**. Die Hinweise zur Durchführung von Studien-, Master- und Diplomarbeiten am IBR sind zu beachten (siehe http://www.ibr.cs.tu-bs.de/lehre/arbeiten-howto/).

Aufgabenstellung und Betreuung:

Prof. Dr. Lars Wolf ______________________

Dipl.-Inform. Oliver Wellnitz ______________________

cand. inform. Dirk Budke ______________________

Contents

Chapter 1

Introduction

Online real-time applications such as multiplayer games have become very popular recently and are a big business in today's Internet expecting increasing revenue. Real-time multiplayer games share the common property that the information, for instance, the game character movements have to be delivered as fast as possible to the counterparts for a smooth game control. If this data is delayed and arrives too late the game is not playable any more. Thus, real-time applications like multiplayer computer games have strict demands on the underlying network and require low latency connections.

1.1 Mobile Ad Hoc Networks

With the constantly increasing amount of powerful mobile devices, such as PDAs, laptops and mobile phones with wireless networking support, real-time applications and in particular multiplayer computer games in wireless environments are gaining much interest. Even though wireless networking infrastructure, such as base stations or access points, might not be available, various heterogeneous mobile devices can connect to each other setting up spontaneously a self-organising mobile wireless multihop ad hoc network (MANET) that does not rely on any existing infrastructure.

However, wireless ad hoc networking reserves several challenges mainly due to the mobility of the nodes, limited device resources, properties of the wireless channel and the lack of central coordination. Since the mobile nodes can move around, the wireless network's topology may vary frequently and unpredictably.

Additionally, new nodes may emerge and join the MANET, whereas existing nodes may vanish at any time. Therefore, it is difficult to provide an up-to-date global state of the wireless network topology. In particular, the properties of the wireless channel can change arbitrarily resulting in unreliable connections with low bandwidth and high latency making it very difficult to deliver real-time traffic. The nodes participating in such a network need not be directly in the transmission range of each other still being able to communicate because every node can act as both, an end host and a router at the same time, which allows the transmission of messages via multiple hops. The routing protocol, which is needed to acquire a route from the source to the destination node via multiple hops, must cope with these challenges. In addition, it has to be able to satisfy the Quality of Service requirements of real-time applications as well. Hence, Quality of Service provisioning in mobile wireless ad hoc networks is even more difficult than in wired networks because of, among others, arbitrary mobility, unreliable wireless links, signal fading and interference and the used channel access mechanisms. Thus, meeting the applications' demands in MANETs is very challenging and has been an open on-going field of research in the last couple of years.

MANETs can be employed in a broad variety of scenarios with either little or no communication infrastructure or when the existing infrastructure is too expensive or has been destroyed for instance, in a natural disaster such as earthquakes, tsunamis and hurricanes. In that case, the rescue team must be able to set up a new communications network in an ad hoc manner to conduct disaster recovery as quickly as possible.

Besides disaster recovery and military applications, MANETs are gaining much popularity in daily life. People would like to have an easily and quickly deployable network to exchange data or distribute a virtual whiteboard, for instance in interactive lessons or business meetings. In particular, multiplayer games are a promising application of MANETs, for example, when travelling or at the schoolyard. With the widespread use of powerful mobile devices people would like to play multiplayer games without the hassle of installing a game or the burden of setting up any networking infrastructure. To deal with these challenges a distributed service provisioning framework called SIRAMON [1] is currently developed at ETH Zurich [2] that provides for description, indication, deployment and management of services in MANETs.

1.2 Focus of this Thesis

In this thesis, common Quality of Service extensions, such as priority queueing, backup route, broken link detection, etc. are analysed and evaluated, and some new ones, such as real-time neighbour aware rate control and hop constrained queueing timeouts are proposed to ad hoc routing in IEEE 802.11 [3] mobile ad hoc networks, focusing on the requirements of real-time applications, specifically real-time multiplayer computer games. As the starting point of the work, the *Ad Hoc On-Demand Distance-Vector* (AODV) [4] routing protocol has been selected because this protocol had shown the best initial performance in simulations compared to some other ad hoc routing protocols. However, the end-to-end communication delay and delay jitter experienced using AODV are still too high to meet the demands of multiplayer computer games. In order to improve the performance of AODV for real-time applications, enhanced congestion handling and coping with mobility and the unreliable wireless channel properties are required. Thus, in this thesis, following a cross-layer design, the following Quality of Service mechanisms have been used: (1) Quality of Service extensions to AODV such as local repair, backup route; (2) Traffic management mechanisms like priority queueing, timeouts, real-time neighbour aware rate control; and (3) MAC (Medium Access Control) layer support mechanisms such as broken link detection, signal strength monitoring, neighbour detection, RTS/CTS (Ready To Send/Clear To Send) adaptation.

1.3 Organisation of the Thesis

The outline of this thesis is organised as follows: Chapter 2 provides the necessary background on Quality of Service, wireless communication, ad hoc routing, Quality of Service in MANETs and multiplayer computer games. Then, selected Quality of Service extensions targeting at Quality of Service provisioning for multiplayer games are presented and discussed in Chapter 3. An overview of the implementation of the Quality of Service extension in the network simulator NS-2 [5] and in a real test environment is given in Chapter 4. Chapter 5 shows the performance results of the routing protocol comparison and Quality of Service extensions in a typical multiplayer game scenario. Chapter 6 concludes the thesis and presents an outlook for future work.

1.4 Acknowledgements

I would like to express my sincere gratitude to all people that have supported me during this master's thesis. Especially, I would like to thank:

Prof. L. Wolf and *Prof. B. Plattner* who made it possible for me to conduct this master's thesis in collaboration with ETH Zurich

Károly Farkas and *Oliver Wellnitz* for advising and supporting me

Florian Maurer for excellent teamwork and teaching me some Swiss German

Frank Lamm for proof-reading

My family and in particular my girlfriend *Uschi Lindert*.

Chapter 2

Fundamentals

This chapter provides the reader with the essential background information. Quality of Service principles and architectures commonly employed in wired networks and the Internet are introduced in Section 2.1. The challenges in wireless communications and in particular problems faced in IEEE 802.11 wireless networks are addressed in Section 2.2. Next, Section 2.3 introduces four popular routing protocols for MANETs. Afterwards, existing Quality of Service approaches for MANETs are described in Section 2.4 and finally, the last section of this chapter presents the characteristics and requirements of real-time multiplayer games.

2.1 Quality of Service

Various types of popular networked applications have emerged in the past couple of years with different demands on the network, such as delay, jitter, bandwidth and reliability.

Delay: determines the end-to-end time it takes to transmit a packet from the source to the destination. This is often called *latency* as well. It is important to distinguish between end-to-end delay and *round-trip time* (RTT) delay which measures the time it takes to send a packet to the destination and back to the source. In particular in wireless networks, the delay might not be symmetric and a connection can experience a higher delay in one direction than in the other direction.

Jitter: describes how much the packets vary in latency and is determined by calculating the standard deviation of latency.

Bandwidth: defines the maximum amount of data the network is able to transmit within a certain time frame.

Reliability: specifies to which degree the network prevents transmission errors and thus garbled packets.

The aim of *Quality of Service* (QoS) provisioning is to assure that these QoS requirements are met by the network. Table 2.1 illustrates typical networked applications and their Quality of Service demands (low: - , medium: ○, high: +) grouped into three categories: (1) high reliability, (2) minimal jitter and (3) high demands on latency and jitter. The first five applications require a strictly reliably transport mechanism, for example, transmission errors during a file transfer session yielding in a compromised copy of the transferred file are not acceptable. Audio and video streaming applications demand very low jitter while reliability is of less importance. Interactive real-time applications such as IP telephony and video conferencing have high demands on both delay and jitter but can cope with garbled packets to some extent. Real-time multiplayer games have similar demands but are more sensitive to garbled packets.

Application	**Delay**	**Jitter**	**Bandwidth**	**Reliability**
E-Mail	-	-	-	+
File Transfer	-	-	○	+
Web	○	-	○	+
Remote Access	○	○	-	+
Chat	○	○	-	+
Audio streaming	-	+	○	-
Video streaming	-	+	+	-
Voice over IP	+	+	-	-
Video Conference	+	+	+	-
Multiplayer Games	+	+	-	○

Table 2.1: Typical Requirements of Popular Networked Applications

2.1.1 QoS Mechanisms

The most general and currently most common approach to allow for QoS in wired networks is *over-provisioning*. Similar to the telephone system more resources are made available as generally consumed by employing commodity high performance interconnects. The following well-known QoS mechanisms are frequently employed in wired networks to meet specifically the QoS demands of networked applications.

Buffering: to cope with applications that are sensitive to jitter but do not have stringent delay requirements, such as video or audio streams, buffering the data at the receiver is very effective.

Priority Queueing: at various layers of the protocol stack, queues are employed for buffering packets. When using a priority queue, traffic with specific demands, for example on latency, can be handled differently by the queue, giving it a higher priority.

Traffic Shaping / Rate Control: the idea of traffic shaping is to prevent for traffic bursts and to limit the maximal bandwidth data flows are allowed to consume, by employing the Leaky Bucket Algorithm (LBA) [6], for instance.

Admission Control: before a new connection can be established it has to pass an admission controller that determines whether enough resources are available and accepts or rejects the connection, respectively.

Resource Reservation: before the application starts its data stream transmission it reserves required resources in the network at each node that is used to carry the stream, for example using the *Resource Reservation Protocol* (RSVP) [7].

Multipath Routing: if redundant links to the destination are available these are used concurrently for data transmission, either to increase reliability or bandwidth.

2.1.2 QoS Architectures

In the Internet two different QoS architectures are widely employed for network traffic engineering. They can be classified into *stateful, reservation-oriented* or *stateless* QoS architectures. A stateful QoS architecture reserves resources at every node the data flow passes through employing a resource reservation protocol. A stateless QoS architecture classifies every packet into different categories depending on its QoS demands and is handled using only local network state information. Thus, intermediate nodes do not keep per-flow state information as in stateful QoS architectures. Both approaches are briefly described in the following two sections.

Flow based Integrated Services

With IntServ, the user negotiates a *Service Level Agreement* (SLA) describing the demands of the application on the underlying network and type of traffic. Then the host requests a particular Quality of Service which might be granted or denied by the network. For each flow, resources according to the SLA are reserved at every intermediate node along the path from the source to the destination before the data flow starts using RSVP. Thereby, a virtual circuit with sufficient available resources is established to allow for QoS guarantees. However, all packets referring to the same flow have to be sent along the same path to benefit from resource reservation. If multipath routing is used then for each path resources have to be reserved.

Class based Differentiated Services

Central components of *Class based Differentiated Services* (DiffServ) are traffic classifiers, traffic shaping and queueing components that support different *Per-Hop Behaviour* (PHB). Data packets are classified into different traffic streams by the traffic classifiers. Depending on the *DiffServ Code Point* (DSCP) field in the IP header the appropriate PHB is selected to further process the packet. Expedited Forwarding [8], for instance, defines two classes, high and low priority. Assured Forwarding [9] defines four priority classes and three discard probabilities resulting in 12 classes. The advantages of DiffServ are that no resource reservation and end-to-end flow negotiation have to be carried out. Furthermore, it is easy to implement, stateless and does not require intermediate nodes to manage single flows. Since the differentiation is done at the network layer replacing the original *Type of Service (TOS)* field in IPv4 or traffic class byte in IPv6 with the DSCP field, it is independent of the underlying network technology. However, DiffServ does not provide strict QoS and allows only for probabilistic QoS guarantees.

2.2 Wireless Communication

Wireless networks allow for convenient and flexible mobile communication. Furthermore, setting up new wireless networks is often much cheaper than wired networks and also easily and very quickly deployable. IEEE 802.16 *Wireless Interoperability for Microwave Access* (WiMAX) [10] has been gaining much interest recently for providing wireless technology to *Metropolitan Area Networks* (MAN). In the case of *Wireless Local Area Networks* (WLAN) IEEE 802.11 [3] dominates the market and for wireless *Wireless Personal Area Networks* (WPAN) IEEE 802.15 Bluetooth [11] has become widespread. In the following sections, the issues that come along with wireless communications that make QoS provisioning rather difficult are introduced and the popular WLAN standard IEEE 802.11 that has been used throughout this thesis is highlighted.

2.2.1 Medium Access Control

Wireless communication faces the challenge of sharing the same wireless medium with several nodes. Thus, an efficient *Medium Access Control* (MAC) strategy is required, that prevents multiple nodes from using the same resources simultaneously, which causes collisions and garbled packets. Common MAC protocols for wireless networks can be categorised as *schedule* or *contention* based protocols. In a schedule based protocol every node has a fixed time slot to transmit data. If the node does not send data at the particular time slot bandwidth is wasted. When the contention based protocol *Carrier Sense Multiple Access* (CSMA) is employed, the node senses if the channel is free before it starts its transmission. However, the sender does not know about concurrent transmissions and resulting interference at the receiving node, which gives rise to the well-known *hidden terminal* and *exposed terminal* problems [6]. The most popular contention based protocol hence is *Distributed Coordination Function* (DCF) that extends CSMA with a hand-shake protocol to prevent the hidden terminal problem.

Hidden Terminal Problem

Figure 2.1 illustrates the hidden terminal problem. Node A and C are out of transmission range of each other and A sends packets to B. Before C is going to start its transmission to node B as well it senses if the medium is available. Since it will not hear node A, it will falsely conclude that the channel is free and start its transmission to node B, resulting in garbled packets sent by A. Thus, the cause of

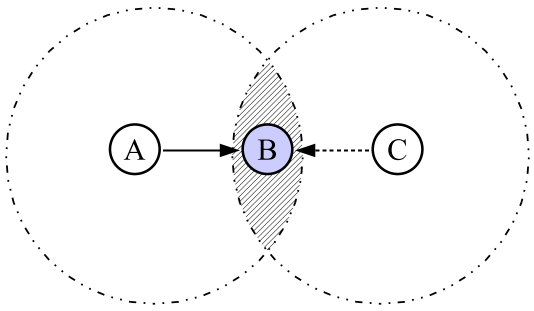

Figure 2.1: Hidden Terminal Problem

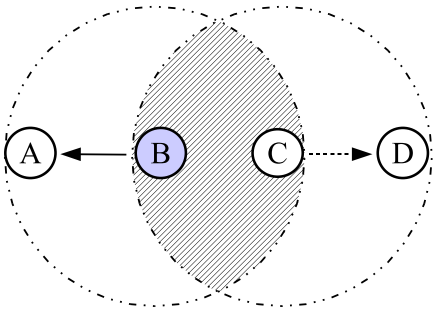

Figure 2.2: Exposed Terminal Problem

the hidden terminal problem is that a sender does not know about other competitors that are out of transmission range and transmitting to the same destination.

Exposed Terminal Problem

In the exposed terminal scenario depicted in Figure 2.2 node *B* is transmitting to node *A* and node *C* is going to send to node *D*, whereas *B* and *C* are within transmission range of each other. Node *C* senses the medium and falsely concludes that the transmission to node *D* will interfere with the data sent by node *B* and thus will not start its transmission. However, the two transmissions will not interfere at the receiver nodes *A* and *D*. In contrast to wired communication where the signals are propagated to all stations, multiple transmissions can occur at the same time without interference in short-range wireless communication as long as any two nodes do not transmit to the same node.

Carrier Sense Multiple Access with Collision Avoidance

To overcome the hidden terminal problem, *CSMA / Collision Avoidance* (CSMA/CA) has been introduced. Figure 2.3 illustrates the basic principle of CSMA/CA. Node *B* and D would like to transmit to node *C* resulting in a potential collision, as depicted in Figure 2.3. Before node *B* starts transmission to *C* a short *Ready to Send* frame RTS containing the required data transmission time is broadcast to node *C*. Node *A* also receives the RTS and knows that it is next to node *B* that requests permission to send to node *C*. Node *D* is out of transmission range and does not receive the RTS. Node *A* remains silent long enough that *C* can send a *Clear to Send* frame CTS. Any station that hears the CTS is close to node *C* and remains silent for the duration of the data transmission from node *B* to node *C*. In this case node *A* is not in transmission range of *C* and does not receive the CTS which allows *A* to transmit in parallel without interfering with the data sent by node *B*. Node *D*, however, receives the CTS and waits until the data transmission has been finished before sending any packets and thus avoiding the hidden terminal problem. The exposed terminal problem in the scenario in Figure 2.2 is solved as well. However, in general the exposed terminal problem is intensified by the use of RTS/CTS as explained in [12]. A node that hears a CTS will not reply to an RTS and waits until the communication has finished before transmitting its own CTS even though the parallel communication would not have interfered with the previous one.

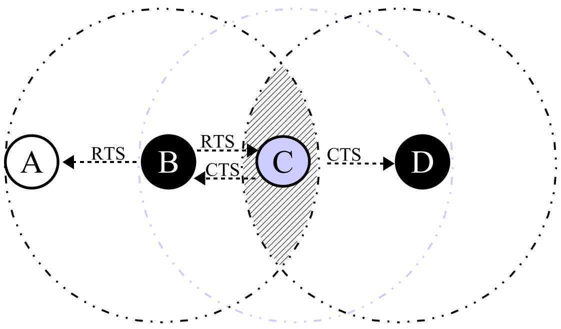

Figure 2.3: Ready to Send (RTS) and Clear to Send (CTS)

Furthermore, collisions can still occur when two nodes send RTS frames concurrently. In this case both senders wait a random timeout using a binary exponential backoff algorithm and then try again. It is important to notice that, on the one hand, the hidden terminal problem is alleviated by RTS/CTS when sending packets that are bigger than a certain threshold and there are many interfering nodes [13], although, it still remains for RTS frames. On the other hand, RTS/CTS introduces

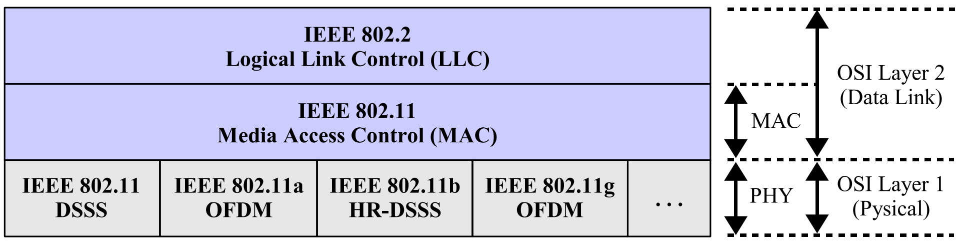

Figure 2.4: IEEE 802.11 Protocol Stack

a non-neglectable overhead to CSMA.

2.2.2 Wireless LAN IEEE 802.11

The IEEE 802.11 wireless LAN family [3] is very popular and widespread used for instance, at private homes, offices, airports, hotels and universities. It can operate in two modes, such as *infrastructure* or *Basic Service Set* (BSS) mode employing access points and the *ad hoc* or *Independent Basic Service Set* (IBSS) mode forming an ad hoc network depending on no further infrastructure.

Figure 2.4 illustrates the IEEE 802.11 protocol stack. At the physical layer, IEEE 802.11 defines several different transmission techniques, whereas the most commonly employed are *High Rate / Direct Sequence Spread Spectrum* (HR-DSSS) in IEEE 802.11b that provides up to 11 Mbit/s and *Orthogonal Frequency Division Multiplexing* (OFDM) in IEEE 802.11g delivering up to 54 Mbit/s transmission rate. Furthermore, the transmission rate adapts to the wireless signal quality and in case of IEEE 802.11b the bit rate is reduced from 11 Mbit/s to 5.5, 2, 1 Mbit/s when the link quality degrades and many packets have to be retransmitted.

In IEEE 802.11 networks, the data link layer known from the OSI reference model is split into two separate sub-layers, the *Medium Access Control* (MAC) layer and the *Logical Link Control* (LLC) layer. The MAC layer controls the access to the shared medium and the LLC layer hides the peculiarities of the different IEEE 802 protocols and provides a unique interface to the network layer that allows for simple bridging from wireless to wired networks.

The IEEE 802.11 MAC layer supports two medium access modes, such as the *Distributed Coordination Function* (DCF) used in ad hoc and infrastructure mode and the *Point Coordination Function* (PCF) that is only used in infrastructure mode. In the case of DCF, CSMA with *Collision Avoidance* (CSMA/CA) is used to share the radio channel in a fair way. It provides physical channel sensing via CSMA and virtual channel sensing via RTS/CTS. Since wireless networks are very noisy and unreliable each unicast data frame is confirmed by ACK frames on the MAC layer to cope immediately with lost frames and collisions instead of handling this on the transport layer.

Figure 2.5 illustrates how a node acquires the channel and transmits a data packet. Before a node can acquire the channel it has to wait for a time slot *DCF Inter Frame Spacing* (DIFS). If the medium is free it can start its transmission. During the communication both nodes have to wait a *Short Inter Frame Spacing* (SIFS) before they are allowed to transmit their frames. Other nodes are deferred from sending and maintain a *Network Allocation Vector* (NAV) to indicate that the channel is busy.

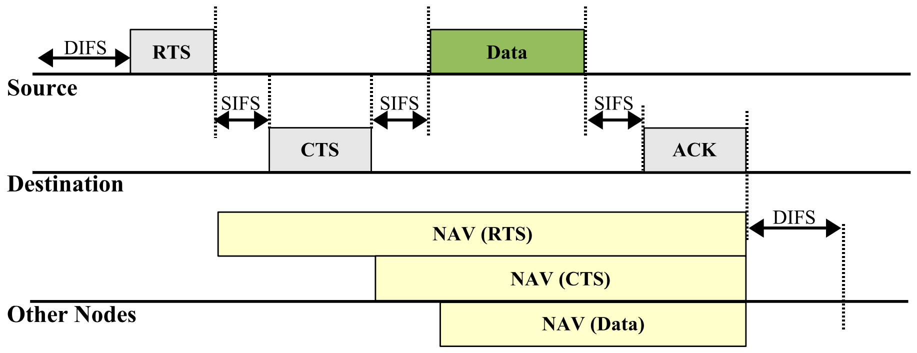

Figure 2.5: IEEE 802.11 DCF Contention using RTS/CTS

Ad Hoc IBSS Mode

In contrast to infrastructure mode, where only the access point sends BEACON management frames, the first active ad hoc node establishes an IBSS and starts sending BEACONs, which are needed to synchronise the nodes and maintain the ad hoc network. Other ad hoc nodes can join the network after receiving a BEACON and accepting the IBSS parameters found in the beacon frame if they have the same *Service Set Identifier* (SSID) and thus belong to the same ad hoc network. All nodes that join the ad hoc network must send a BEACON itself if they do not hear one from another node after the beacon interval and a short *random* timeout. The random timeout minimises the concurrent transmission of multiple BEACONs from several nodes. After receiving a BEACON, each node updates its local internal clock with the time-stamp found in the BEACON. This allows an ad hoc network to be partitioned if the connectivity is broken as well as to be merged.

Quality of Service in IBSS

IEEE 802.11b IBSS employs DCF to share the channel in a probabilistically fair way among the nodes. However, there is no guarantee that a node can transmit at a certain data rate or within a certain time interval. Moreover, the actual bit rate might be much lower than the maximum 11 Mbit/s due to a weak signal and the bit rate adaption mechanism, mentioned before. As a result the performance of other nodes that are still transmitting at a higher rate, for instance 11 Mbit/s, is considerably degraded due to the CSMA channel access method that allows for an equal channel access probability for all hosts. A host transmitting a frame at 1 Mbit/s acquires the channel for a longer time and thus, penalises other nodes that are still running at 11 Mbit/s. Effectively the data rate is limited even below 1 Mbit/s. Furthermore, when just taking the SIFS and DIFS timeouts into account the maximum theoretical throughput at MAC-Layer is limited to a maximum of 7.74 Mbit/s [14]. Thus, DCF just provides best-effort type of service and does not guarantee any Quality of Service neither bandwidth nor latency.

There are attempts to add QoS to 802.11 by tweaking the DIFS timeouts. The IEEE 802.11e draft [15] introduces two new coordination functions, such as a decentralised *Enhanced DCF* (EDCF) for ad hoc networks and a centralised *Hybrid Coordination Function* (HCF). Furthermore, IEEE 802.11e defines *Traffic Classes* (TC) by prioritising the traffic into classes with short (higher priority) or longer (lower priority) inter frame spacings. This way, nodes with a shorter inter frame spacing have a higher probability to access the channel than others. Email traffic for instance could be assigned a low priority and multiplayer games a high priority TC. In particular for real-time traffic, latency and jitter can be significantly reduced. On the other hand, IEEE 802.11e puts an additional overhead and reduces the overall throughput by 30 % as reported in [16]. Besides, still no strict QoS is guaranteed since IEEE 802.11e just provides a probabilistic QoS service.

In [17] the authors mention that IEEE 802.11b suffers from *grey zones* that exist at the border of the transmission range where the reception of packets is very unreliable as illustrated in Figure 2.6. The AODV routing protocol, for example, uses broadcast HELLO messages for neighbour link management. Broadcast messages are transmitted with a lower data rate at 1 Mbit/s in contrast to data packets and do not require an ACK frame. Furthermore, HELLO messages are very small packets. Thus, HELLO messages have a much higher probability to be received successfully at the destination than data packets. Node A and B exchange HELLO messages and node A adds node B to its neighbour list. Next, node A starts data

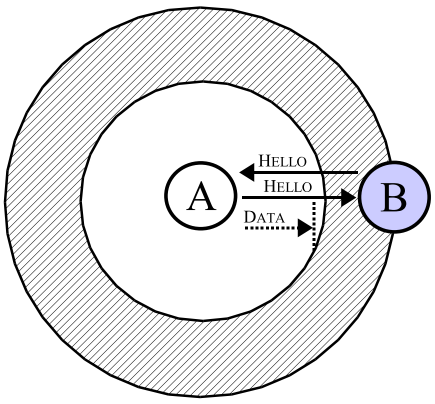

Figure 2.6: IEEE 802.11b Gray Zones

transmission to another node and uses *B* as the next hop. However, the packet will not arrive at *B*. If node *A* just relies on HELLO messages for broken link detection it will not recognise the broken link and after some retransmissions the packets are lost. Even if node *A* gets feedback from the link layer about excessive retransmissions the link will be marked as broken just for a short time until *A* receives the next HELLO from node *B*. Two mechanisms have been proposed in [17] to cope with grey zones.

Exchanging Neighbour Sets: nodes include their current neighbour set into HELLO messages. Thereby, the receiver of a HELLO message can detect whether the link is bidirectional or not.

Signal Strength Threshold: the signal quality of routing control messages is retrieved from the MAC layer and compared to a threshold. If the signal quality is too low the message is discarded.

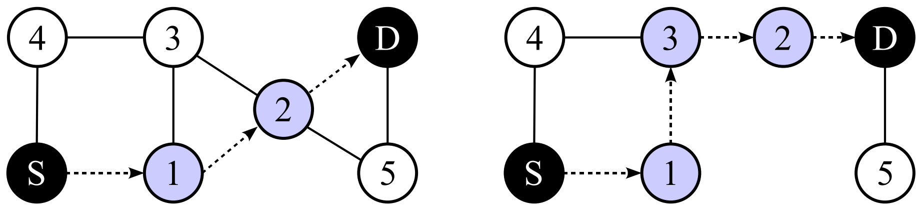

Figure 2.7: Node Mobility Changes Network Topology

2.3 Routing Protocols for MANETs

The wireless transmission range of a node is limited, thus, two nodes that are out of transmission range cannot directly interact with each other. To overcome this, every node participating in a cooperative MANET acts as a router and relays packets for other nodes in contrast to networks that are based on existing infrastructure. In order to send packets to nodes that are not within its vicinity, a routing algorithm running at each node provides a means to select the next hop along the path to the destination. However, due to the nodes' mobility the topology of the network may vary abruptly and a new path to the destination is required as illustrated in Figure 2.7. As node *2* moves towards the destination *D* the link to node *1* and thus the path *S-1-2-D* breaks. The routing protocol has to deal with changes in topology and provide for intact routes. In this scenario the routing protocol determines *S-1-3-2-D* as the new path.

In this section, basic routing mechanisms for ad hoc networks are introduced and four popular routing protocols that have been employed in network simulations are explained.

2.3.1 Classification of Routing Protocols

Many routing protocols optimised for mobile wireless ad hoc networks emerged recently and they can be classified into *topology-based* and *location-aware* protocols. In this thesis, the focus is on topology-based protocols that make routing decisions based on a logical model of the network. The *topology-based* protocols can be further divided into *proactive* (table-driven), *reactive* (on-demand), *hybrid* and *hierarchical* algorithms [18], as illustrated in Figure 2.8.

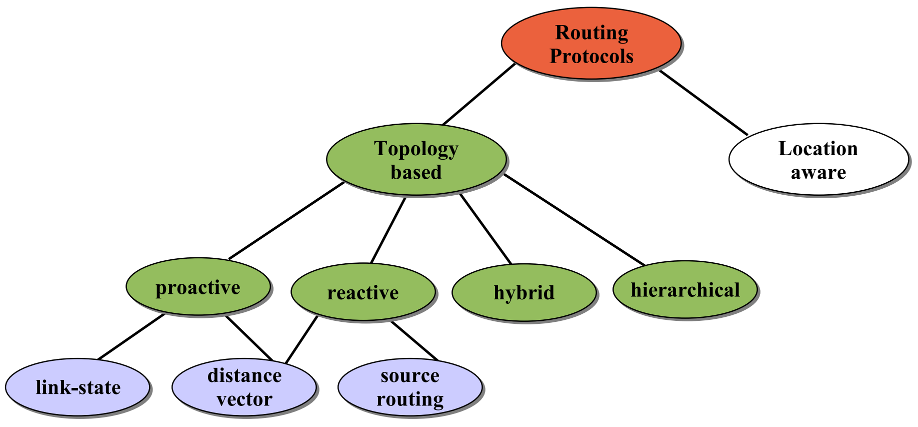

Figure 2.8: Classification of MANET Routing Protocols

Proactive Routing Protocols

Proactive or table-driven routing protocols hold entries specifying the next hop to all known nodes in their routing table. Thereby, the entire network topology is, in theory, known to all nodes. Routing messages are exchanged among the nodes periodically to update their routing tables. Depending on how significant the topology changes are, either full scale or incremental update messages are sent. Since a route to all nodes in the network is maintained in the routing table, data packets do not experience any additional delay and can be forwarded instantly. On the other hand, the periodic maintenance of routes, in particular to nodes towards no packets are currently sent, induces a certain overhead. Popular examples of proactive routing protocols are *Destination-Sequenced Distance-Vector* (DSDV) [19] and *Optimized Link State Routing* (OLSR) [20].

Reactive Routing Protocols

Reactive or on-demand routing protocols consist of two basic mechanisms, *Route Discovery* and *Route Maintenance*. Global broadcast messages are applied to discover a route on-demand. If a node S would like to send packets to node D it primarily has to obtain a route to the destination and starts the *Route Discovery* process. A route request message RREQ is disseminated throughout the ad hoc network. When it reaches the destination node D, it sends a route reply message RREP to the originator of the RREQ. As soon as node S receives the RREP a route has been established and it can begin with the transmission of data packets to node D using that route. Since the topology of a wireless ad hoc network can be very dynamic reactive protocols provide a means for *Route Maintenance* to propagate a broken link to affected nodes. Typically, route error RERR messages are sent to the concerned nodes. On the one hand, reactive routing protocols tend to require less routing overhead than proactive protocols because of the on-demand route discovery. No routing packets are exchanged for routes that are not currently in use. On the other hand, they take additional time to discover a new route and thus suffer from higher delays before starting the actual data transmission. Popular examples of reactive routing protocols are *Ad Hoc On-Demand Distance-Vector* (AODV) [4] and *Dynamic Source Routing* (DSR) [21].

Hybrid Routing Protocols

Hybrid routing protocols attempt to combine the best practises from both approaches using proactive and reactive techniques. Hybrid routing has been implemented in the *Zone Routing Protocol* (ZRP) [22, 23] where the nodes that are reachable within a certain hop-distance belong to a *Routing Zone*. Within this zone, a proactive routing algorithm is employed and for communication outside the routing zone nodes use a reactive protocol.

Hierarchical Routing Protocols

In *Hierarchical* routing protocols, nodes are grouped together building a backbone of the network that consists of several groups. When a node sends data outside its own group, the packet is first forwarded to the destination's group and then delivered to the destination. Thus, the actual path to the destination need not be known, just a path to the group. This reduces the routing overhead and hierarchical protocols provide a scalable architecture for very large ad hoc networks. Popular routing protocols based on hierarchical routing are *Core Extraction Distributed Ad hoc Routing* (CEDAR) [24] and *LANMAR* [25].

In the following sections, four popular reactive and proactive routing protocols are discussed. In Chapter 5, these protocols have been compared and evaluated in a typical multiplayer game scenario.

2.3.2 Destination-Sequenced Distance-Vector

Destination-Sequenced Distance-Vector (DSDV) [19] is one of the first proactive distance-vector routing protocols in particular targeting MANETs. It is based on the classical *Distributed Bellman-Ford algorithm* (DBF) [26] that is widely employed in the Internet and known as the *Routing Information Protocol* (RIP) [27].

Basic Algorithm

For each destination node x every node i manages a set of distances $D_{ij}(x)$ whereas j are the neighbours of i. Node i chooses neighbour k as the next hop for packets to x if $D_{ik}(x) = min(D_{ij}(x))$ for all neighbours j of node i. Following this strategy the packet is sent along the shortest path to the destination. In order to have an accurate state of the distances each node periodically broadcasts to its neighbours the routing table containing the estimated shortest path to every other node in the network.

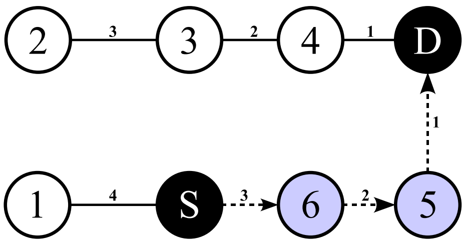

Figure 2.9: Basic Distance-Vector Algorithm with Distances to Node *D*, Step 1

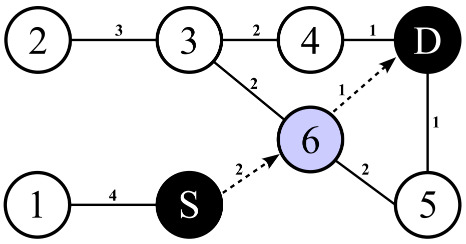

Figure 2.10: Basic Distance-Vector Algorithm with Distances to Node *D*, Step 2

Figure 2.9 and 2.10 illustrate the concept of the distance-vector protocol. In the example, every link is marked with the distance to node *D*. In the beginning in Figure 2.9, it takes 3 hops for node *S* to send a packet to node *D* using the path *S-6-5-D*. As node *6* moves towards node *D* as depicted in Figure 2.10 two new links, *6-3* and *6-D*, are established after *3* and *D* sent their routing tables. Then, node *6* broadcasts to its neighbours its own new routing table as depicted in Table 2.2. As a result, node *S* updates its routing table as well. Since the new route to *D* just takes 2 hops it will choose *S-6-D* as the new path. Until the next periodic update, node *1* does not know about the link change and still assumes that it takes 4 hops to node *D*. The nodes determine the shortest path in a completely distributed way without additional coordination but the basic DBF algorithm mainly suffers from two problems and need to be adapted to properties of MANETs:

Routing Loops: As the topology of the network changes, nodes make routing decision based on stale and thus incorrect information and routing loops oc-

Destination	Hops
6	0
D	1
1	2
2	2
3	1
4	2
5	1

Table 2.2: Routing Table of Node *6*

cur. Packets are forwarded in a loop until the maximal hop count has been exceeded without ever reaching the destination. Additional centralised co-ordination among the nodes to avoid routing loops is no option because the network topology can change rapidly in MANETs.

Count-to-Infinity: Assume the link between node *S* and *6* in Figure 2.9 breaks, thus node *S* will not receive the routing table from node *6* any more but still from node *1*. Since node *1* claims to have a route to *D* with 4 hops, node *S* will change its hop count to 5 to account for the additional hop from *1* to *S*. In the next turn, node *1* receives the updated hop count from *D* and increments it to 6 and so forth, until counting to ∞.

Route Management

DSDV works as DBF but takes some precaution to cope with routing loops and the count-to-infinity problem. It basically marks routes with sequence numbers to distinguish fresh from stale routes. Each node increments its own sequence number and broadcasts periodically route advertisements to the neighbours containing the distance and current sequence number of all known nodes. Always, routes marked by more recent sequence numbers are preferred. If two routes have the same sequence number, the route with the smaller metric such as the number of hops is chosen. Due to the periodic updates and the fact that fresher links are preferred, it might happen, that the next hop to a particular node oscillates as illustrated in Figure 2.11.

Regarding node *D*, the intermediate node *I* has the current hop count of 3 and sequence number 41 in its routing table. Thus, node *S* can send message to the destination via 4 hops. Next, node *I* receives the advertisements from node *1* which

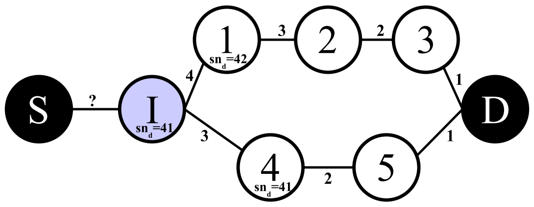

Figure 2.11: Oscillation of Links in DSDV

has a higher sequence number to *D* and thus is preferred. Therefore, the hop count is set to 4 and propagated to *S* which sets the hop count to 5. However, the next update from node *4* propagates the smaller hop count to *I* and then to *S* again. If the timeouts between the periodic advertisements are high enough the link change might oscillate. To cope with fluctuation of routes an *average settling time* for each route has been introduced. When a node receives new routing information it waits some seconds, before advertising the new route to prevent for route fluctuation.

When a node receives considerably modified routing information, for instance due to broken links or change of network topology, this is broadcast to the neighbours immediately. Additionally, the hop count is set to ∞ and the sequence number is increased. When a node receives a route with a ∞ metric and it is aware of a fresh route, that is a route with a higher sequence number, it propagates this quickly to its neighbours. Incremental routing updates, containing only changes since the last full update, are employed to keep routing message overhead at a minimum. Furthermore, packets are queued for a specific amount of time if currently no route to the destination is available. The routing table at each node lists all available destinations with their number of hops as depicted in Table 2.3. Each routing table entry is tagged with a sequence number generated by the destination and a time-stamp when the route had been installed or updated.

Destination	**Hops**	**Time-stamp**	**Sequence Number**
D	3	123	42

Table 2.3: Structure of DSDV Routing Table

2.3.3 Optimized Link State Routing

Optimized Link State Routing (OLSR) specified in the experimental RFC 3626 [20] is a proactive table-driven routing protocol based on the classical link state algorithm [6]. In link state routing each node periodically broadcasts link state messages containing the list of neighbours. The messages are flooded throughout the network and every node creates a graph with the entire topology of the network. Next, each node runs Dijkstra's shortest-path [28] algorithm and stores the results in its routing table. OLSR attempts to optimise and reduce the routing overhead caused by flooding. OLSR improves the classical link state algorithm by selecting particular intermediate nodes to reduce the amount of broadcast routing traffic that has to be flooded through the entire network. Due to these optimisations OLSR is in particular suitable for large and dense mobile wireless ad hoc networks. As a MANET routing protocol OLSR does not require reliable connections and in-order delivery of routing control messages.

Multipoint Relay Nodes

In particular in wireless networks, the classical link state routing algorithm suffers from high overhead induced by needlessly retransmitted link state information called *traffic control* TC messages. OLSR attempts to minimise the routing overhead caused by flooding TCs by selecting special *Multipoint Relay* (MPR) nodes, which forward the link state messages. To prevent for routing loops and to make sure that nodes can distinguish between stale and fresh link states, every TC carries a sequence number. Figure 2.12 illustrates the classical link state flooding. Node *S* broadcasts the link state information to its neighbours. Next, every blue node forwards the message reaching the green nodes which forward the message as well. However, not all messages are necessary to make sure that every node receives the TC.

In contrast, OLSR employs MPR nodes to reduce the amount of needlessly retransmitted TCs as depicted in Figure 2.13. Node *S* broadcasts the TC to its neighbours. Since node *I* is not an MPR node it does not forward the message. The same holds for the green nodes. Only MPR nodes forward broadcast TCs to flood the network yielding in less routing messages. In particular in larger networks this approach can reduce significantly the amount of redundantly transmitted TCs. However, the nodes have to distributedly determine an optimal set of MPRs under the condition that every node has at least one MPR as a neighbour or is an MPR itself. It has been shown in [29] that calculating the optimal set of MPRs, which form a dominating set [30], is an *NP-complete* [31] problem. Therefore RFC 3626 [20] suggests a heuristic for the calculation of the MPR set.

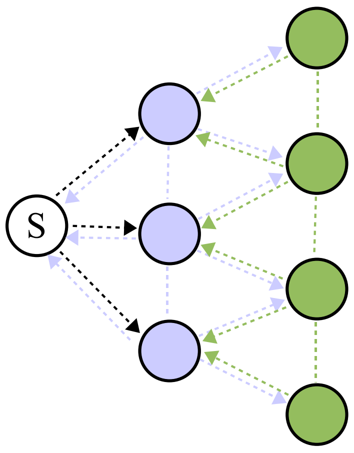

Figure 2.12: Flooding in Classical Link State Routing

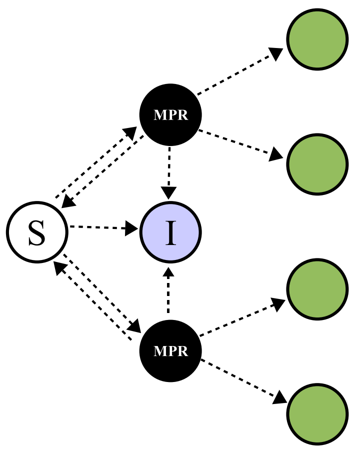

Figure 2.13: Optimised Flooding based on MPR Nodes in OLSR

Maintenance of Data Structures

Every node maintains a set of data structures as depicted in Table 2.4. Old entries time out and are removed from the sets. For maintenance of these data structures each node broadcasts periodically HELLO messages that contain the current set of neighbours and a flag that indicates whether the node can be reached via a bidirectional link. Furthermore, it includes a flag determining if the node has been selected as MPR node. When a node receives an HELLO message it updates its one-hop neighbour set and the two-hop neighbour set. Furthermore, the node refreshes the MPR selector set of neighbours that have chosen that node as an MPR node.

MPR Selection

Every node in the network selects independently its own set of MPR nodes from the one-hop neighbour set that have a bidirectional connection to that particular node and cover the set of two-hop neighbours. The heuristic algorithm works as follows. First, the one-hop neighbours are selected as MPRs that cover nodes that are not covered by any other one-hop neighbour. The nodes covered by the selected MPR node are removed from the two-hop set. If no such node exists the node that covers most of the nodes that still remain in the two-hop list is selected. If there are still multiple choices one node is selected randomly. This approach is continued until all nodes are covered by MPRs. Figure 2.14 illustrates the MPR selection

Sets	Description
neighbours	A set of all neighbours in the vicinity of a node. Neighbours are detected by periodical exchange of HELLO messages.
two-hop nodes	A set of one-hop and two-hop nodes that can be reached via a neighbour using a bidirectional link.
MPR nodes	Set of neighbours that have been selected as an MPR node by this node.
MPR selector	A subset of neighbours that have selected this node as an MPR node.

Table 2.4: OLSR Data Structures Maintained by Every Node

process running at node *S*. At first, node *S* selects node *A*, which is the only node that covers node *1*, and node *B*, which covers node *2* exclusively, as MPR nodes. The green nodes are covered and thus removed from the two-hop set. Nodes *C* and *D* both cover each other and node *4* and thus *C* is selected by chance finishing the algorithm. When node *S* broadcasts the next *Hello* message it will indicate that it has selected *A*,*B*,*C* as MPR nodes.

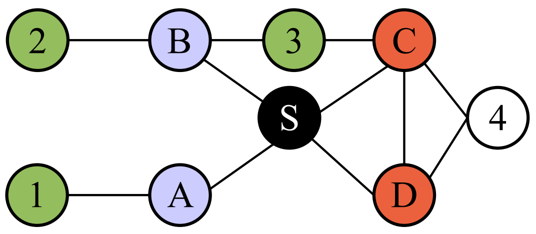

Figure 2.14: Selection of MPR Nodes in OLSR

Topology Control Messages

Only MPR nodes broadcast periodically jittered *topology control* TC messages containing the neighbours from the *MPR Selector Set*. The TC messages are processed by all nodes, but forwarded only by MPR nodes. The information about the nodes that have selected the MPRs is sufficient to create an entire network and then run Dijkstra's shortest path algorithm [28] locally at each node. The results are stored in the routing table and is used to send and forward unicast data packets.

OLSR Optimisations

The reduction of routing overhead is thus due to the following optimisations:

- In classical link state routing, every node broadcasts link state information. In OLSR, only MPR nodes periodically generate TC messages and thus yield fewer redundant retransmissions.
- In contrast to classical link state routing, only MPR nodes forward broadcast messages during the flooding process resulting in fewer broadcast messages.
- Partial link state information containing only the selector nodes is used in TC messages obtaining smaller packets.

2.3.4 Dynamic Source Routing

Dynamic Source Routing (DSR) [32] belongs to the reactive routing protocols and has currently reached the status of an Internet draft [21]. DSR employs source routes [6] and carries in the header of every packet a list of nodes, which comprises a path along which the packets should be forwarded by intermediate nodes. In contrast to AODV, refer to Section 2.3.5, where a new routing decision is made at every intermediate node, routing is only performed at the source node and intermediate nodes do not need to be aware of the route or keep up-to-date routing information. The basic algorithms consists of two phases *Route Discovery* and *Route Maintenance*.

Route Discovery

Since DSR is a reactive protocol it does not maintain a routing table with entries for *all* nodes in the network and thus needs to discover a new route on-demand. If a node would like to send data packets to a particular destination but does not have a valid route to that destination in its route cache, it broadcasts a *route request* RREQ throughout the ad hoc network and starts the *route discovery* process. The RREQ contains the source and destination address as well as a unique RREQ identifier chosen by the sender. Furthermore, a list of visited intermediate nodes along the route to the destination is stored in the RREQ. In the beginning this list is empty. At every hop, intermediate nodes add routing information to the RREQ that make up a path from the source to the destination. Eventually, the RREQ holds the complete list of visited nodes when it arrives at the destination node, the *source route*. If the underlying ad hoc network just supports bidirectional links the destination node can use the path learned from the RREQ to send data packets to the source.

Figure 2.15 illustrates the dissemination of the RREQ that incloses the address of the source node *S*. The intermediate nodes *1* and *2* receive the RREQ and populate their routing table with the gained routing information from the RREQ. Next, they add their own address to the RREQ and broadcast it, as depicted in Figure 2.16. Node *5* receives a message from node *1* and *2*. Due to the source routes included in the RREQ, node *5* obtains additionally two different routes back to node *S*. Next, node *5* adds its own address to the RREQ and forwards the message again.

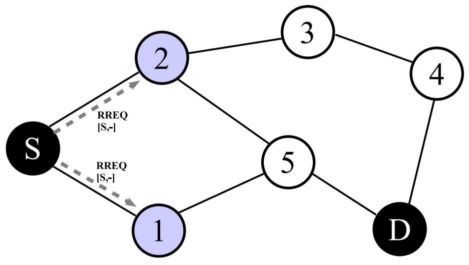

Figure 2.15: Route Discovery in DSR based on Route Requests

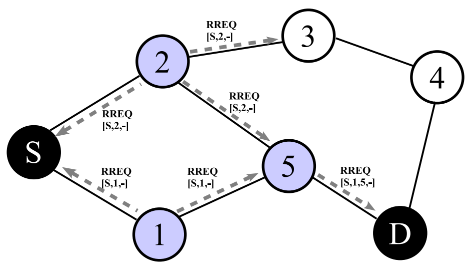

Figure 2.16: Intermediate Nodes Add Their Addresses to the Route Request

The destination replies to the RREQ and sends a *route reply* message RREP back to the source. Either the RREP is source routed along the path the RREQ took as depicted in Figure 2.17 or DSR sends the RREP on a different route back to the source. Hence, DSR can support unidirectional links for the forward and reverse route. In that case, the destination node discovers a new route to the destination itself. In order to prevent for routing loops, that might happen if both, source and destination reply to an RREQ by sending another RREQ, the destination node encapsulates the RREP in the RREQ as illustrated in Figure 2.18. Node *D* broadcasts the RREQ that piggybacks the original RREP asking for a route to the node *D*. Since the link between *1-5* is unidirectional as depicted in Figure 2.18, node *S* receives the RREP over the route *D-5-2-S*. Next, node *S* sends an RREP using the confirmed route *S-1-5-D* to unicast the RREP back to the destination node. Thus, route *S-1-5-D* is applied by node *S* to send data packets to node *D*. However, node *D* employs the route *D-5-2-S* for transmitting data packets to node *S*. In order to reduce the routing overhead caused by dissemination of RREQs intermediate nodes might reply to an RREQ if they have up-to-date routing information. Furthermore, they can learn new routes from ”passing-by”RREQs.

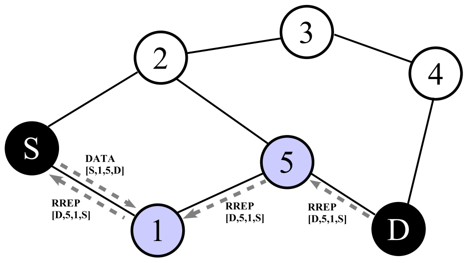

Figure 2.17: Forwarding of the Route Reply Message in DSR

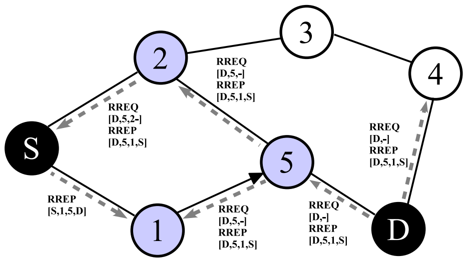

Figure 2.18: The Route Request Piggybacks the Route Reply

Route Maintenance

When a node detects a broken link and thus cannot forward a packet according to the route in the header of the packet it sends a *route error* packet RERR back to the source node. The source node can either start *Route Discovery* again or can attempt to use a different source route it already has in its cache. Figure 2.19 illustrates the route maintenance procedure. Node *S* keeps two source routes to the destination *D*, *S-2-5-D* and *S-1-5-D*. The former route has been selected by node *S* to send packets to *D*. However, node *2* detects the broken link between nodes *2-5* and transmits an RERR to notify node *S* about the broken link. Node *S* has another route to the destination node *D* in its route cache and starts sending the data packets using the alternative route *S-1-5-D* indicated in red.

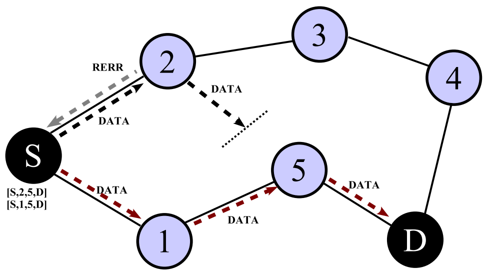

Figure 2.19: Route Maintenance via Route Error Messages in DSR

2.3.5 Ad Hoc On-Demand Distance-Vector

Ad Hoc On-Demand Distance-Vector (AODV) [4] is a reactive routing protocol for ad hoc networks and was one of the first protocols that obtained the status of an RFC (RFC 3561). Many implementations for a variety of operating systems and network simulators are available. AODV has been designed by the same people who invented DSDV and the primary aim was to reduce the amount of system-wide broadcasts and therefore the core AODV routing algorithm does not depend on periodic system-wide advertisements. It provides loop-free routes and adapts quickly to topology changes. By default, AODV requires bidirectional links, however, extensions exist to enable AODV to cope with unidirectional links. AODV is a combination of DSDV and DSR. It employs the on-demand route discovery as in DSR and hop-by-hop table-driven routing as in DSDV. In contrast to DSR, each node participating in a route from node A to node B does not store the complete route in the routing table but just the next hop and the hop count to the destination.

Route Discovery

If a node does not have a valid route to the destination it broadcasts a *route request* packet RREQ to its neighbours containing a *request id* to identify the RREQ, its own *sequence number* incremented by one and the last known sequence number of the destination. A recipient of an RREQ verifies if a packet with the same *request id* has already been received. If this is the case, the packet is dropped. However, if the packet has not been received before, the hop count of the RREQ is incremented by one and the source address of the packet is added to its routing table, thus a *reverse route* is created that can be used to send packets into the opposite direction from the destination to the source. The first steps of the route discovery process are depicted in Figure 2.20. Node S requires a route to node D and starts the route discovery process by broadcasting an RREQ. Node *1* and *2* receive the RREQ update their routing table and create a *reverse route* to node S, indicated as green arrows. Then, node *2* and *1* forward the RREQ and node *3* and *5* receive the request and create a reverse route as illustrated in Figure 2.21 by red arrows. Node *5* has received the RREQ from node *1* earlier and thus, ignores the RREQ from node *2* and does not create another reverse route.

The RREQ packet is broadcast again until it reaches the destination or an intermediate node that already knows a *fresh* route to the destination. The route is fresh if the sequence number is as high or higher than the one used in the RREQ. Either an intermediate node with a fresh enough sequence number or the destination itself sends a unicast *route reply* message RREP to the source node using the routing information from the received RREQ. The *reverse route* that has been created by propagating the RREQ is used to send the RREP back to the source node. This time, however, intermediate nodes receiving the RREP create a *forward route* to the destination as depicted in Figure 2.22. The destination node D receives the

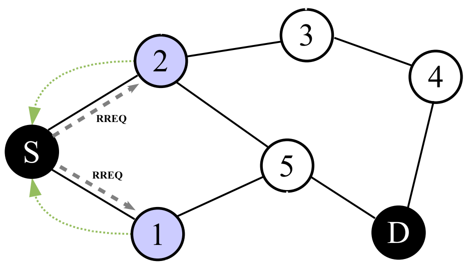

Figure 2.20: On-Demand Route Discovery by Sending Route Requests

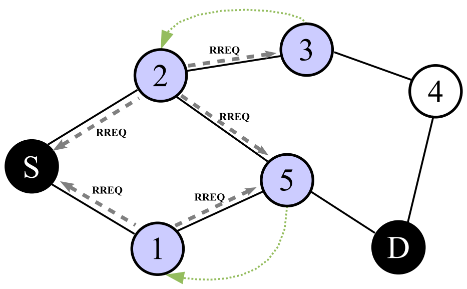

Figure 2.21: Intermediate Nodes Forward RREQ and Create Reverse Routes

RREQ and creates a reverse route. Then it unicasts a RREP to node *5* using that reverse route. Node *5* itself creates a *forward route* indicated as a red arrow and forwards the RREP along the reverse path that has been established before.

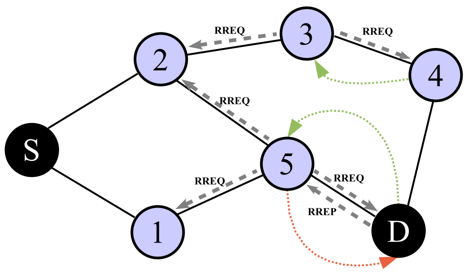

Figure 2.22: The Destination Sends a Route Reply Back to the Source Creating a Forward Route

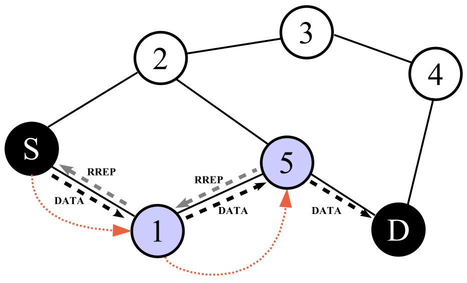

Figure 2.23: The Source Receives the Route Reply and Starts Transmitting Data Packets

As soon as the source node *S* receives an RREP it can start its data communication with the destination since both *forward route* and *reverse route* have been created as illustrated in Figure 2.23. The source node *S* receives the RREP and starts transmitting data packets along the new route *S-1-5-D*. If the source does not receive an RREP within the route discovery timeout the process is started again.

Route Maintenance

RFC 3561 [4] suggests either to employ HELLO messages or link layer mechanisms to detect broken links. Broken link detection by the link layer is much quicker and does not generate additional signalling overhead. However, support from the link layer might not be always available and to make no assumptions on the underlying link layer HELLO messages are often employed. If several successive HELLOs have not been received the link is considered as broken and active neighbours that currently use that link are notified by a *route error* message RERR. The set of active neighbours is maintained in a *precursor list* that has been established during *Route Discovery*. The predecessors forward the RERR to their predecessors and so on until it reaches the source node. The RERR contains the IP addresses of nodes that have become unavailable and their sequence number that is increased by the node that has detected the link failure. Figure 2.24 depicts a scenario where the link between node *5* and the destination has been broken. Node *5* sends a RERR to the precursor node *1* that itself sends a RERR to the source which might start route discovery to find a new route. Besides, node *5* might decide to repair the broken link locally as illustrated in 2.25. In this case, node *5* broadcasts an RREQ to its neighbours and node *2* answers with an RREP. Since the new route is much longer node *5* sends an RERR to notify the source about the changed route. Node *S* might decide to continue to use the repaired path or start a new route discovery process.

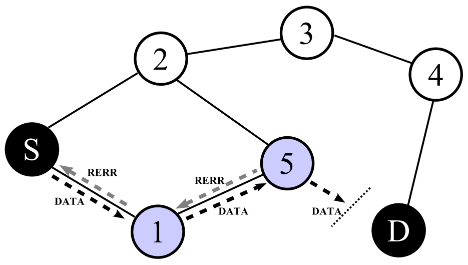

Figure 2.24: Route Maintenance in AODV in Case of Broken Links

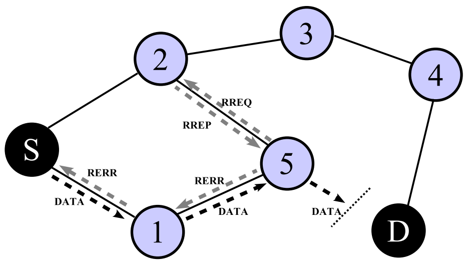

Figure 2.25: Local Repair Carried Out by an Intermediate Node

Parameters

AODV provides many parameters that have to be adapted to the actual scenario to yield the optimal performance. In particular, AODV relies on timers to distinguish from old routes and keep the routing table as small as possible. Table 2.5 illustrates some of the important parameters.

2.3.6 Comparison of Routing Protocols

All discussed routing protocols provide for loop-free routes and use the hop count as a metric to decide which route to the destination is the best one.

Parameter	Description
Net Diameter	The NET_DIAMETER determines the maximum number of hops between two nodes in the network and is used by AODV to calculate timers such as the MAX_REPAIR_TTL and NET_TRAVERSAL_TIME. The default value is 35.
Active Route	AODV makes heavily use of timers to maintain the routing table which is updated every time a data or routing packet is sent, received or forwarded. ACTIVE_ROUTE_TIMEOUT specifies the time interval a route is marked as active.
Hello Loss	If HELLO messages are employed for broken link detection, this parameter determines the number of successive HELLO messages a node accepts until the link is considered as broken.
Hello Interval	The time interval a node sends HELLO messages. The default value is 1 second.
Route Discovery	Time interval a node waits until the route discovery is considered as failed and another RREP is broadcast to find a route.
Route Delete	Time interval after an invalid route is removed from the routing table.

Table 2.5: AODV Routing Protocol Parameters

DSDV and OLSR benefit from the proactive approach of having routes to all destinations available and thus new connections are set up quickly. DSDV is computationally efficient and easy to implement with moderate memory requirements $O(n)$, whereas n is the number of nodes. It recognises one-way links and uses bidirectional links only. However, the worst-case convergence behaviour of DSDV is not optimal and it may take too long until a change in the topology has been propagated throughout the network [19]. In contrast, OLSR is fairly complex and the calculation of the MPR nodes and the shortest-path algorithm require continuous processing power and puts memory burdens on mobile devices. On the other hand, due to the reduction of broadcast routing messages, OLSR is applicable for large and dense networks. Moreover, link-state routing has been considered more stable than distance-vector as mentioned in [6] and allows OLSR to converge within reasonable time. DSDV is no longer a potential candidate for a standardised MANET routing protocol and the use of OLSR is recommended instead [20]. Both protocols have the disadvantage, that messages are exchanged periodically, although the topology of the network might not have changed. Furthermore, the routing tables hold entries for all destinations in the network even though nodes will very unlikely communicate with all nodes. The reactive protocols AODV and DSR in contrast discover routes on-demand and tend to generate less routing overhead, however, with the cost of additional delay during the set up of a new connection.

The table-driven protocol AODV is more conservative than DSR and always uses sequence numbers to distinguish between fresh and old routes, whereas DSR suffers from pollution by stale routes since it does not have a mechanism to expire old routes. However, AODV requires more route requests, since during a route discovery cycle the nodes along the path just learn a route to the next hop, source and destination node. In contrast to DSR, where every node automatically obtains a route to every other node on that particular route. Therefore, AODV has to rely on RREQs more often than DSR. Furthermore in DSR, the destination node replies to all route requests and thus, the source will receive multiple routes that can be employed as backup routes. AODV replies just to the first RREQ and thus, the source will not gather routing information about further alternative routes. On the other hand, source routing as employed by DSR adds some routing overhead to every data packet. This might be rather inefficient if the data packets are very small as it is the case with real-time multiplayer game traffic [33].

In contrast to DSR and DSDV, AODV and OLSR are both standardised and accepted as experimental RFCs by the IETF and several implementations for network simulators and operating systems exist. As a result of this comparison one has to conclude that there does not exist an optimal routing protocol for all scenarios and it is always a trade-off between many factors. Table 2.6 illustrates the features and compares the four protocols discussed in this section. The proactive protocols rely on periodic routing control messages. The reactive protocols, in contrast, make use of periodic HELLO messages for neighbour management only if the employed MAC layer cannot provide the required information. Besides the features listed in the table variants of the standard routing protocols exist, for example, multicast support for AODV. Furthermore, an expired draft version of a QoS enabled AODV has been proposed in [34]. Currently, none of the mentioned routing protocols deal with security.

Feature	**DSDV**	**OLSR**	**DSR**	**AODV**
Proactive	$\surd$	$\surd$	$\times$	$\times$
Reactive	$\times$	$\times$	$\surd$	$\surd$
Loop free	$\surd$	$\surd$	$\surd$	$\surd$
Source Routes	$\times$	$\times$	$\surd$	$\times$
Multiple Routes	$\times$	$\times$	$\surd$	$\times$
Unidirectional Links	$\times$	$\times$	$\surd$	$\times$
Metric	hops	hops	hops	hops
Periodic messages	$\surd$	$\surd$	$(\times)$	$(\times)$
Multicast	$\times$	$\times$	$\times$	$(\times)$
QoS Support	$\times$	$\times$	$\times$	$\times$
Security	$\times$	$\times$	$\times$	$\times$

Table 2.6: Routing Protocol Comparison (DSDV, OLSR, DSR and AODV)

2.4 QoS in MANETs

This section describes particular QoS provisioning challenges in MANETs and explains why common QoS mechanisms employed in wired networks cannot by easily applied in mobile wireless multi-hop networks. The following subsections introduce QoS aware routing protocols and QoS architectures that have been proposed for MANETs.

2.4.1 Challenges

As mentioned in Section 2.1, the aim of QoS provisioning is to fulfil certain demands on the network and provide guaranteed packet delivery ratio, available bandwidth, latency and jitter. However, giving QoS guarantees in MANETs is a very challenging task due to the dynamic network topology and the lack of any central coordination which might yield in imprecise network state information. Moreover, nodes have to share the error-prone wireless medium using distributed medium access protocols, such as IEEE 802.11 DCF, instead of centralised MAC protocols like *Time Division Multiple Access* (TDMA). To provide QoS guarantees it is mandatory that all lower layers support QoS as well. For instance, the IEEE 802.11 DCF is not QoS enabled. Even higher layers that have QoS extensions cannot give hard QoS guarantees since they do not receive any QoS assurance from the MAC layer. Furthermore, the available bandwidth of the shared medium is still scarce and more and more devices and technologies are competing for the shared medium.

Some QoS mechanisms that are widely employed in wired networks cannot be used in MANETs because of the special properties of MANETs. In wireless networks, the bandwidth of the shared medium is still precious and over-provisioning that is the simplest approach in wired networks, is very bandwidth consuming. In addition to that, resource reservation introduces message and state overhead at every node. Since the topology of MANETs might change frequently reserved resources might not be used efficiently. Furthermore, although resources for instance bandwidth has been reserved at one particular node A there is no guarantee to receive the bandwidth because other nodes in the vicinity of the node A might send packets as well and do not know about the reserved bandwidth at node A. Complex protocols that take this into account suffer from the inability to cope with high degree of mobility and from message overhead.

2.4.2 QoS Routing Protocols

QoS enabled routing protocols attempt to provide routes that meet the QoS demands of the application and recognise and communicate QoS violations to other nodes. In general, routing protocols employ the hop count as the only metric to find the shortest path from the source to the destination. The hop count, however, does not directly relate to latency, jitter and loss rate and even a route with a very low hop count might experience high latencies and vice versa. Therefore, instead of using the number of hops, latency, jitter, queue length, signal strength, link loss rate etc., might be more appropriate to find a route that meets certain QoS demands. In [35] the authors distinguish between three different types of metrics or path constraints. Let $(u, u_1, u_2, ..., u_k, v)$ be the path connecting node u and v. Then, $m(u, v)$ is the performance of the metric m on the path (u, v).

Additive constraints: the hop count and latency, for example, are additive constraints. The summation of the end-to-end delays for every hop along the path makes up for the total delay. The following equation must hold for additive constraints:
$m(u, v) = m(u, u_1) + m(u_1, u_2) + ... + m(u_k, v)$

Multiplicative constraints: a constraint is multiplicative if the constraint equals the product of the single-hop metric, for example, the loss rate:
$m(u, v) = m(u, u_1) * m(u_1, u_2) * ... * m(u_k, v)$

Concave constraints: the available bandwidth is a concave constraint since it is defined as the minimum bandwidth between the links on the path (u, v):
$m(u, v) = min(m(u, u_1), m(u_1, u_2), ..., m(u_k, v))$

If the routing protocol has to follow multiple constraints to meet an application's QoS demands, the optimal path selection algorithm gets very complex, especially, if more than one additive or multiplicative constraint is used. In that case the problem has been shown to be NP-Complete [36, 31].

QoS for AODV

In [34] the authors suggest to extend the RREQ of the reactive routing protocol AODV by *QoS Objects* to specify latency, jitter and bandwidth demands for a certain path. Every RREQ additionally carries a *session ID* to identify every single flow. An intermediate node needs to be aware of average latency and jitter values and subtracts the average value from the accumulated value of the QoS parameter in the RREQ, for instance, the accumulated delay. If the value is negative the RREQ is not forwarded and discarded. If a node cannot provide the agreed QoS any more it sends a control packet to the source including the *session ID*.

The problem of neighbourhood bandwidth utilisation has not tackled and remains as an open issue. Furthermore, intermediate nodes are not allowed to reply to RREQ carrying QoS extensions and have to forward the RREQ to the destination node to reserve the resources along the path to the destination, resulting in much higher routing signalling overhead.

Ad Hoc QoS On-Demand Routing

Ad hoc QoS on-demand Routing (AQOR) [37] follows a similar approach and tries to accurately measure the available bandwidth and end-to-end delay of the wireless channel to reserve resources and provide QoS guarantees. In order to calculate the available bandwidth at a certain node, AQOR employs periodical HELLO messages that are exchanged with the neighbours. Resources, such as bandwidth and end-to-end delay are reserved temporally for each flow during route discovery and activated if the first data packets are transmitted along a certain path. Furthermore, reserved resources are released as soon as they are not used any longer. QoS violations in delay are detected at the destination as every packet is marked with a time-stamp of the sender. During route discovery the round-trip time (RTT) is determined and from that the offset of the system clocks of the source and destination host is calculated, assuming symmetric links. AQOR is compatible to today's best-effort MAC protocols but will work with QoS aware MAC protocols as well.

Further QoS Protocols

The hierarchical routing protocol CEDAR [24] provides QoS route computation with certain bandwidth requirements by *core* nodes. First, the core nodes are distributedly elected and form an approximated dominating set [30]. The core nodes propagate link state information of stable and high bandwidth routes. During route computation the core nodes provide a route that satisfies the requested bandwidth.

In [38] multi-path extensions for MANET routing protocols are described. The aim is either to load-balance or to minimise latency by sending the traffic over multiple links. In reactive protocols, for instance AODV, multi-path routing can be employed to provide a backup route in case of broken links [39]. Load-aware protocols on the other hand take the interface queue length, contention delay, number of neighbours and traffic pattern of the neighbours into account to select the best path, for instance, *Load-Aware On-Demand Routing* (LAOR) and *Load Sensitive Routing* (LSR). Besides, the *Signal Stability-based Adaptive* (SSA) [40] routing protocol measures the signal strength of BEACON messages to choose links with the best signal to noise ratio (SNR). On the other side, link-stability aware routing protocols select routes with the longest lifetime [41].

2.4.3 QoS Frameworks

In contrast to the QoS routing protocols described in the previous section, this section concentrates on existing stateful and stateless QoS frameworks for MANETs. IntServ, introduced earlier, requires too much processing, memory and signalling overhead and cannot be applied in MANETs. But using DiffServ instead is not appropriate since the peculiarities of wireless multi-hop networks have to be taken into account.

SWAN

Service Differentiation in Stateless Wireless Ad hoc Networks (SWAN) [42] is a stateless QoS architecture that provides soft real-time QoS relying on standard best-effort MAC technology, such as IEEE 802.11 and is independent of the employed routing protocol. It avoids additional signalling and complex control mechanisms. Furthermore, data packets pass a classifier that distinguishes between real-time or high priority and best-effort or low priority traffic. Moreover, it makes use of rate control for low priority traffic and employs admission control for high priority UDP traffic. Instead of packet loss as indicator of congestion, SWAN measures the MAC delay of ACK frames and uses this information to configure the rate controller of low priority traffic. In addition, it counts the number of flows passing through the neighbours. Admission control of high priority traffic is carried out only at the source node which sends a PROBE message to the destination specifying the desired bandwidth. Nevertheless, false admission might happen if several nodes send PROBE messages at the same time. To indicate a QoS violation or congestion, nodes can set the *Explicit Congestion Notification* (ECN) flag [43]. If the destination receives too many packets with the ECN flag set it notifies the source that might stop or adapt the transmission to the network conditions or probes for a better route. In [44] the authors suggest to combine DiffServ and SWAN to be able to distinguish between different real-time flows and handle them separately.

INSIGNIA

In contrast to the stateless QoS framework SWAN, the idea behind INSIGNIA [45] is to provide a stateful or reservation-oriented lightweight QoS architecture. The key component is the in-band signalling protocol that is employed similarly to the out-band resource reservation protocol RSVP but does not require additional signalling. Instead, control signals are encapsulated into IP data packets with an IP INSIGNIA option for doing the necessary resource reservation. Adaptive soft-state timers are employed to determine for how long a resource has been reserved. Every data packet that belongs to a flow with QoS demands refreshes the soft-state timers. Furthermore, the experienced QoS is measured at the destination node that can send QoS reports to the source to adapt the flow reservation if necessary. For the routing protocol, any of the protocols discussed in Section 2.3 can be used which shows the modular architecture and separation between routing, signalling and admission control in INSIGNIA.

Flexible QoS Model for MANET

A combination of IntServ and DiffServ is the QoS framework *Flexible QoS Model for MANET (FQMM)* [46]. Flows with very high real-time requirements follow the flow-based approach of IntServ. Traffic with lower QoS demands are handled in the service differentiation style of DiffServ. The classification of the traffic is carried out at the source node. On the one hand FQMM attempts to exploit the benefits from both IntServ and DiffServ in a hybrid approach, on the other hand, the same problems of IntServ and DiffServ remain and in [47] the authors claim that FQMM does not cope any better with the particular conditions in MANETs.

iMAQ

The *Integrated Mobile Ad-hoc QoS framework* (iMAQ) [48] makes use of a cross layer design of the location-aware routing and middleware layer. The framework relies on a location-based QoS routing protocol to predict movement of the nodes and network partitioning. Furthermore, it includes an additional middleware for data replication and service lookup on mobile devices attempting to provide the best service for data-accessibility to the users. The update protocol floods location and resource information throughout the network. To minimise the flooding MPR as in OLSR is used.

2.5 Multiplayer Games

Multiplayer games are gaining much interest and almost every new game can be played in a LAN or the Internet with other players. There are two types of multiplayer-games: (1) round-based, like chess; and (2) real-time, such as car-racing simulations. This thesis focuses on real-time multiplayer games which can be itself classified into (a) first person shooters (FPS), (b) real time strategy (RTS) and (c) sports games. In the following sections the architectures of real-time multiplayer games and the requirements on the mobile wireless network are discussed.

2.5.1 Real-Time Game Architectures

Traditionally, multiplayer games follow either the client-server or the peer-to-peer approach to maintain the game state.

In the *client-server* game architecture, all clients connect to one central server that acts as a master of the game. The central server receives game state messages from the clients, for example avatar movements, and the server verifies that the actions are compliant to the game rules. Next, the central server recalculates the global game state upon the information received from the clients and finally shares the new game state back to the clients. This architecture suffers from mainly two problems which disqualifies it to be employed in mobile wireless multi-hop networks. First, the central server is a single point of failure, that is, if the server is down or out of transmission range some clients cannot connect to the server any more and play the game. Second, the centralised architecture does not scale if all clients have to connect to the same server.

In the *peer-to-peer* approach every node or so called peer is both client and server at the same time. Every peer distributes its actions to all other peers in a completely distributed manner without the need of a central server or any central coordination. Thus, each peer maintains the game state on its own. This architecture avoids the single point of failure, however, the synchronisation with all other peers in the network requires much bandwidth and does not scale either. In addition, peers can easily cheat, since there is no central master server that coordinates the game.

A combination of both is the *zone server architecture* [49] that attempts to exploit the benefits of both approaches mentioned before and provides a robust game architecture for mobile wireless networks. Instead of just one single server, multiple *zone servers* are responsible for the clients as illustrated in Figure 2.26. Every client connects to the nearest zone server (Z_1 or Z_2) and sends game state changes. The zone server recalculates the next game state and disseminates the new game state to all clients via the other zone servers. If a zone server goes down, the according clients can continue the game and connect to another zone server. The problem of selecting the best appropriate zone servers has been simulated and implemented in [50].

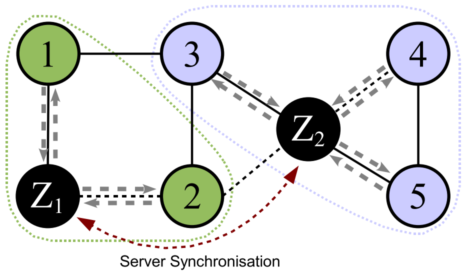

Figure 2.26: Zone Server Game Architecture

2.5.2 Network Requirements of Real-Time Multiplayer Games

The actual demands of multiplayer games on the network highly depend on the game type. In case of real-time multiplayer games, end-to-end communication delay, jitter and packet loss are the most relevant QoS attributes of the network, while the available network bandwidth is of less importance [33, 51] because current wireless networks provide enough bandwidth.

For most real-time multiplayer games, a maximum of 150 ms round trip delay is acceptable [52]. However, the still tolerable delay highly depends on the particular game, for instance, in [53] 50 - 100 ms round trip delay has been reported to be the limit for real-time car-racing simulations. This is due to the fact, that some games can cope better with latency and jitter and make use of adaptive prediction mechanisms, such as *dead-reckoning* [54]. In addition to the actions, the client sends a vector, comprising for example speed and direction of the avatar to the server that can anticipate the next positions if it has not received data from the client recently. Therefore, data that arrives too late for the current game state might be still useful to predict the current position. In general, FPS and other action games have higher demands on latency than RTS games that can still cope with a round trip delay up to 300 - 500 ms [55, 51].

The impact of jitter on real-time multiplayer games has not been scrutinised in depth, yet. Jitter and latency are strongly coupled for games in the Internet [56] and most players focus on reducing latency in selecting an appropriate server with low round-trip time delays. Furthermore, players' perception of jitter is game-dependant, as reported in [57]. In general, a high level of jitter leads to packets not arriving in time thus requiring the use of prediction mechanisms, as mentioned before. These prediction mechanisms, however, cannot always anticipate players' actions accurately. Therefore, a high level of jitter degrades the players' experience and must be kept as low as possible.

Moreover, packet loss rate shows a similar impact on real-time multiplayer games. According to [52], it should be kept below 3 - 5 %, depending on the game.

Other real-time applications like audio conferencing have similar demands on the network and can be also used by a QoS framework that provides for real-time multiplayer games. Additionally, many multiplayer games provide online audio conferencing for team playing.

2.6 Summary

This chapter has presented the necessary background information for the following chapters of the thesis. In the first section, the principle of Quality of Service has been introduced and the demands of popular networked applications have been compared. The basic mechanisms that are employed to provide QoS have been described and two popular QoS frameworks namely, the stateful reservation-oriented IntServ that can provide hard QoS guarantees and the stateless DiffServ approach that provides probabilistic guarantees, have been discussed. Afterwards, the basics of wireless communications and the challenges that arise, such as the hidden and exposed terminal have been explained. As this thesis focuses on IEEE 802.11 as the wireless technology for MANETs, the essential parts of the IEEE 802.11 ad hoc mode have been discussed. A routing protocol is mandatory to allow for multi-hop communication and reactive and proactive routing protocols for MANETs have been described and compared in Section 2.3 These protocols will be simulated and evaluated with respect to real-time multiplayer game scenarios in the following chapters. In Section 2.4 the related work on QoS routing and QoS frameworks for MANETs has been presented and many of the QoS mechanisms will be applied and analysed in the course of the thesis. Finally, the last section introduced the architectures and QoS requirements of real-time multiplayer games. The aim of this thesis is to provide QoS mechanisms and QoS extensions that allow for playing real-time multiplayer games in mobile wireless multi-hop networks. The employed and analysed QoS mechanisms are described in the next chapter.

Chapter 3

Design Concepts

Real-time applications, especially multiplayer games such as first person shooters, real-time strategy games, or sports games have strict Quality of Service demands on the underlying network. Moreover, in wireless mobile ad hoc environments, compared to wired networks like the Internet, these applications encounter additional problems due to the special challenges reserved by MANETs. In this chapter, the design of common and some new QoS extensions which are supposed to improve the performance of mobile ad hoc routing is described. The following chapters explain the implementation details and simulation results in the network simulator NS-2 [5].

3.1 Objective and Challenges

The objective is to extend mobile ad hoc routing with QoS mechanisms by which the performance of the ad hoc routing protocol and the network can meet the demands of real-time multiplayer games. As initial simulations in Section 5.2 show, latency and loss rate of real-time traffic are far too high and multiplayer games and other real-time applications cannot be used in mobile wireless networks without further improvements. Therefore, QoS mechanisms are needed that improve the performance and reduce latency and loss rate by one order of magnitude to meet the demands of real-time multiplayer games (refer to Section 2.5). However, giving QoS guarantees in MANETs is a difficult task and potential QoS extensions have to face the following challenges:

Mobility: the topology of the ad hoc network might change rapidly and unpredictably resulting in broken links and stale routes. The routing protocol has to detect broken links quickly and discover a new route if necessary. Additionally, the routing protocol should not make decisions upon global network state information since it might be outdated and very expensive to keep up-to-date.

Congestion: real-time traffic must arrive in-time even if the network is highly loaded. Although, the bandwidth of wireless networks is increasing constantly, it is still much lower than in wired networks where every node has a dedicated bandwidth. Hence, it is crucial to avoid wasting scarce bandwidth and keep routing and management packets to a minimum.

Shared Medium: IEEE 802.11 wireless networks operating in ad hoc mode do not provide any QoS guarantees at the MAC layer due to the applied contention based medium access mechanism [3]. Furthermore, there is no central administration in ad hoc networks like access points and thus applied algorithms have to be completely distributed.

Wireless Signal: the wireless signal suffers from fading and interference which gives rise to grey zones [17] and frequent retransmissions.

Resource reservation (refer to Section 2.1) as done in IntServ is very expensive in mobile wireless networks because some signalling is required to reserve the resources at every node the traffic flow passes through. Due to the mobility of the nodes the network's topology can change and makes maintenance of reserved resources very difficult. It is very likely that after a change in topology either too many resources or not enough resources have been reserved. Constantly running a resource reservation protocol is impractical and consumes too much bandwidth. The in-band signalling approach followed by INSIGNIA (refer to Section 2.4.3) to reserve resources is very interesting but requires an additional IP-Options header for every IP packet. Therefore, a trade-off has been made and a stateless QoS approach that just relies on local network state information has been followed. Furthermore, the proposed QoS extensions aim at minimising periodic broadcast messages and routing protocol signalling. *Ad Hoc On-Demand Distance-Vector* (AODV) [4] has been employed as the routing protocol since it has shown the best performance results in the routing protocol comparison (refer to Section 5.2).

As QoS provisioning is a complex task and should be handled on several layers in the protocol stack, a cross-layer design has been used in this thesis. Thus, the collected and proposed QoS extensions provide QoS support on the routing, interface queue and MAC layer, as illustrated in Figure 3.1. They can be classified into the following categories: (1) AODV Enhancements; (2) Traffic Management; and (3) MAC Layer Support.

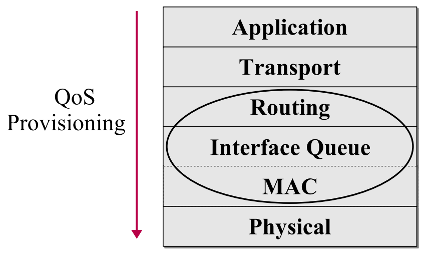

Figure 3.1: Quality of Service in a MANET Node's Protocol Stack

Moreover, the different QoS extensions address different challenges of the mobile ad hoc environment. Table 3.1 depicts the extensions and the corresponding challenges they are supposed to tackle.

QoS Mechanism	**Mobility**	**Congestion**	**Shared Medium**	**Signal**
AODV Enhancements				
Local Repair	√			
Backup Route	√			
Traffic Management				
Priority Queueing		√		
Timeouts		√		
Rate Control		√	√	
Mac Layer Support				
Broken Link Detection	√	√		
Neighbour Detection	√	√		
Signal Strength Monitoring				√
RTS/CTS Adaptation			√	

Table 3.1: QoS Extensions

3.2 AODV Enhancements

As the results of the ad hoc routing protocol comparison in Section 5.2 show, the reactive protocol *Ad Hoc On-Demand Distance-Vector* (AODV) [4] provides the best overall performance compared to DSR, DSDV and OLSR. Therefore, the AODV-UU [58] implementation has been selected as the starting point of the work and extended by additional QoS mechanisms (refer to 2.3.5 for an introduction to AODV).

Because of its on-demand route discovery behaviour the routing overhead is much smaller than in proactive routing protocols. Only network state information for active routes is maintained, in contrast to proactive protocols, where every node maintains routes to all other nodes. If more broken links are detected more routing messages are disseminated for discovering and maintaining routes. However, if no routes are broken, AODV does not sends routing messages. Thus, AODV scales automatically with the mobility rate. Furthermore, AODV has already achieved the status of an RFC [4] and implementations are freely available for several platforms and operating systems. To cope with mobility and broken links and further increase the performance, AODV makes use of a *local repair* mechanism. In addition, AODV has been extended with the use of *backup route*s to repair broken links transparently and without any delays.

3.2.1 Local Repair

RFC 3561 suggests a local repair mechanism for AODV that allows an intermediate node that detects link failures, to queue packets temporally while trying to repair the route. This mechanisms is illustrated in Figure 2.25 and has been explained in Section 2.3.5 in more detail. The advantage of local repair is that packets that already have been sent over multiple hops are not dropped if the intermediate node can find an alternative route to the destination. As a result, the source node does not have to retransmit affected packets. In the best case, the mechanisms is totally transparent to the source node and the application will not experience a degraded performance. On the other side, repaired routes are likely to be longer than the original route and thus a RERR message has to be sent to the source node which will start the route discovery process as well. Furthermore, real-time packets might be delayed too much if they are queued locally and will not arrive in-time.

3.2.2 Backup Route

AODV just stores the next hop entry to a certain destination in its routing table (refer to Section 2.3.5). If the link to that node breaks a new route discovery process has to be started which might require too much time for real-time data. The idea is to provide a backup route right from the beginning that can be used instead. A backup route is a path with the same hop count as the default path but with a different next hop. During the route discovery a node might receive multiple RREQ from different neighbours. Nodes, that receive a RREQ create a *reverse path* back to the source in its routing table, as mentioned in Section 2.3.5. However, if the particular RREQ has already been received from a different neighbour, it is only processed again if the hop count is lower. Hence, the node is not aware of an alternative route. For example, assume a scenario as illustrated in Figure 3.2. Node *S* broadcasts a RREQ message to find a route to node *D*. Node *I* receives two RREQs, one from node *2* and the other from node *4*. Both describe a 3-hop path to node *S* with the same sequence number. In this event, node *I* employs either node *2* or *4* as the next hop and uses the other as backup route. If a link to a neighbour on an active route breaks the backup route becomes the new active path. If hops with a higher hop count were used as backup paths, routing loops might occur as in the following example. Node *I* receives a RREQ from node *5* with a 5-hop path to node *S* but this path contains a routing-loop and cannot be employed as backup route.

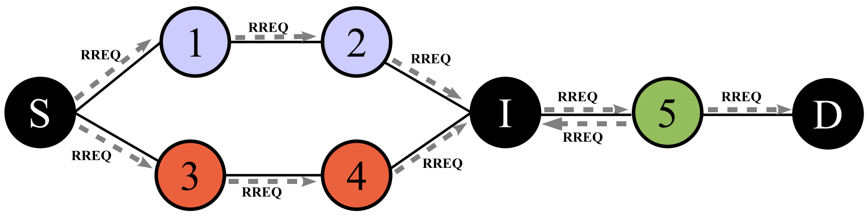

Figure 3.2: AODV Extension Using Backup Routes

3.3 Traffic Management

The goal of traffic management is to differentiate among various types of traffic and give a higher level of support to high priority traffic at the cost of lower priority traffic. When employing *priority queueing*, real-time packets are preferably transmitted to make sure that even in situations with higher load real-time packets do not become obsolete. Still, it might happen that real-time packets have to wait too long in the high priority queue. To prevent for transmission of outdated packets, hop constrained queue *timeouts* are used to drop obsolete real-time packets and thus save bandwidth. In addition, to limit the amount of low priority traffic, real-time neighbour aware *rate control* policies are employed, which prevent the occupation of the communication channel by nodes sending low priority traffic if other nodes have high priority traffic to be sent.

3.3.1 Priority Queue

To provide priority queueing mechanisms the interface queue sublayer of the network simulator NS-2 [5] has been modified. The interface queue consists of three sub-queues with different priorities for real-time game traffic (high priority), routing control messages (medium priority) and best-effort traffic (low priority). Real-time game traffic and routing control messages are handled in separate queues to allow for load-aware routing. Nodes that do not send as much high priority traffic will respond quicker to RREQ messages than nodes that already forward multiple real-time traffic flows. Every packet is classified into one of the three categories, as shown in Figure 3.3, employing the *Type of Service* (TOS) field in the IP header. All data packets that do not have any particular real-time QoS demands are marked as *low priority*, for instance, file transfer and e-mail traffic. AODV routing control packets such as RREQ, RREP and RERR packets are marked as *medium priority*. And finally, real-time data packets are marked as *high priority* as well as AODV RREP messages if the high priority flag had been set in the corresponding RREQ. In contrast to medium and low priority, high priority packets are quickly outdated and dropped if they cannot be delivered in-time. Low priority packets might on the one hand experience higher delays, but on the other hand they are forwarded and delivered to the destination more reliably. Every sub-queue has a limited size and allows a packet to be queued for a certain time interval. In order to limit the amount of real-time and routing packets a node is allowed to transmit, the size of the high and medium priority sub-queues has been restricted to 10 slots. A much longer queue for real-time and routing packets would result in more outdated packets. The queue size limit for low priority traffic is 80 packets. This is a trade-off between queueing as many best-effort packets as possible if the channel is currently used by high and medium priority traffic and the risk that the topology of the network has already changed and the next hop in the IP header of low priority traffic is not up-to-date any more which will result in expensive retransmissions.

As the packets are marked with different priorities they can be handled specifically by the routing protocol and the interface queue. RREQ messages are marked with medium priority by default. By doing this, the actual current queue length of intermediate nodes are indirectly taken into account during the route discovery process in AODV. Nodes that have fewer packets in their high and medium priority sub-queues will be able to forward their packets quicker than nodes that already have to forward multiple high priority data streams. However, to minimise the delay in the event of broken links or unavailable route for high priority data, an ad-

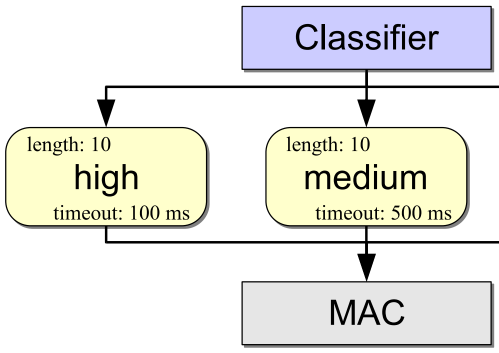

Figure 3.3: Design of the Priority Queue

ditional flag (bit 6 in the AODV RREQ header) has been introduced that indicates that a RREQ is of high priority to enable nodes speeding up the route discovery process. But without marking the RREQ the node that replies to the RREQ is not aware that the reply is urgent. Therefore, when the destination node receives a RREQ with a high priority flag, the TOS field of the corresponding IP packet that holds the RREP message is set to high priority to make sure that the RREP is queued in the high priority sub-queue and returned as quickly as possible.

In addition to the interface queue, AODV itself queues data packets if no route to the destination is currently available. Because the route discovery process generally takes much more than 150 ms, high priority packets are not queued by AODV but are dropped immediately. In order to cope with selfish nodes that simply mark all packets with high priority, packets can be filtered at the incoming interface to limit the amount of real-time packets from a certain node.

Before a packet is going to be transmitted either the application or the middleware marks the packet with one of the three priorities. Then the routing protocol makes its routing decision and passes the packet down to the *interface queue* (see Figure 3.1). The following steps need to be carried out when a packet is enqueued and dequeued, respectively.

Enqueue

The *classifier* delegates the packets coming from the routing protocol according to their priority flag to one of the sub-queues as long as enough queueing space is available. If the corresponding sub-queue is full, the queue is checked for outdated packets. If the queue is still full the packet is simply dropped. Otherwise, the packet is put into the particular sub-queue and marked with the current time-stamp.

Dequeue

Every time a packet is passed down to the MAC layer, the packets from the high priority queue are preferred and afterwards the medium priority queue. If both queues are empty the packets are taken from the low priority queue. If the next packet that is going to be dequeued from the queue is a low priority traffic the rate control system decides whether or not it processes the packet or refrains from transmitting the low priority packet in favour of neighbouring nodes that might want to send high priority traffic (refer to Section 3.3.3). If the rate controller accepts the packet, the current time is compared to the time-stamp the packet entered the system (refer to Section 3.3.2). If the time interval exceeds the time limit the packet is dropped and another packet is dequeued if available. If the packet has not timed-out yet, the packet is passed down to the MAC layer.

3.3.2 Timeouts

Real-time multiplayer games demand low latency connections with less than 150 ms round trip or 75 ms one-way delays, assuming symmetric latencies as discussed in Section 2.5. However, packets that are delayed slightly more than 75 ms, might still be useful to some prediction mechanisms such as dead-reckoning. Therefore, only packets which take more than 100 ms one-way and thus are clearly obsolete for real-time applications are dropped by the interface queue. Outdated real-time packets needlessly consume bandwidth and delay other real-time packets. Therefore, every packet that is stored in the queue is marked with a time-stamp. When the packet is dequeued the time the packet spent in the queue is compared with the queue policy. The maximum timeout for high priority traffic has been set to 100 ms. This timeout interval is further decreased depending on the number of hops the packet went and still has to go. This information is retrieved from the local routing table. For every hop the maximal timeout is reduced by 10 ms which approximates the time it takes to process and forward the packet in the optimal case. This is still a rather conservative approach and aims at dropping just the packets that will definitely come too late. The medium priority sub-queue has a timeout of 500 ms which has shown reasonable performance in the simulations. Since both priority sub-queues are rather small and the packets are outdated quickly compared to the low priority sub-queue, additional mechanisms that prevent for starvation of low priority traffic have not been added

Similar to the QoS for AODV protocol, discussed in Section 2.4.2, an IP-

Options header could have been used instead to measure the accumulated time of real-time packets at intermediate nodes. Although this approach is more concise, since it calculates the absolute time a packet spent at every node, this is not necessary as simulations in Section 5.2 and Figure 5.4 show. In general, packets are either processed and forwarded quickly or they are queued for considerably more than 100 ms seconds. Moreover, this requires additional signalling in every real-time packet which might be a huge overhead in particular if real-time packets are very small.

3.3.3 Real-Time Neighbour Aware Rate Control

The current IEEE 802.11 protocol standards do not support QoS at the MAC layer and the medium access is carried out by the Distributed Coordination Function (DCF) handling every node equally. All nodes have the same probability to gain access to the wireless channel and DCF does not distinguish between high priority and low priority data traffic. Furthermore, there is no guarantee that a node will send even high priority data packets within a certain time frame. Therefore, nodes that send high amounts of low priority data might consume most of the shared bandwidth. Due to the contention mechanism in DCF, other nodes which send high priority data might wait too long to access the channel and cannot transmit their high-priority data in time, although a priority queue has been used. The reason behind this is, that priority queueing works only locally and does not take neighbouring nodes into account. To solve this problem at the MAC layer is one of the aims of the new QoS IEEE 802.11e [3] protocol that is still under development and not yet available. Even with the advent of IEEE 802.11e the current standard will not be displaced immediately and thus, a mechanism is required that enables nodes that send real-time data to refrain other nodes from accessing the channel.

The proposal in this thesis to overcome the mentioned limitations of the IEEE 802.11 MAC layer, is to restrain the amount of low priority packets a node is allowed to transmit within a time interval depending on the amount of nodes transmitting real-time traffic in its neighbourhood. Therefore, the interface queue needs access to the actual number of neighbours sending high priority traffic to adapt the rate control system of low priority traffic.

For this purpose, every node maintains a time-stamp *PRIO_TIMEOUT* that is updated when nodes transmit a high-priority packet. If the node has sent real-time traffic within the last *PRIO_TIMEOUT* seconds the node marks broadcast routing messages with a real-time flag. Employing the unused bit 5 in the AODV header of RREQ and RREP messages, a node indicates whether it currently transmits high priority traffic or not. In addition to that, the routing table entry has been extended by a time-stamp *LAST_PRIO*. Every time a node receives high priority data or a routing message with the high priority flag, the *LAST_PRIO* time-stamp for that particular routing table entry is updated. Now, the actual amount of neighbours sending high priority traffic with an up-to-date *LAST_PRIO* time-stamp can be derived from the neighbour list and the interface queue can adapt the rate control of low priority traffic upon the number of neighbours sending real-time traffic. If the mobility rate of the nodes is low and real-time nodes have not sent routing messages for a while, the actual number of real-time neighbours might be higher, however. In addition, selfish nodes might just mark all routing packets with the priority bit.

3.4 MAC Layer Support

Many features, already implemented in the IEEE 802.11 MAC layer, remain unused in higher layers and are implemented again, however, less efficiently. With *broken link detection* based on *Link Layer Feedback* (LLF) the routing protocol is notified instantly if packets cannot be sent any longer over a certain link. In general, routing protocols periodically broadcast messages for *neighbour detection*. However, IEEE 802.11 ad hoc networks already broadcast periodic advertisements (BEACON frames) and thus the periodic routing messages are not required any more, neither for broken link detection nor for neighbour detection. With the help of *signal strength monitoring* more stable routes can be selected. In addition, IEEE 802.11 relies on RTS/CTS (Ready To Send/Clear To Send) to avoid the hidden terminal problem (refer to Section 2.2). *RTS/CTS adaptation* to the application's requirements is essential to not degrade the overall performance. In order to have accurate access to the current network state, such as an up-to-date neighbour list and the wireless signal strength for each neighbour, support from the IEEE 802.11 MAC layer is required.

3.4.1 Broken Link Detection

As mentioned in Section 2.3.5 AODV relies by default on HELLO messages for neighbour management and broken link detection. If, for example, a node does not receive three consecutive HELLO messages from a particular neighbour the node regards the link to that neighbour as broken. It might take several seconds until the node realises the link failure because HELLO messages are sent generally only every second. To improve the broken link detection system and react fast to link failures, the *Link Layer Feedback* (LLF) mechanism provided by NS-2 and the AODV-UU implementation has been employed. If a packet cannot be transmitted successfully, that is the sender does not receive an ACK from the destination within a certain time interval, the packet is retransmitted. Excessive retransmissions, however, can be reported by the MAC layer and can be propagated to the routing protocol to deal with the broken link. Figure 3.4 illustrates this mechanisms. Node A sends a data packet x to node B. However, it does not arrive at B and after a timeout node A retransmits packet x. After the third timeout node A gives up and triggers a broken link event to the routing protocol. This mechanism has two benefits. First, it reacts to broken links by usually one order of magnitude quicker than relying on HELLO messages and second, HELLO messages used for broken link detection are not required any more which reduces the routing overhead. But HELLO messages are still needed to maintain a neighbour list if data packets are not regularly transmitted to all neighbours. The following mechanism explains how this can be achieved even without HELLO messages.

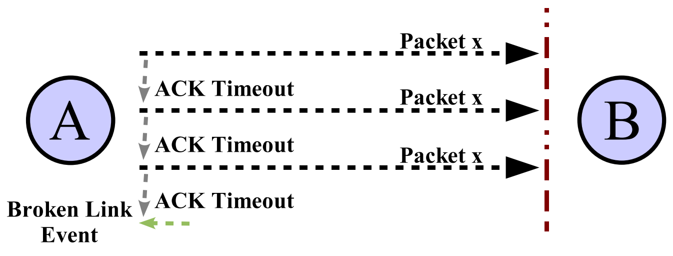

Figure 3.4: Broken Link Detection based on Link Layer Feedback

3.4.2 Neighbour Detection

To make routing decisions or management actions in the ad hoc environment every node requires an up-to-date neighbour list. In order to receive this list, usually each application implements its own neighbour discovery procedure based on network probes or HELLO messages. This results in message overhead and waste of the scarce bandwidth. Since this service is commonly demanded, it makes sense to provide a common interface to upper layers accessing the neighbour list from the MAC layer. Unfortunately, NS-2 does not provide a way to maintain a neighbour list based on management frames from the MAC layer since it employs a simplified

MAC layer that does not uses BEACON frames to manage the ad hoc network. Within the time constraints of this thesis the NS-2 MAC layer implementation has not been extended to provide BEACON support. Thus the effects of BEACON based neighbour detection have not been investigated in NS-2. However, a real environment implementation relies on BEACON frames that are sent out regularly every 100 ms by the IEEE 802.11 MAC layer if operating in ad hoc mode. An implementation of BEACON based neighbour detection has been provided for the real test environment running Linux (refer to Section 4.3.3).

3.4.3 Signal Strength Monitoring

To provide for link stability and avoid the grey zones mentioned in [17] (refer to Section 2.2.2) the *Signal-to-Noise Ratio* (SNR) is measured for routing messages. If the signal strength of a RREQ is not beyond a certain SNR threshold the request is not processed. Thus, other neighbours that have received the RREQ with a higher signal strength will forward the packet and create a more stable route. On the opposite, intermediate nodes that receive an RREQ that is above a certain SNR threshold do not forward the RREQ. In this case, it is assumed that these nodes are very close to the node that has originated the RREQ and thus, will needlessly broadcast the RREQ as well. Figure 3.5 illustrates both mechanisms. The red nodes *1*, *2* are very close to node *S* and receive the RREQ with a very high SNR value and thus do not forward the RREQ. Node *I* receives the routing message as well, but the SNR value is very low and the link might break soon or does not provide reliable transport. Therefore, node *I* does not forward this RREQ. Node *3* and *4*, however, receive the routing message with an SNR value that is in the given range and forward the RREQ. This time, node *I* receives a RREQ with a higher SNR value and forwards it to the destination node *D*. AODV employs *expanding ring search* based on the *Time To Live* (TTL) field in the IP header for dissemination of RREQ messages. If a node receives a RREQ which has been sent with the maximal TTL, it does not drop the RREQ even though the signal strength is below the threshold, since no better route is available. Due to the limitations of the employed two-ray ground propagation model in NS-2, signal strength monitoring does not provide realistic simulation results and has not been used in the evaluation. Within the time constraints of this thesis a different propagation model has not been implemented in NS-2. However, signal strength monitoring can be exploited in a real environment implementation (refer to Section 4.3.4).

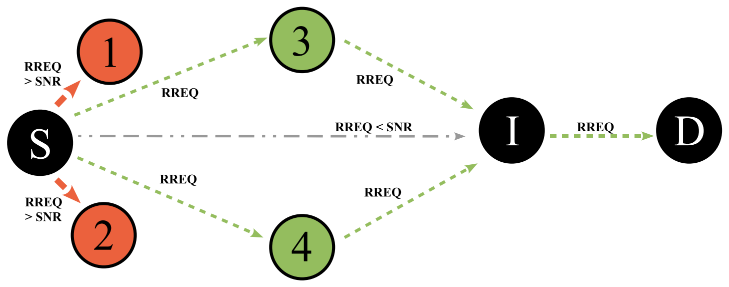

Figure 3.5: Signal Strength Dependent RREQ Processing

3.4.4 RTS/CTS Adaptation

NS-2 supports a certain packet size threshold to indicate if RTS/CTS (refer to Section 2.2.1) should be used. By default the threshold is 0 bytes and RTS/CTS is always enabled. Without RTS/CTS the transmission channel is shared equally among the contenting nodes and it requires less bandwidth and presumably reduces end-to-end delay and delay jitter.

3.5 Summary

In this chapter, several QoS extensions have been proposed to cope with typical challenges in MANETs: mobility, congestion, shared medium and the wireless signal. The extensions follow a cross-layer design between the routing, interface queue and MAC layer. In the following chapter, implementation details of some QoS extensions for the network simulator NS-2 [5] and if appropriate for Linux are given.

Chapter 4

Implementation

This chapter describes the implementation details of the QoS extensions in the network simulator NS-2 [5] and the Linux operating system. The main focus of the QoS extensions in NS-2 is on AODV enhancements like backup route and neighbour list provisioning, traffic management such as the priority queue, timeouts and rate control and the interaction of the priority queue with the routing protocol. QoS extensions such as local repair, link layer feedback, RTS/CTS adaption are covered only briefly since the employed routing protocol implementation AODV-UU [58] or the NS-2 MAC layer already support these features. Furthermore, a prototype implementation under Linux of selected QoS extensions such as priority queueing with timeouts, broken link detection, neighbour detection based on IEEE 802.11 BEACONs and signal strength monitoring are described. AODV-UU provides a concurrent implementation for NS-2 and Linux based on the same source code and thus has been employed in the real test environment, as well. A brief introduction to NS-2 and AODV-UU is given in Appendix A and B, respectively.

4.1 NS-2 Implementation

Since NS-2 has just one single thread of execution synchronisation issues that have to be dealt with in a real system like the Linux kernel do not occur in NS-2. Furthermore, cross-layer design of different layers in the protocol stack, for instance, the MAC layer, interface queue and routing protocol is much easier to implement in NS-2 as these modules are linked to a single binary that makes up the network simulator and does not distinguish between *user space* and *kernel space* as in Linux, for example.

The implementation of the QoS extensions, in particular the traffic management QoS extensions, follow a cross-layer design between the interface queue and the routing protocol. Existing NS-2 or AODV-UU source code that has been modified or extended is marked with own flags to allow for conditional compilation. Thus, the features that have been added in this thesis can be switched on and off using additional compilation flags listed in Table 4.2 in the `Makefile` of NS-2.

QoS Mechanism	**Compilation Flags**	**Dependencies**
Backup Route	BACKUP_ROUTE	none
Neighbour List	AODV_NEIGHBOUR	*optional* LLF_HELLO
Priority Queue	PRIORITY_QUEUEING	QOS_QUEUE
Timeouts	QUEUE_TIMEOUTS	PRIORITY_QUEUEING
Rate Control	RATE_CONTROL	AODV_NEIGHBOUR
Signal Monitoring	SIGNAL_STRENGTH	none

Table 4.1: Conditional Compilation Flags

4.1.1 AODV Enhancements

This section focuses on the implementation of the AODV enhancements such as backup route and neighbour list provisioning in NS-2. Since AODV-UU already supports local repair, it is not discussed here and Section 4.2.2 explains how the feature can be enabled or disabled.

Backup Route

In MANETs, packets can often be routed along several potential paths as discussed in Section 3.2.2. The AODV standard, however, just maintains a single path for a certain destination, although different routes might be known from other routing messages. To allow AODV-UU to store a backup route the routing table entry struct `rt_table` has been extended by the field `backup_hop` to store the next hop on the backup route. In addition to that, the processing of RREQ and RREP messages needs to be extended. Listing 4.1 depicts the necessary modifications to verify if the routing information from a certain RREQ message can be used as a backup route. At the beginning, the routing table entry for the reverse route of the RREQ is retrieved from the routing table. In order to use the information from the RREQ as a backup route a number of conditions have to be met. First, a reverse route must already exist and second, the sequence number of the RREQ has to be fresh. Third, the hop count of the RREQ has to be the same as the existing hop count. If a route with a lower hop count is available this route is used as the default route and not as a backup route. Fourth, the next hop of the existing route must be different to the one in the RREQ message. An installed backup route is removed if a new default route with a lower hop count or fresher sequence number becomes available. The code for processing RREP messages is analog. If a broken link has been detected, either

```
// ns/aodv-uu/aodv_rreq.c
// void NS_CLASS rreq_process(RREP * rreq, [..])
#ifdef BACKUP_ROUTE
    // retrieve the routing table entry for the reverse route
    rev_rt = rt_table_find(rreq_orig);
    // found a backup route with the same hop count
    // and up-to-date destination number
    if (rev_rt &&  rev_rt->dest_seqno <= rreq_dest_seqno &&
            rreq_new_hcnt == rev_rt->hcnt &&
            rev_rt->next_hop.s_addr != ip_src.s_addr ) {
        if (rev_rt->backup_hop.s_addr == 0)
            // different strategy possible
            rev_rt->backup_hop = ip_src;
    }
#endif
```

Listing 4.1: Installing a Backup Route

by relying on link layer feedback or employing HELLO messages, an available backup route can be used as shown in Listing 4.2. The source code deals with two cases. Both have in common that a packet from the interface queue could not be transmitted successfully. In the first case, a route has changed but still packets might exist in the interface queue that are marked with the old route that does not exist any longer. Thus, the packet cannot be transmitted successfully and the error handler is called. Instead of dropping the packet it is passed to the interface queue again using the latest information from the routing table. In the second case, the next hop of the packet and the next hop according to the routing table are the same. Thus, the current routing table entry is broken, as well. If a backup route exists, the next hop is taken from the backup route and the packet is retransmitted.

Neighbour List

By default, AODV does not use HELLO messages any longer if the link layer based broken link detection mechanism is employed. One disadvantage is that without a neighbour detection mechanism like HELLO messages neighbours are only de-

```
// ns/aodv-uu/ns/aodv-uu.cc
// void NS_CLASS packetFailed(Packet *p)
#ifdef BACKUP_ROUTE
// packet has been sent over an old next hop
if (rt->next_hop.s_addr != next_hop.s_addr) {
    // next hop mismatch - changing next hop, backup route
    sendPacket(p, rt->next_hop, 0.0);
    goto end;}
// packet has been sent over the current next hop
else if (rt->next_hop.s_addr == next_hop.s_addr &&
        rt->backup_hop.s_addr != 0  &&
        rt->next_hop.s_addr != rt->backup_hop.s_addr)  {
    // next hop is broken use the backup
    rt->next_hop = rt->backup_hop;
    sendPacket(p, rt->next_hop, 0.0);
    goto end;}
#else
```

Listing 4.2: Applying the Backup Route

tected by AODV control messages, such as RREQ, RREP and RERR. Thus, the number of neighbours in the vicinity of a certain node can be higher if some nodes have not sent or forwarded a RREQ for a while. To avoid extra signalling for neighbour detection IEEE 802.11 BEACON based neighbour detection has been proposed in Section 3.4.2. However, since the employed NS-2 MAC layer does not support BEACON frames in ad hoc mode according to the IEEE 802.11 standard, the NS-2 implementation of AODV-UU has been extended to employ HELLO messages for neighbour management although link layer feedback for broken link detection is enabled to obtain a more accurate neighbour list. This feature can be enabled and disabled by the conditional compilation flag `LLF_HELLO`.

In order to provide a neighbour list to other modules, for instance, the interface queue (refer to Section 4.1.2) and the application layer, every module interested in the neighbour list can register itself at the routing protocol. Whenever the neighbour list is modified all registered modules are notified by the routing protocol following the observer design pattern [59]. The neighbour list `neigbors` and the callback list `nbListenerList` are defined in `ns/aodv-uu/aodv_neighbor_callback.h` and declared in `ns/aodv-uu/aodv_neighbor.h`. AODV-UU's generic list management C macros (file `ns/aodv-uu/list.c`) are taken from the Linux kernel to allow the same list manipulation functions to be used with different types of lists. In addition to the list declaration several list management functions have been added, as shown in Listing 4.3. Neighbours can be added, removed and searched for by specifying a particular routing table entry as an index. Furthermore, the total amount and the amount of priority neighbours can be retrieved, for example, when a callback function is executed.

```
// ns/aodv-uu/aodv_neighbor.h
void addNeighbor(rt_table_t *rt);
int removeNeighbor(rt_table_t *rt);
neighbor_t* findNeighbor(rt_table_t *rt);

unsigned int getNumNeighbors();
unsigned int getNumPrioNeighbors(unsigned int timeout);
// retrieves the neighbour list, nb is output parameter
unsigned int getNeighborList(list_t * nb);

// callback is a function pointer
void addNeighborListener(NeighborCallback callback, void * data);
```

Listing 4.3: Neighbour Management Functions

In order to determine whether a neighbour sends real-time traffic and thus is a high priority neighbour the routing table entry structure needs to be extended by the field `last_prio` to indicate the last known reception time of a high priority packet from a certain neighbour. If a node receives real-time traffic from a particular neighbour the time-stamp `last_prio` of the corresponding routing table entry is updated to mark the node as high priority. Other neighbours that do not receive real-time traffic from this node will not consider the neighbour as high priority since with this mechanism only neighbouring nodes that send real-time traffic directly to the concerning node are taken into account. If a neighbour sends real-time traffic to a different node it is still a high priority neighbour but the mentioned mechanism will not account for that. Therefore, routing control messages in particular RREQ messages that are broadcast to all neighbours are marked with a flag if the node sends high priority traffic. To accomplish this every node maintains a time-stamp `prio_time` that indicates when it has sent the last high priority packet. Depending on this time-stamp, bit five of the AODV header is set in all routing messages if the current node has sent high priority traffic within three seconds (`ACTIVE_PRIO_TIMEOUT`). Since RREQ messages are broadcast, all neighbours get to know if the node sends high priority traffic as soon as the corresponding bit has been set. The number of high priority neighbours can be determined by iterating the neighbour list and comparing the `last_prio` time-stamp with the current time. If the time difference is smaller than `ACTIVE_PRIO_TIMEOUT` the node is regarded as a high priority neighbour. The corresponding code can be found in the function `getNumPrioNeighbors` in file `ns/aodv/aodv_neighbor.c`.

4.1.2 Traffic Management

This section describes the implementation of the priority queue extended by timeouts and rate control of low priority traffic. Furthermore, the necessary integration with the routing protocol is explained.

Priority Queue

Figure 4.1 shows the class diagram of involved classes of the new priority interface queue using the *Unified Modeling Language* (UML). In NS-2 every interface queue has to be derived from class `Queue`. To be as compatible as possible to the original NS-2 queue implementation and still allow for a flexible solution the class `QoSQueueAdapter` is a sub-class of `Queue` and hides all NS-2 peculiarities and can employ own queue implementations that are of type `QoSQueueInterface`. Furthermore, the class `QoSQueueAdapter` holds a reference to the `AODVUU` routing agent to be able to access the routing table from the interface queue to retrieve the number of priority neighbours used in the rate control mechanism and to provide hop dependent queueing timeouts. Since the used interface between `AODVUU` and `QoSQueueAdapter` is very small the code can be easily ported to other routing protocols. The class `AODVUU` had to be modified as well to create the bidirectional association as shown in the UML class diagram. Listing 4.4 is part of the initialisation code of the routing protocol `AODVUU` and shows how the bidirectional association is created. If `QOS_QUEUE` is defined the object `obj` that comes from the `tcl` simulation file is casted into `QoSQueueAdapter`. By calling the member function `setRoutingAgent(this)` on the `queue` object the `QoSQueueAdapter` obtains a reference to the routing agent. Finally, the reference is stored in the member variable `ifqueue` that is of type `Queue` to be compatible with other interface queues, as well. This has been also changed because the original version of AODV-UU relies on the NS-2 class `PriQueue`.

The class `QoSQueue` is a simple *First In First Out* (FIFO) queue and the class `QoSPriorityQueue` models the priority queue that has been described in Section 3.3.1. Both queues are sub-classes of `QoSQueueInterface` as depicted in Fig-

```
// ns/aodv-uu/ns/aodv-uu.cc
#ifdef QOS_QUEUE
    QoSQueueAdapter* queue = (QoSQueueAdapter *) obj;
    queue->setRoutingAgent(this);
    ifqueue = queue;
#else
    ifqueue = (Queue *)obj;
#endif
```

Listing 4.4: Association between AODV-UU and the QoSQueue

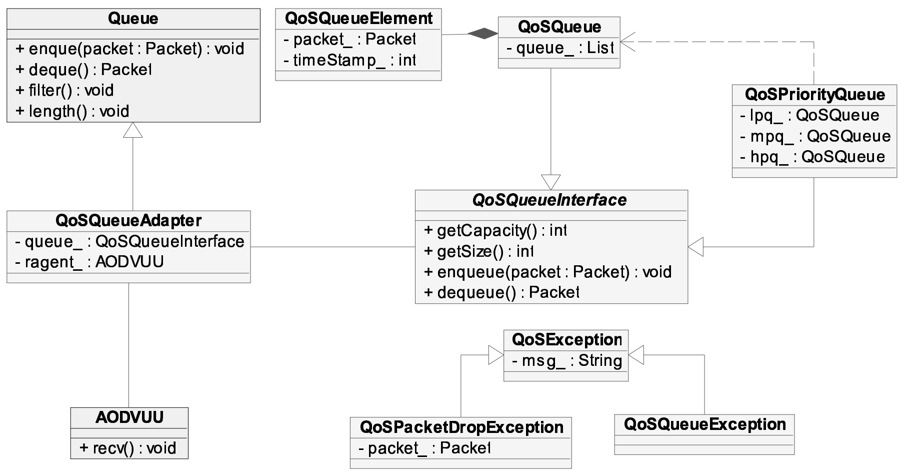

Figure 4.1: UML Diagram of the QoS Queue

ure 4.1. Thus, instances of these classes can be employed by the `QoSQueueAdapter`. The `QoSPriorityQueue` is made up of three sub-queues of type `QosQueue` to model the low, medium and high priority queues (refer to Section 3.3.1). It does not provide its own queue implementation and thus delegates all function calls to one of its three sub-queues. That is, the main queue implementation is in the class `QoSQueue` that stores packets wrapped in `QoSQueueElement` in its internal list. During the queue operations `enqueue` or `dequeue` exceptions might occur. If the queue is empty and the function `dequeue` is called a `QoSQueueException` is thrown, for instance. The same happens if the queue is full and the queue attempts to put another element into the queue. In addition to that, a `QoSPacketDropException` is thrown if a packet is dequeued but the packet has already been timed-out. All exceptions are caught and handled by the class `QoSQueueAdapter`.

In order to use the QoSQueue or the QoSPriorityQueue in simulations the type of the queue and the length of the queue need to be specified, as illustrated in Listing 4.5. If the priority queue is selected the timeouts and queue lengths of the QoSQueue sub-queues are defined in the constructor of the class QoSPriority-Queue.

```
# ns/tcl/lib/ns-defaults.tcl
# 1 FIFO QoSQueue , 2 Priority QoSPriorityQueue
Queue/QoSQueue set queueType_ 2
Queue/QoSQueue set queueLimit_ 100
```

Listing 4.5: QoSQueue Parameters

The tcl file that describes the NS-2 application traffic pattern has to mark the packets with priorities using the prio_ field, as shown in Listing 4.6.

```
# udp is an UDP agent that sends UDP traffic
# 0 low priority , 1 medium priority , 2 high priority
$udp set prio_ 2
```

Listing 4.6: UDP Agent sends Priority Traffic

Queue Timeouts

The code that is specific for queueing timeouts resides in the files qosqueue/-qosqueue.h and qosqueue/qosqueue.cc. If a packet is going to be enqueued the QoSQueueAdapter delegates the function call to its private queue that is either of type QoSQueue if priority queueing is disabled or of type QoSPriorityQueue. In the latter case the function call is delegated to the corresponding QoSQueue sub-queue. Listing 4.7 shows the according code of the method enqueue of class QoS-Queue. At the beginning the current simulation time is retrieved from the simulator using the QoSQueueAdapter which itself accesses the simulator's Scheduler. If no space is available in the queue the method dropOldPackets is called to delete packets that are stored in the queue but are already out of date. If some packets have been deleted the current queue size is smaller than the queue capacity and a new QoSQueueElement that holds the packet p and the current time-stamp is put either at the front or at the back of the queue depending on the flag front. Urgent RREP packets that have been sent as high priority are queued at the front of the high priority sub-queue. If no space is left in the queue and the packet could not be added a `QoSQueueException` is thrown that has to be handled by the caller, in this case the `QoSQueueAdapter`.

```
// ns/qosqueue/qosqueue.cc
void QoSQueue::enqueue(const Packet &p, bool front) {
Time now = QoSQueueAdapter::getCurrentTime();
#ifdef QUEUE_TIMEOUTS
if(getSize() >= getCapacity()) {
    dropOldPackets();
}
#endif
if(getSize() < getCapacity()) {
    if (front)
        queue_.push_front(new QoSQueueElement(p,now));
    else
        queue_.push_back(new QoSQueueElement(p,now));
}
else throw QoSQueueException("Queue is full");}
```

Listing 4.7: Method QoSQueue::enqueue

```
// ns/qosqueue/qosqueue.cc
const Packet& QoSQueue::dequeue() {
if (getSize() > 0) {
    QoSQueueElement *qqe =  queue_.front();
    queue_.pop_front();
    Time now = QoSQueueAdapter::getCurrentTime();
    const Packet &p = qqe->getPacket();
#ifdef QUEUE_TIMEOUTS
    if (isOldPacket(qqe)) {
        delete qqe;
        throw QoSPacketDropException("Packet has timed-out", p);
    } // no else
#endif
    delete qqe;
    return p;
}
else throw QoSQueueException("Queue is empty");}
```

Listing 4.8: Method QoSQueue::dequeue

Every time a packet is taken from one of the sub-queues the steps according to Listing 4.8 are carried out. If the queue is empty a `QoSQueueException` is thrown. Otherwise, the front element of the queue is taken from the queue and the method `isOldPacket` verifies whether the packet is outdated and if not the packet is returned. If the packet has already been stored in the queue for too long a `QoSPacketDropException` is thrown that is caught by the `QoSQueueAdapter` and triggers another execution of the method `dequeue`. The member function `isOldPacket` compares the current time with the time-stamp of the packet. Additionally, it subtracts a hop penalty of 10 ms (`HOP_PENALTY`) for every hop the packet already went or still has to be forwarded from the maximum queueing timeout. The idea is to account for the queueing delays of real-time packets that are transmitted over several hops. Listing 4.9 depicts how the timeout penalty is calculated. The `adapter_` object in the mentioned code is of type `QoSQueueAdapter` and determines the number of hops the packet already went and the remaining hops to the destination from the routing agent. For every hop the timeout hop penalty is increased by `HOP_PENALTY` as defined in `qosdefs.h`. The following example explains the calculation of the hop dependent queueing timeout of a high priority packet that is sent over a route that takes four hops. The maximum queueing timeout of a high priority packet at every node is 100 ms. Since the high priority packet requires four hops to reach the destination the queueing timeout is reduced by the hop penalty of 4*10 ms. Thus, the packet is only allowed to be queued for 60 ms at every node. In the worst case the packet spends 60 ms in every interface queue and it reaches the destination too late after more than 240 ms. However, if the packet spends 60 ms in one and 5 ms in the other three interface queues the packet can still arrive in-time with a latency of 75 ms.

```
// ns/qosqueue/qosqueue.cc
float QoSQueue::getHopPenalty(const Packet &p) {
    struct in_addr dest = QoSQueueAdapter::getDest(p);
    struct in_addr src = QoSQueueAdapter::getSrc(p);
    int hops_dest = adapter_->getRemainingHops(dest);
    int hops_src  = adapter_->getRemainingHops(src);
    int hops = hops_dest + hops_src;
    float penalty = (float)(hops*HOP_PENALTY)/1000;
    return penalty;}
```

Listing 4.9: Calculation of the Hop Penalty

Number of Real-time Neighbours	Reduction of Low Priority Traffic
0	10 %
1	14 %
2	20 %
>=3	33 %

Table 4.2: Rate Control Parameters

```
// ns/qosqueue/qospriorityqueue.cc
// const Packet& QoSPriorityQueue::dequeue()
// take packet from low priority sub-queue
if (lpq_.getSize() > 0 ) {
#ifdef RATE_CONTROL
    if(rand() % getRateControl()  == 0)
        throw QoSQueueException("Rate Control
                of Low Priority Traffic");
#endif } [..]
#ifdef RATE_CONTROL
unsigned int QoSPriorityQueue::getRateControl() {
    unsigned int rate = 10;
    if(numPrioNeighbors_ >= 1) {
        rate = 7;
    if (numPrioNeighbors_ >= 2)
        rate = 5;
    if (numPrioNeighbors_ >= 3)
        rate = 3;
    }  return rate; }
#endif
```

Listing 4.10: Rate Control of Low Priority Traffic

Rate Control

The rate control system has been implemented in class `QoSPriorityQueue` and is shown in Listing 4.10. As mentioned in Section 3.3.3, rate control of low priority traffic adapts to the amount of neighbouring nodes that transmit high priority traffic. To accomplish this, the routing protocol monitors high priority neighbours as described in Section 4.1.1 and the interface queue registers at the routing agent during initialisation to be notified if the neighbour list has been updated.

```
ragent_->addNeighborListener(&neighborCallback, this);
```

Depending on the number of neighbours that transmit high priority traffic the rate controller reduces the amount of low priority traffic a node is allowed to send. Table 4.2 depicts the relation between real-time neighbours and the reduction of low priority traffic, for instance, if a node has two high priority neighbours the maximum low priority traffic transmitted by this node is reduced by 20 %. In initial simulations these values delivered a reasonable performance. However, further simulations have to be carried out to find the optimal set up. The corresponding source code is illustrated in Listing 4.10. If a low priority packet is dequeued from the priority queue, a uniformly distributed random number is generated. If this random number modulo the according rate control value from Table 4.2 equals zero a `QoSQueueException` is thrown that is handled by `QoSQueueAdapter`. The adapter returns `NULL` to the MAC layer that will not use the channel this time and allows other nodes that are in its vicinity to send their high or medium priority traffic.

```
// ns/aodv-uu/ns/packet_input.cc
// void NS_CLASS processPacket(Packet * p)
// no route to the destination available
// ih : IP header of packet p
#ifdef PRIORITY_QUEUEING
        if (ih->prio() == PRIORITY_HIGH)
            drop(p);
        else
            packet_queue_add(p, dest_addr);
#else
        packet_queue_add(p, dest_addr);
#endif [..]
// start route discovery and set high priority flag if necessary
#ifdef PRIORITY_QUEUEING
          if (ih->prio() == PRIORITY_HIGH) {
                rreq_flags |= PRIO_ROUTING_FLAG;
          }
#endif
```

Listing 4.11: Service Differentiation of Data Packets in AODV-UU

Traffic Management in AODV

The parts in AODV that have been modified to account for different traffic types are marked with `PRIORITY_QUEUEING`. The function `processPacket` deals with routing and forwarding of data packets. By default, AODV queues packets that are waiting for a new route becoming available. However, high priority packets are quickly out-dated and hence are not queued and can be dropped instead. This is shown in the first part of Listing 4.11. In the second part, the high priority flag PRIO_ROUTING_FLAG is set in the AODV header if the packet that is waiting for a new route is a real-time packet. This allows the node that replies to the RREQ to set the *Type of Service* (TOS) field of the RREP to high priority and thus can be privileged by the interface queue as mentioned in Section 3.3.1. In addition

```
// ns/aodv-uu/aodv_socket.c
// void NS_CLASS aodv_socket_send(AODV_msg * aodv_msg, [..])
#ifdef PRIORITY_QUEUEING
// AODV route request packet
if (aodv_msg->type == AODV_RREQ)
        ih->prio() = PRIORITY_MEDIUM;
// AODV route reply packet
else if (aodv_msg->type == AODV_RREP ) {
    RREP* rrep = (RREP*) aodv_msg;
    if (rrep->h)
        ih->prio() = PRIORITY_HIGH;
    else
        ih->prio() = PRIORITY_MEDIUM;
}
// AODV route error packet
else if ( aodv_msg->type == AODV_RERR )
    ih->prio() = PRIORITY_MEDIUM;
#endif
```

Listing 4.12: Service Differentiation of Routing Packets in AODV-UU

to data packets, also routing packets are marked with different priorities, as depicted in Listing 4.12. Function aodv_socket_send is used by AODV to transmit any type of control routing messages. RREQ and RERR messages are always set to PRIORITY_MEDIUM while RREP messages are set to PRIORITY_HIGH if the corresponding priority flag (bit six in the header of the RREQ) has been set.

4.1.3 MAC Layer Support

Support from the MAC layer is mandatory to perform efficient routing with low signalling overhead and provide QoS, as discussed in Section 3.4. The employed NS-2 MAC layer does not fully support the IEEE 802.11 IBSS standard and does not provide BEACON messages for network management. Hence, neighbour detection based on IEEE 802.11 BEACON messages has not been implemented in NS-2. However, Section 4.3.3 shows how BEACON messages can be captured by a Linux device driver and employed for neighbour management in a real environment. Broken link detection based on link layer feedback is already implemented in the NS-2 MAC layer and in AODV-UU. Section 4.2.1 illustrates how the link layer feedback mechanism can be used by routing protocols. The following section explains which parts of NS-2 and AODV-UU have been modified to provide basic signal strength monitoring support.

Signal Strength Monitoring

The two-ray ground propagation model [60] used by NS-2 does not support the actual *Signal to Noise Ratio* (SNR) value of a captured packet. Instead, the signal strength only depends on the distance of two nodes and has always the same value if the distance between two nodes does not change. To get more realistic values a different propagation model that provides more accurate SNR values needs to be developed for NS-2. Due to the time constraints of this thesis a new propagation model has not been developed. In the following it is assumed that the used two-ray ground propagation model in NS-2 already provides accurate SNR values. The source code shows which parts have to be modified to enable AODV-UU to make routing decisions upon the signal strength of a received packet, for instance, to cope with grey zones (refer to Section 2.2.2). First, the packet structure `struct hdr_cmn` has been extended by the field `rxSignal_` to hold the signal strength that is retrieved from the NS-2 MAC layer, as shown in Listing 4.13. With the currently employed two-ray ground propagation model the signal strength is not accurate. If

```
// ns/mac/wireless-phy.cc
// int WirelessPhy::sendUp(Packet *p)
Pr = propagation_->Pr(&p->txinfo_, &s, this);
#ifdef SIGNAL_STRENGTH
    hdr_cmn *hdr = HDR_CMN(p);
    hdr->rxSignal() = Pr;
#endif
```

Listing 4.13: Signal Strength Monitoring in NS-2

AODV receives a RREQ message the function `rreq_process` is executed. Within this function the signal strength is compared against two threshold values, as illustrated in Listing 4.14 and Listing 4.15, respectively. If the signal strength is below the threshold RREQ_SIGNAL_THRESH the RREQ is ignored since the route might not be very stable. However, if the *time to live* (TTL) field has been set to the maximum value NET_DIAMETER the route request is not discarded. In that case, the node assumes that no better route exists and continues processing the RREQ. On the other

```
// ns/aodv-uu/aodv_rreq.c
// void NS_CLASS rreq_process(RREQ * rreq, [..])
#ifdef SIGNAL_STRENGTH
// Discard RREQ due to low signal strength
if (rreq_new_hcnt + ip_ttl < NET_DIAMETER &&
        rxSignal < RREQ_SIGNAL_THRESH)
    return;
#endif
```

Listing 4.14: Discarding RREQs with Weak Signal Strength

hand, the signal strength of the received packet might be very high if the recipient node is very close to the sender. As mentioned in Section 3.4.3, the recipient needlessly forwards the RREQ as this node does not cover a significantly larger area. Hence, the node ignores the RREQ if the signal strength is higher than a certain threshold RREQ_SIGNAL_MAX_THRESH. In order to obtain meaningful simulation re-

```
// ns/aodv-uu/aodv_rreq.c
// void NS_CLASS rreq_process(RREQ * rreq, [..])
#ifdef SIGNAL_STRENGTH
// Do not forward RREQ due to high signal strength
if (rreq_new_hcnt + ip_ttl < NET_DIAMETER &&
        rxSignal > RREQ_SIGNAL_MAX_THRESH) {
    return;
#endif
```

Listing 4.15: Discarding RREQs from Close Neighbours

sults a different, more probabilistic propagation model has to be employed in NS-2. Then, the introduced code can be reused to allow for signal strength aware routing in AODV-UU.

4.2 Available QoS Extensions in NS-2

Some of the employed QoS extensions have already been implemented either in AODV-UU or NS-2. This section describes where the particular sections can be found in the source code and explains how the given features can be enabled or disabled.

4.2.1 Broken Link Detection

By default, AODV-UU employs HELLO messages to detect broken links. However, this approach requires additional signalling and does not respond quickly enough to link changes as the simulation results in Section 5.4.1 show. The solution to that is to rely on the information from the MAC layer as described in Section 3.4.1. To accomplish this, the protocol that is interested in feedback from the link layer registers a callback function and the MAC layer calls this function if the packet could not be delivered. Right before passing the packet to the interface queue AODV-UU registers the callback `link_layer_callback`, as illustrated in Listing 4.16. If a broken link due to excessive retransmissions has been detected

```
// ns/aodv-uu/ns/aodv-uu.cc
if (llfeedback) {
    ch->xmit_failure_ = link_layer_callback;
    ch->xmit_failure_data_ = (void *) this;}
```

Listing 4.16: Link Layer Feedback in AODV-UU

by the MAC layer the callback function executes `packetFailed` to deal with the broken link. The mechanism can be globally activated in the configuration file `ns/tcl/lib/ns-defaults.tcl`, as shown in Listing 4.17.

```
# ns/tcl/lib/ns-defaults.tcl
# 1 - enable ; 0 - disable
Agent/AODVUU set llfeedback_ 1
```

Listing 4.17: Enabling / Disabling Link Layer Feedback

4.2.2 Local Repair

If AODV cannot deliver a packet due to a broken link it can either decide to repair it locally or to notify the source of the packet by sending a RERR message. Listing 4.18 depicts the necessary steps. First, the packet that could not be delivered is queued in the routing queue. In addition, all packets from the interface queue that are supposed to be sent over the same next hop are removed and put into the routing queue, as well. Next, the route is marked to be repaired and neighbour management functions are called. Finally, the local repair process and a new route discovery is started. AODV's local repair mechanism can be enabled in the configuration file `ns/tcl/lib/ns-defaults`, as shown in Listing 4.19.

```
// ns/aodv-uu/ns/aodv-uu.cc
// void NS_CLASS packetFailed(Packet *p)
packet_queue_add(p, dest_addr);
interfaceQueue((nsaddr_t) next_hop.s_addr, IFQ_BUFFER);
rt_next_hop->flags |= RT_REPAIR;
neighbor_link_break(rt_next_hop);
rreq_local_repair(rt, src_addr, NULL);
```

Listing 4.18: Local Repair Mechanism

```
# ns/tcl/lib/ns-defaults.tcl
# 1 - enable ; 0 - disable
Agent/AODVUU set local_repair_ 1
```

Listing 4.19: Enabling / Disabling Local Repair

4.2.3 RTS/CTS Adaptation

NS-2 supports a packet size threshold for the use of RTS/CTS. If the packet is bigger than the defined threshold, RTS/CTS is applied otherwise it is disabled. By default the threshold is 0 and thus RTS/CTS is always applied. This threshold can be defined in the main NS-2 simulation `tcl` file, as illustrated in Listing 4.20. In this case, RTS/CTS is only activated for packets that are bigger than 1000 bytes.

```
# simulation.tcl
Mac/802_11 set RTSThreshold_  1000
```

Listing 4.20: Setting RTS/CTS Threshold

4.3 QoS Extensions under Linux

This section describes the implementation of selected QoS extensions in a real test environment using Linux. The main focus is on QoS support by the MAC layer and priority queueing using timeouts. DLink DWL-G650 that is IEEE 802.11/b/g compatible has been employed as the wireless network interface. Since some extensions require the modification of the device driver of the employed network card these extensions are not compatible with other network cards that depend on a different chip-set. The DLink card employs an Atheros chip-set and the corresponding Linux *Multiband Atheros Driver for WiFi* (MADWIFI) [61] driver is well designed and supports all the required features to implement the MAC layer QoS extensions mentioned in Section 3.4. Previously, the Lucent Orinoco Gold and Cisco Aironet cards have been tested, as well. However, these cards cannot be used to provide neighbour detection based on IEEE 802.11 BEACON messages, as recommended in Section 4.3.3.

4.3.1 Priority Queue

Linux supports several different built-in interface queues called *queueing disciplines* or *qdisc*, such as *First In First Out* (FIFO), *Priority FIFO* (PFIFO), *PRIO*, *Random Early Detection* (RED), *Class Based Queueing* (CBQ), as explained in [62]. A qdisc can be either *class-less* or *class-based*. Class-less qdiscs like FIFO and PFIFO do not allow to employ other qdiscs as sub-queues. Class-based qdiscs like PRIO and CBQ might have further sub-queues, for instance, a PRIO qdisc might consist of three class-less FIFO sub-queues. Each network interface has one egress root queueing discipline called *root qdisc* which forms a tree of qdiscs with class-based qdiscs as inner qdiscs and class-less qdiscs as leaves. By default the root qdisc is the class-less *PFIFO_FAST* queueing discipline that supports priority queueing and consists of three built-in sub-queues called *bands* which are actual FIFO queues. Based on the *Type of Service* (TOS) flag in the IP header the packets are queued into one of the bands. If the high priority band is empty packets are taken from the medium and then from the low priority band. Thus, the basic architecture is very similar to the queueing system proposed in Section 3.3.1. Unfortunately, the *PFIFO_FAST* queueing discipline and all other existing qdiscs do not support the queueing timeout mechanism that has been shown to be very powerful in the simulations (refer to Section 5.3.2). The queueing discipline *PRIO* is similar to *PFIFO_FAST*, however, since it is class-based other queueing disciplines can be applied as sub-queues. The idea is to employ the PRIO queueing discipline as the egress root qdisc and instead of the default FIFO sub-queues, three class-less qdiscs called *QoSQueue* derived from the FIFO qdisc and extended with timeouts are used.

In order to modify the qdisc or install a different queueing discipline for a particular network interface at run-time the command `tc` (traffic control) can be used. `tc -s qdisc` depicts the current interface queue configuration for all known network interfaces. Hence, if a new qdisc implementation is added to the Linux kernel `tc` has to be extended and recompiled, as well, to be able to configure the new qdisc. The next section describes the Linux kernel module `qosqueue.ko` that provides the qdisc QoSQueue. Afterwards the modifications to `tc` are presented.

QoSQueue Linux Kernel Module

The QoSQueue qdisc has been implemented as a Linux kernel module and is based on the Linux kernel FIFO queue in file `linux/net/sched/sch_fifo.c`. The definition of the data structure that describes the configuration parameters of the queueing discipline are stored in the Linux kernel header file `pkt_shed.h`, as illustrated in Listing 4.21. In the following, important functions of the QoSQueue qdisc are

```
// include/linux/pkt\_sched.h
struct tc_qosqueue_qopt {
  __u32 limit;     /* Length of the queue */
  __u32 timeout;   /* Timeout in ms */
};
```

Listing 4.21: QoSQueue Options

explained. Listing 4.22 illustrates the necessary code to enqueue a packet. If the queue is full all out-dated packets are removed from the queue to make space for up-to-date packets. Before the packet `skb` is added to the list of queued packets it is marked with the current time-stamp `skb->stamp` by calling `do_gettimeofday`. Every time a packet is removed from the queue the time-stamp is compared with the current time. If the packet has not been out-dated the packet is returned, as shown in Listing 4.23. A pointer to the top element of the queue is stored in the socket buffer `skb`. If the packet is not `NULL` the current time is compared with the time the packet had been enqueued. If the packet is still valid according to the queue timeout policy `q->timeout` the packet is returned. Otherwise, the packet is dropped and another packet is dequeued from the queue. The traffic control tool

```
// linux/qosqueue/qosqueue.c
static int qosqueue_enqueue(struct sk_buff *skb, struct Qdisc* sch) {
    struct qosqueue_sched_data *q = qdisc_priv(sch);
    struct sk_buff *skb_q;                          // skb in the queue
    if (sch->q.qlen >= q->limit) {
        skb_q = qosqueue_dequeue(sch);
        if(skb_q)
            qosqueue_requeue(skb_q, sch);
    }
    // this is not an else branch
    if (sch->q.qlen < q->limit) {
        do_gettimeofday(&skb->stamp);
        __skb_queue_tail(&sch->q, skb);
       [..]}}
```

Listing 4.22: Linux QoSQueue Enqueue Function

```
// linux/qosqueue/qosqueue.c
static struct sk_buff * qosqueue_dequeue(struct Qdisc* sch){
    struct timeval now;
    struct qosqueue_sched_data *q = qdisc_priv(sch);
    struct sk_buff *skb;
    while ((skb = __skb_dequeue(&sch->q)) != NULL ) {
        do_gettimeofday(&now);
        __u64 diff = timeval_diff(&now,&skb->stamp);
        if (diff > q->timeout*1000) {     // usecs
            sch->qstats.drops++;
            kfree_skb(skb);
        } else
            return skb;
    }
    return skb;
```

Listing 4.23: Linux QoSQueue Dequeue Function

`tc` employs the *Netlink* [63] interface to allow user space programs to interact with the Linux kernel. In order to configure the QoSQueue and set parameters like the queue length and the maximum timeout the function `qosqueue_init` is called (Listing 4.24). If no options have been applied default values are used. Otherwise the queue size and the timeout are stored in `q->limit` and `q->timeout`, respectively. In order to retrieve the current qdisc configuration the `tc` employs the `qosqueue_dump` function. Both queue size and timeout are stored in `opt` and sent via Netlink

```c
// linux/qosqueue/qosqueue.c
static int qosqueue_init(struct Qdisc *sch, struct rtattr *opt) {
    struct qosqueue_sched_data *q = qdisc_priv(sch);
    if (opt == NULL) {
        q->limit = sch->dev->tx_queue_len ? : 1;
        q->timeout = 10000;
    } else {
        struct tc_qosqueue_qopt *ctl = RTA_DATA(opt);
        if (opt->rta_len < RTA_LENGTH(sizeof(*ctl)))
            return -EINVAL;
        q->limit = ctl->limit;
        q->timeout = ctl->timeout;
    }
    return 0;}
```

Listing 4.24: Linux QoSQueue Init Function

to the `tc` tool, as illustrated in Listing 4.25.

```c
// linux/qosqueue/qosqueue.c
static int qosqueue_dump(struct Qdisc *sch, struct sk_buff *skb) {
    struct qosqueue_sched_data *q = qdisc_priv(sch);
    unsigned char    *b = skb->tail;
    struct tc_qosqueue_qopt opt;
    opt.limit = q->limit;
    opt.timeout = q->timeout;
    RTA_PUT(skb, TCA_OPTIONS, sizeof(opt), &opt);
    return skb->len;
rtattr_failure:
    skb_trim(skb, b - skb->data);
    return -1;}
```

Listing 4.25: Linux QoSQueue Read Configuration Function

Traffic Control Extension

The `tc` traffic control tool as part of the `iproute2-2.6.9` Debian Linux source packet has been extended with support for the QoSQueue queueing discipline. The file `q_qosqueue.c` in `iproute/tc/` sub-directory has been added and describes how `tc` interacts with the Linux kernel module `qosqueue.ko`. In addition to that, the `iproute/tc/Makefile` has been modified to compile the new file as well. Furthermore, a local version of `include/linux/pkt_sched.h` that is part of the iproute source code has been extended as described in Listing 4.21. The additional code to manage the QoSQueue is in file q_qosqueue.c. It parses command line parameters such as queue length and queue timeout and sends these parameters via Netlink to the particular instance of the QoSQueue qdisc. This is described in the following section. In the opposite direction, the user can retrieve the current qdisc configuration with the command tc -s qdisc.

Testing the QoSQueue

First the QoSQueue and the wireless card modules of the *Madwifi* driver have to be loaded into the Linux kernel, as shown in Listing 4.26. The modules do not depend on each other and can be loaded in any order. In contrast to insmod, the program modprobe is used to load automatically all other kernel modules the Madwifi device driver ath_pci depends on.

```
insmod qosqueue.ko
modprobe ath_pci
```

Listing 4.26: Loading QoSQueue and Madwifi Linux Kernel Modules

```
tc qdisc add dev ath0 root handle 1: prio
tc qdisc add dev ath0 parent 1:1 handle 10:   # high priority
        qosqueue limit 10 timeout 100
tc qdisc add dev ath0 parent 1:2 handle 20:   # low priority
        qosqueue limit 80 timeout 10000
tc qdisc add dev ath0 parent 1:3 handle 30:   # medium priority
        qosqueue limit 10 timeout 500
```

Listing 4.27: Linux QoSQueue Qdisc Configuration using TC

Listing 4.27 installs for the wireless interface ath0, that is used by default by the Madwifi driver, the *PRIO* queueing discipline as root qdisc. Then, three sub-queues of type *QoSQueue* are plugged into the root qdisc. Using default settings of the PRIO qdisc, the first sub-queue models the high priority queue and has a size of 10 packets and a timeout of 100 ms. Hop dependent timeouts as implemented in NS-2 are currently not supported and thus the timeout is set to a fixed value, in this case 100 ms. The second sub-queue handles best-effort or low priority traffic and has a size of 80 packets and a queue timeout of 10 s. The third sub-queue is responsible for medium priority traffic and has a maximum queue size of 10 packets and a timeout of 500 ms (refer to Section 3.3.1 for more details regarding the selected queue parameters). The PRIO qdisc classifies packets based on the TOS field of the IP header into one of the three sub-queues using its default priority map settings.

To illustrate how the priority queue works a UDP traffic generator that makes use of the *TOS* field has been developed. The commands in Listing 4.28 illustrate three UDP connections with different priorities. With the help of `tc` the current status of the queue can be monitored. When the AODV-UU routing daemon is started, routing messages are enqueued into the medium priority queue and best-effort traffic into the low priority queue according to the priority map of the PRIO qdisc.

```
// usage : ./udpc <Server-IP> <Packet Size> <TOS>
./udpc 192.168.0.1 42000  0  // low priority best-effort traffic
./udpc 192.168.0.2   200  8  // medium priority routing traffic
./udpc 192.168.0.2    64  16 // high priority game traffic
```

Listing 4.28: Using the UDP Traffic Generator

4.3.2 Broken Link Detection

Broken link detection based on feedback from the link layer allows AODV to abandon HELLO messages and detect broken links very quickly, as discussed in Section 3.4.1. This requires some communication between the device driver in kernel space and the routing protocol in user space. AODV-UU already provides a Netlink

```
// ns/aodv-uu/llf.c
// llf_print_event(struct iw_event *event, [..])
// Decode the NETLINK event
switch (event->cmd) {
    case IWEVTXDROP:
        if (mac_to_ip(&event->u.addr, &ip,
                this_host.devs[0].ifname) != 0)
            return 0;
        rt = rt_table_find(ip);
        if (rt)
            neighbor_link_break(rt);
```

Listing 4.29: Reception of a Broken Link Event in AODV-UU

interface to receive messages from the Linux kernel and in particular `IWEVTXDROP` events. To activate the code the following compilation flag has to be set in the AODV-UU `Makefile`:

```
DEFS=-DLLFEEDBACK
```

If the routing protocol receives the `IWEVTXDROP` event, triggered by the device driver due to excessive retransmissions, AODV-UU regards the according link as broken. Listing 4.29 illustrates the existing code in AODV-UU to process wireless event messages. First, the destination hardware address of the packet that has not been transmitted successfully is translated into an IP address. This IP address is used to index the routing table and retrieve the corresponding routing table entry. Finally, the routing table entry is invalidated by calling the function `neighbor_link_break`.

```
// madwifi/ath/if_ath.c
// static void ath_tx_processq([..])
if (ds->ds_txstat.ts_status & HAL_TXERR_XRETRY) {
    struct net_device *dev;
    struct sk_buff *skb;
    struct sockaddr *mac;
    union iwreq_data wrqu;       // wireless request data
    struct ieee80211_frame *wh; // IEEE 802.11 frame

    dev = ic->ic_dev;            // get device handle
    skb = bf->bf_skb;            // get socket buffer
    mac = &wrqu.addr;            // will hold MAC address
    // retrieves IEEE 802.11 frame from the socket buffer
    wh = (struct ieee80211_frame *) skb->data;
    mac->sa_family = ARPHRD_ETHER;
    // copy the MAC address into the wrqu struct
    IEEE80211_ADDR_COPY(mac->sa_data, wh->i_addr1);
    // send wireless event
    wireless_send_event(dev, IWEVTXDROP, &wrqu, NULL);
```

Listing 4.30: Madwifi Device Driver triggers Broken Link Event

In order to send the event message to the routing protocol the Madwifi driver has been modified. Listing 4.30 shows the extension of the error handling routine that is called if a packet has not been transmitted successfully. Basically, the data of the socket buffer `skb` is casted into an IEEE 802.11 frame to access the hardware address of the frame's destination. The address is stored in the *wireless request* *data union* wrqu. Finally, the IWEVTXDROP event and the union containing the MAC address are sent to the routing protocol using the Netlink interface.

The implementation works very reliably in the test environment. However, one disadvantage of the reactive approach of this mechanism is that a broken link is only detected if a packet is sent over the particular link. If no packets are sent, the link is still going to be considered as active until the route times out. If also BEACON based neighbour detection is employed the neighbour timeout should be reduced to detect lost neighbours faster.

4.3.3 Beacon based Neighbour Detection

Normally AODV HELLO messages are employed for both, broken link and neighbour detection since the AODV standard does not suggest additional mechanisms for neighbour detection. If broken link detection based on link layer feedback, as discussed in Section 3.4.1, is employed AODV does not maintain an accurate neighbour list any longer. The idea is to use the BEACON messages from the MAC layer that are periodically transmitted by every node to maintain the ad hoc network (refer to Section 2.2.2) for neighbour detection, as well. In order to implement this QoS extension, AODV and the Madwifi device driver have been extended. For communication between the device driver in kernel space and the routing protocol in user space the existing Netlink interface has been used again. Every time the

```
// ns/aodv-uu/llf.c
// llf_print_event(struct iw_event *event, [..])
// Decode the NETLINK event
switch (event->cmd) {
[..] // handle IWEVTXDROP event
    case IWEVREGISTERED:
        // convert MAC to IP address
        if (mac_to_ip(&event->u.addr, &ip,
                this_host.devs[0].ifname) != 0)
            return 0;
        rt = rt_table_find(ip);
        if (!rt)
            rt_table_insert(ip, ip, 1, 0, ACTIVE_ROUTE_TIMEOUT,
                VALID, 0, this_host.devs[0].ifindex );
        else
            rt_table_update_timeout(rt, ACTIVE_ROUTE_TIMEOUT);
```

Listing 4.31: Processing a Beacon Event in AODV-UU

device driver captures a BEACON message an IWEVREGISTERED event is triggered and passed to the routing protocol. In general, this event is employed by access points to announce new nodes.

Listing 4.31 shows the required code to act upon the reception of an IWEVREGISTERED event in AODV-UU. First, the MAC address is translated into an IP address. If a routing table entry does not already exist, a new neighbour has been found and is inserted into the routing table. If the routing table entry already exists the entry is updated. With the help of this QoS extension AODV does not rely on periodic HELLO advertisements any more and can maintain an accurate neighbour list without any additional signalling. In order to translate remote MAC addresses into IP addresses the *Address Resolution Protocol* (ARP) table has to be fully populated since the AODV-UU `mac-to-ip` implementation does not employ *Reverse ARP* (RARP) [64] to send a request to the corresponding node. If AODV-UU cannot translate the MAC address into an IP address the event is discarded. In order to gain the full power of BEACON based neighbour detection RARP should be integrated in future work.

```
// madwifi/ath/if_ath.c
// static void ath_recv_mgmt([..], struct sk_buff *skb,[..])
switch (subtype) {
case IEEE80211_FC0_SUBTYPE_BEACON:
    struct net_device *dev;
    struct sockaddr  *mac;
    union iwreq_data wrqu;
    struct ieee80211_frame * wh;
    dev = ic->ic_dev;
    mac = &wrqu.addr;
    wh = (struct ieee80211_frame *) skb->data;
    mac->sa_family = ARPHRD_ETHER;
    IEEE80211_ADDR_COPY(mac->sa_data, wh->i_addr2);
    wireless_send_event(dev,IWEVREGISTERED, &wrqu, NULL);
```

Listing 4.32: Generating a Beacon Event for Neighbour Detection

Before using the DLink wireless card some tests have been carried out with Lucent Orinoco and Cisco Aironet cards. However, both cards do not allow the device driver to capture BEACON frames if the card operates in *ad hoc* mode. Only, in *monitor* or *rfmon* mode the cards can capture all packets, but then they cannot transmit packets any more. According to the author of the Aironet device driver this is due to some limitations of the employed firmware. The Madwifi card in contrast can operate in *ad hoc* mode and capture BEACON frames at the same time and therefore has been employed in this thesis. The device driver function `ath_recv_mgmt` is called every time an IEEE 802.11 management frame has been received. In case of BEACON frames the code in Listing 4.32 is executed to send a wireless event to the routing protocol. In contrast to the previous code the source MAC address of the BEACON is stored in the wireless request union `wrqu` and the event `IWEVREGISTERED` is passed to the routing protocol.

4.3.4 Signal Strength Monitoring

The Linux kernel module part of AODV-UU already provides some support for signal strength monitoring of routing control messages to cope with the grey zone problem, discussed in Section 2.2.2. IEEE 802.11 defines the signal strength as *Received Signal Strength Indication* (RSSI) which is a vendor dependent measurement in arbitrary units and does not directly refer to the *Signal to Noise Ratio* (SNR). The employed DLink wireless cards are based on an Atheros chip-set that provides a maximum RSSI value of 60. In [65] some hints are given on how to convert RSSI values from common vendors into SNR. In order to pass a packet from the device driver to the Linux networking system socket buffers `skb` are employed. Extending the `struct sk_buff` to hold the RSSI value is an obvious step, however, it involves compiling and installing a new Linux kernel. Therefore, a different approach has been taken. The `struct sk_buff` has a control buffer field `cb`, where protocols can put their private data. This field has been used to store the RSSI value. The Madwifi driver has been extended, as illustrated in Listing 4.33, and puts the RSSI value into the `skb->cb` array before the packet is passed to the Linux networking system. If the packet is an AODV-UU routing control packet

```
// madwifi/net80211/ieee80211_input.c
// int ieee80211_input([..], struct sk_buff *skb,  int rssi) {
    int *prssi =(int*) skb->cb;
    *prssi = rssi;
    netif_rx(skb);
```

Listing 4.33: Passing the RSSI to the Routing Protocol

and the RSSI value is below a certain threshold, that can be specified at start-up of the AODV-UU daemon, the packet is dropped, as shown in Listing 4.34.

```
// aodv-uu/lnx/kaodv-main.c
// static unsigned int kaodv_hook([..], struct sk_buff **skb)
    int *prssi = (int*)(*skb)->cb;
    qual = *prssi;
    if (qual && qual < qual_th) {
        pkts_dropped++;
        return NF_DROP;}
```

Listing 4.34: Evaluating the Packet Signal Strength

4.4 Summary

In this chapter the implementation details of the proposed QoS extensions in NS-2 and under Linux have been presented.

The QoS extensions that have been implemented in NS-2 focus on AODV enhancements, priority queueing and the interaction with the AODV-UU routing protocol. The new QoS extensions can be enabled and disabled using conditional compilation flags. The backup route extension for AODV-UU attempts to provide a new route without starting a time-consuming route discovery process. The processing of RREQ and RREP messages has been extended to find further routes with the same hop count and sequence number but a different next hop as the existing route in the routing table. Such a route can be employed as a backup route. In order to provide an up-to-date neighbour list to other modules within NS-2 the routing agent provides an interface to register callback functions that are called if the neighbour list has changed. For traffic management a new priority *QoSQueue* consisting of three sub-queues using hop dependent queueing timeouts has been implemented in NS-2 to support high, medium and low priority traffic. Furthermore, the priority queue makes use of rate control to limit the amount of low priority traffic in favour of high priority neighbours.

The implementation of QoS extensions under Linux, and in particular those extensions that are currently not available in NS-2, have been described. Based on the Madwifi device driver solutions for broken link detection, BEACON based neighbour detection and signal strength monitoring have been presented. Furthermore, a new Linux queueing discipline that supports queueing timeouts to construct the priority queue has been discussed. Standard rate control of low priority traffic can be added to the queueing system by plugging in a token bucket filter.

Chapter 5

Evaluation

This chapter starts with a comparison of four routing protocols for MANETs and presents the simulation results of the QoS extensions in the light of the demands of real-time multiplayer games. The single impact as well as the accumulated effects when gradually applying the QoS extensions are presented. QoS extensions that yielded varying results for different simulation scenarios are discussed next. A summary of experiments of the QoS extensions in a real test environment completes this chapter.

5.1 Simulation Environment

The main reason for employing a network simulator to evaluate mobile wireless ad hoc networks is that it is very difficult to run experiments with many mobile nodes in a real test-environment. Furthermore, the experiments are time-consuming, error-prone and difficult to reproduce. Moreover, hardware components might not yet be available and simulation is the only way to have a first glance at new technology. On the other side, however, carrying out realistic network simulations, in particular of mobile wireless ad hoc networks, is very challenging, since most network simulators assume a simplified wireless channel and, for instance, do not account for obstacles. In addition to that, the simulation results highly depend on the network scenario consisting of the employed traffic and mobility model.

The simulations have been performed with the discrete event network simulator NS-2 [5] including the wireless extensions from the Monarch group [66, 67] to model the IEEE 802.11 MAC layer, node mobility, radio network interfaces and physical layer. Throughout the simulations, each mobile node shares a 2 Mbit/s radio channel with its neighbouring nodes, using the IEEE 802.11 MAC protocol and two-ray ground reflection model [60]. The transmission range of each node is 250 m, which is a typical value for WLAN in a free area without any obstacles.

5.1.1 Mobility Model

Many mobility models, in particular for mobile ad hoc networks, have been developed recently [68]. The most frequently employed mobility model is the *Random WayPoint* (RWP) model that is supposed to model the movement of nodes, for instance conference or museum visitors, quite reasonable compared to other models [69]. In the RWP model a node chooses randomly a location in the simulated area and moves directly towards the destination at a uniformly distributed speed. If the node arrives at the destination it pauses for some time and then randomly selects a new position as illustrated in Figure 5.1. The RWP model, however, has

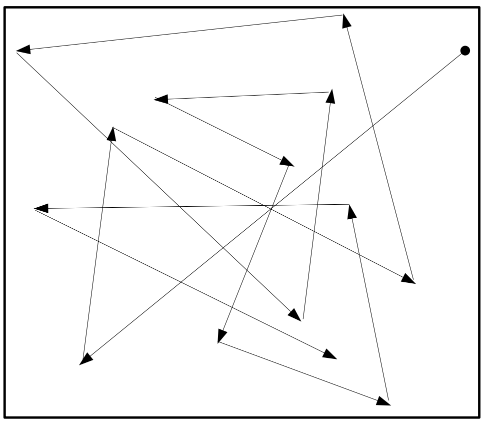

Figure 5.1: Random Waypoint Mobility Model

some serious drawbacks as well. It does not consider groups of nodes and models every node independently of all other nodes. Furthermore, the direct node movement towards the destination is not very realistic, for example, inside buildings or when modelling vehicular nodes. Moreover, the density of nodes in the centre of the simulated area is much higher than at the borders [69].

In addition to the mobility model a traffic model is needed that describes the traffic behaviour of applied applications, in this case, typical traffic of real-time multiplayer games. In general, multiplayer-games make use of small UDP packets that are exchanged between the client and server (refer to Section 2.5). According to the traffic analysis of the popular first person shooter Counter-Strike^{TM}client and server traffic are very similar, while client traffic tends to be more bursty [33]. On the average very small UDP packets with the size of 64 bytes are employed at a constant rate of 20 packets/second.

5.1.2 Metrics

In order to evaluate the performance of ad hoc routing protocols and the investigated QoS mechanisms, the following metrics have been used:

Latency: the average time in ms it takes to transmit a packet from the source to the destination.

Jitter: it describes how much the packets vary in latency and is determined by calculating the standard deviation of the latency.

Loss rate: the loss rate determines the amount of sent packets in relation to the amount of packets that have not been received successfully at the destination.

Routing overhead: the routing overhead has been calculated as the relation between the amount of routing packets and the amount of successfully transmitted data packets or bytes, respectively.

5.2 Ad Hoc Routing Protocol Comparison

An initial ad hoc routing protocol comparison has been performed to find out, which protocol has the best performance and can be used as the starting point of the work in this thesis. Four different protocols, namely *Ad Hoc On-Demand Distance-Vector* (AODV) [4], *Dynamic Source Routing* (DSR) [21], *Destination-Sequenced Distance Vector* (DSDV) [19] and *Optimized Link State Routing* (OLSR) [20] have been evaluated (refer to Section 2.3 for a detailed description of the mentioned routing protocols). NS-2's built-in implementations of DSR and DSDV, AODV-UU [58] from University of Uppsala and UM-OLSR [70] from University of Murcia have been employed in the simulations.

5.2.1 Simulation Settings

The simulation scenario consists of 20 nodes in an area of 650x650 m^2, for example a schoolyard. All nodes follow the RWP model with an uniformly distributed speed between 0-3 m/s and a pause time of 180 s. On the average 15 randomly selected unidirectional UDP data flows transmit game traffic for 150 s. For every routing protocol 60 game sessions have been simulated with a duration of 600 seconds using different seed values. The detailed simulation parameters are depicted in Table 5.1. The gained simulation results have been averaged and are presented in the following sections.

Parameter	**Value**
Simulation Time	600 s
Nodes	20
Mobility Model	Random Way Point (RWP)
Area	650x650 m^2
Speed	0-3 m/s
Pause Time	180 s
Traffic Type	CBR with 20 packet/s
Connections	15 parallel unidirectional, 150 s
Packet Size	64 bytes

Table 5.1: Parameters of the Ad Hoc Routing Protocol Comparison

5.2.2 Latency and Jitter

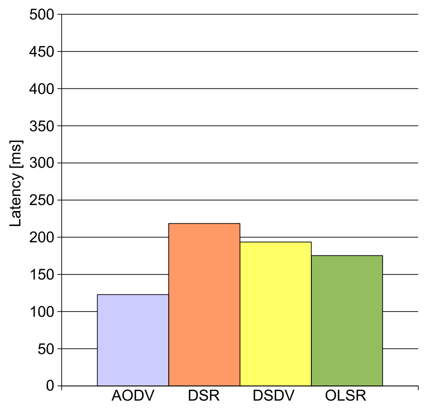

Figure 5.2: Comparison of Latency

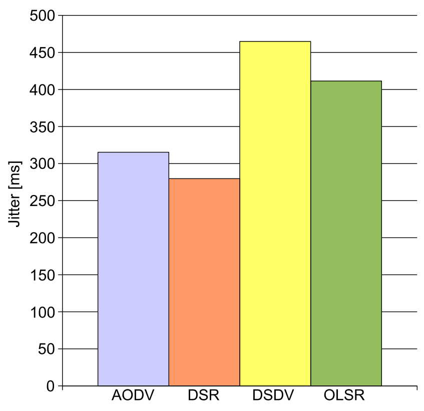

Figure 5.3: Comparison of Jitter

Figure 5.2 and Figure 5.3 illustrate the performance of latency and jitter for the four routing protocols AODV, DSR, OLSR and DSDV. AODV shows the lowest average latency of 125 ms. All other protocols have much higher latency values, in particular DSR, that has the highest latency of 218 ms, which is far from meeting the demands of multiplayer games. When comparing jitter, the reactive protocols DSR and AODV provide the lowest delay jitter around 300 ms while the proactive protocols OLSR and DSDV suffer from higher jitter values of 400 ms and 450 ms, respectively. Since the jitter in AODV is rather high compared to latency, one can assume, that latency is either rather small or very high and rarely in between. Figure 5.4 illustrates the latency characteristics of an optimal 2.3-hop connection using AODV. Due to the on-demand behaviour, AODV has to discover new routes in the first seconds of a new connection and the packets are thus delayed. Furthermore, right at the beginning, the network is highly loaded because all nodes start in parallel their route discovery process and disseminate RREQ messages throughout the network. After the starting phase, latency is much lower and mainly around 20 ms. Due to link breakages and retransmissions, there are several peaks at 100-150 ms and higher. In particular, connections that experience more data loss the graph looks differently and the amount of peaks is much higher.

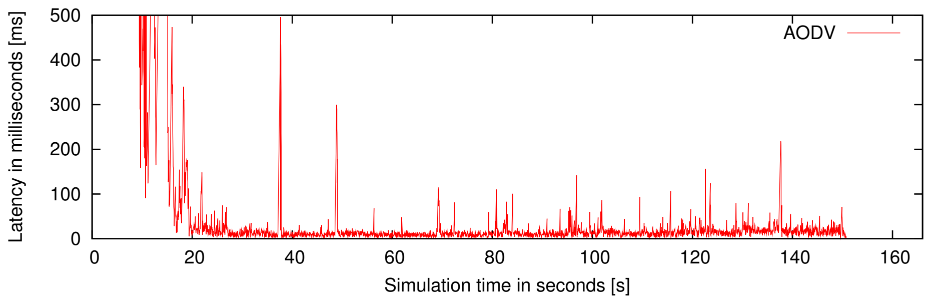

Figure 5.4: High Latency at the Beginning of AODV Connections

5.2.3 Loss Rate

The reactive protocols DSR and AODV show the lowest loss rate, namely 7 %, as depicted in Figure 5.5. The proactive protocols OLSR and DSDV have a loss rate of 13 % and 19 %, respectively, which is more than twice as high. Better loss rate results for the proactive protocols might be achieved by increasing the rate of periodic route advertisements to propagate link changes faster. Reactive protocols automatically adapt to the given scenario and send more routing messages in case of broken links to discover a new route.

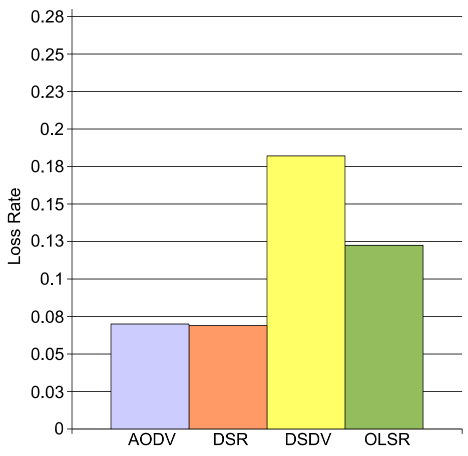

Figure 5.5: Comparison of Loss Rate

5.2.4 Routing Overhead

DSR and DSDV have the lowest packet overhead about 1 % as illustrated in Figure 5.7. The packet overhead of AODV and OLSR is of one magnitude higher, whereas AODV shows the highest overhead of 14 %. The reason that DSR has such a low packet overhead is due to intermediate nodes that can update their own routing table without any additional signalling since DSR makes use of *source routing*. Moreover, DSR operates in promiscuous mode and listens to data packets from the neighbours to learn new routes and thereby can reduce the amount of RREQs and RREPs further. DSDV sends periodically route advertisements in a very high interval and only a few routing messages are disseminated. In contrast, AODV and OLSR both broadcast periodically HELLO messages for neighbour and link management and AODV has to discover a new route more often than DSR because it does not benefit from *source routes*.

When looking at the byte overhead, as illustrated in Figure 5.6, AODV and DSR show a very similar routing overhead of 9 % and almost 8 %. This is due to the source routes employed by DSR. Every single packet contains the complete route from the source to the destination. AODV, in contrast, relies on very small routing packets. The simulated scenarios had on the average 2-hop connections. This is an advantage for DSR, that would show worse results, if the connections were longer and took 3 or more hops. DSDV has still the lowest routing overhead, however OLSR shows an overhead of 26 %. Although it does not send as many routing packets as AODV the packets are much bigger in size.

The proactive routing protocols DSDV and OLSR showed an almost constant routing overhead in all simulations. The routing overhead of on-demand protocols, in contrast, increases as the mobility of the scenario rises and thus adapts to the scenario.

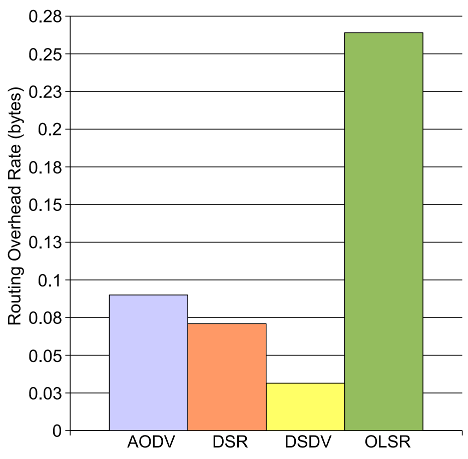

Figure 5.6: Comparison of Byte Routing Overhead

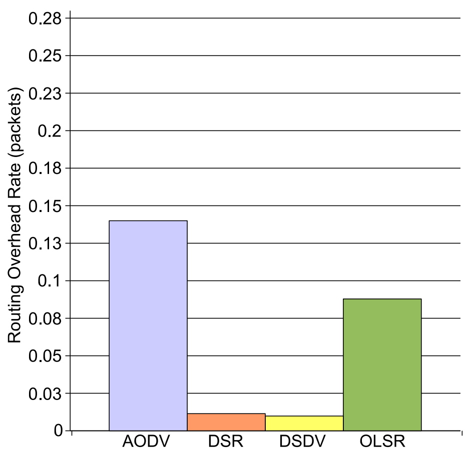

Figure 5.7: Comparison of Packet Routing Overhead

5.2.5 Summary

With regard to latency and jitter, AODV showed the best average performance while all other protocols showed much higher latency and jitter in the range of some hundred milliseconds. DSR's performance was by far the worst in these comparisons. Concerning loss rate, the reactive protocols (AODV and DSR) showed the best performance with less than 10 % packet loss while the proactive protocols (DSDV and OLSR) produced significantly higher losses. In [71] the authors identified the routing overhead in AODV is mostly due to broadcast RREQ messages whereas the routing overhead in DSR is due to unicast RREP and RERR messages. They claim that broadcast messages less utilise the wireless channel since no ACK messages and retransmissions are carried out. Furthermore, they show that AODV outperforms DSR in scenarios with increased load. Based on these initial simulation results, AODV has been selected as the routing protocol of choice for the following simulations. Nevertheless, for real-time applications, additional QoS mechanisms are required to reduce the peaks in latency and loss rate and ensure that real-time traffic arrives in-time.

5.3 Effects of the QoS Extensions

The QoS extensions discussed in the previous two chapters that aim at providing QoS for real-time multiplayer games in MANETs are evaluated in this section. The simulations of the QoS extensions have been run independently to show the single impact of each QoS extensions. Furthermore, step-wise extended simulations show the accumulated simulation results and total performance improvement.

5.3.1 Simulation Settings

The simulation settings have been chosen slightly differently from the protocol comparison, as shown in Table 5.2, to provide high priority game and low priority background traffic and distinguish between player nodes and those that are not involved in the game. AODV has been employed as the routing protocol since it has shown the best average performance in the routing protocol comparison (refer to Section 5.2).

The simulation scenario consists of 20 nodes in an area of 650x650 m^2. The nodes that do not take part in the multiplayer game follow the RWP model with a uniformly distributed speed between 0-3 m/s and a pause time of 180 seconds. Player nodes are assumed to move only rarely, within a distance of 0-15 meters, since it is rather difficult to move and play at the same time with today's available mobile gaming devices.

The multiplayer game traffic has been simulated as described in Section 5.1 and consists of three high priority bidirectional UDP data flows with a *Constant Bit Rate* (CBR) traffic of 20 packet/second and a packet size of 64 bytes. In general, there is only one game server or multiple zone servers as discussed in Section 2.5 and all clients connect to the game servers. However, the bidirectional high priority traffic flows are selected randomly among the nodes instead of connecting all to the same server node since the choice of one single game server affects the simulation results. This thesis does not deal with the optimal choice of game server nodes and to alleviate the problem the connections are selected randomly. As mentioned before, real-time multiplayer games tolerate round-trip delays in the range of 50 - 150 ms. Some packets that are slightly more delayed up to 100 ms might be still meaningful to prediction mechanisms such as dead-reckoning. Hence, real-time packets that take more than 100 ms are regarded as lost. Consequently latency and jitter of high priority packets are reduced significantly in all simulated scenarios, while the loss rate increases the more packets are outdated. Therefore, the main focus will be on minimising the loss rate when looking at the following QoS extensions.

In addition to the high priority traffic, five bidirectional low priority data flows between any two random nodes have been simulated as background traffic using the same traffic pattern as the game traffic. For every QoS extension 10 game sessions with a duration of 600 seconds and different seed values have been simulated while the simulation results have been averaged. The detailed simulation parameters are depicted in Table 5.2.

Parameter	Value
Simulation Time	600 s
Nodes	20
Mobility Model	Adapted Random Way Point (RWP)
Area	650x650 m^2
Speed	0-3 m/s
Pause Time	180 s
Traffic Type	CBR with 20 packet/s
Low Priority Traffic	5 bidirectional connections, switching every 150 s
High Priority Traffic	3 bidirectional connections, game session 600 s
Packet Size	64 bytes

Table 5.2: Simulation Scenario Parameters for QoS Extension Evaluation

5.3.2 Single Impact of the QoS Extensions

In the following, every QoS extension is compared to a basic configuration without any QoS extensions just relying on RTS/CTS which is the default setting in NS-2. This configuration has been labelled with *No QoS* in the figures. It has to be noted, that in real environments, such as the Linux wireless sub-system, RTS/CTS is not activated by default. The loss rate, latency and jitter of high and low priority traffic are separately analysed. The simulations have been carried out in the order of expected performance improvement. Only the queue timeouts and the rate control system require the priority queue and thus could not be simulated completely independently. Figures 5.8, 5.9 and 5.10 illustrate the loss rate, latency and jitter performance of high priority traffic in relation to the hop count for each QoS extensions. Since all real-time packets that arrive after 100 ms are regarded as lost, latency and jitter is rather low and do not show much variation and therefore the critical metric is the loss rate. Figure 5.11 and 5.12 show the effects of the QoS extensions on low priority traffic.

RTS/CTS Adaptation

RTS/CTS adaptation has a huge impact on all metrics for both high and low priority traffic. In all cases the performance has been significantly improved by disabling RTS/CTS. The average loss rate of high priority traffic, as illustrated in Figure 5.8, has been reduced by 50 %. When looking at 2-hop connections the loss rate is even five times lower. Latency and jitter have been reduced from 18 ms to 13 ms. On top of that, the loss rate of low priority traffic has been reduced by more than 50 % to 12 %, as depicted in Figure 5.12, and latency is four times lower than before. Furthemore, jitter has been half cut, as shown in Figure 5.11.

As a result, RTS/CTS should not be activated by default and instead a threshold should be maintained by the system that depends on the number of neighbours and the size of the data packets that have to be transmitted. A RTS/CTS threshold value of 800 bytes for five contending nodes and 180 bytes for 25 contending nodes has been reported as the optimal threshold in [72]. In [13] the authors came to a similar conclusion and suggest a RTS/CTS threshold of 500 bytes for 20 nodes and a data rate of 2 Mbit/s. Since the employed packet size in the simulation scenario is only 64 bytes RTS/CTS should be disabled.

Broken Link Detection

Broken link detection based on direct feedback from the link layer did not show the expected improvements in loss rate of high priority traffic as depicted in Figure 5.8. On the average the loss rate is even slightly higher than without this QoS extension. As link layer feedback has been considered very promising these results are rather surprising and Section 5.4.1 describes another scenario where link layer feedback has shown significant improvements. The reason it could not perform well in this scenario is that the mobility of the game players is very low and thus high priority traffic cannot benefit as much from this feature as low priority traffic does as depicted in Figure 5.11 and 5.12. Furthermore, the scenario is heavily loaded and congested and therefore the link layer reports more broken links than actually exist.

Priority Queue

By applying the priority queue the average loss rate of high priority traffic has been reduced by 15 % from 55 % to below 40 %, as depicted in Figure 5.8. In particular three-hop connections benefit from priority queueing and the loss rate has been reduced by almost 30 %. A reason for this is that without priority queueing the probability that a high priority packet reaches the destination within the 100 ms time constraint is very low, as a packet loss rate of more than 70 % shows. With the help of priority queueing, however, real-time packets are preferred and have a higher chance to be delivered in-time and thus reduces the loss rate. In case of four-hop connections this effect is alleviated and even with priority queueing many packets do not reach the destination in-time. Latency and jitter remain almost unchanged in two-hop connections. For connections with more than two hops latency and jitter have been increased by 5 ms, which might be due to the same reason mentioned before. However, regarding low priority traffic, latency and jitter have been more than doubled and show values around 2400 ms, as depicted in Figure 5.11. The loss rate of low priority traffic has been increased by 1 %. These results for the low priority traffic have been expected since the priority queue prefers high priority traffic in favour of low priority traffic.

Queue Timeouts

The queue timeout mechanism is based on the priority queue and therefore the simulation results can only be compared to the configuration that uses the priority queue. The average loss rate of high priority traffic is reduced from 40 % to 32 % by applying queue timeouts while latency and jitter have been slightly increased in all cases. At the first glance this might be surprising since high priority packets are dropped on purpose and at the same time the loss rate has been reduced. However, only already outdated packets that most likely will not arrive within the 100 ms time constraint are dropped by the queue timeout mechanism. These packets cannot be used by the destination node and are discarded and thus needlessly have consumed bandwidth. Queue timeouts have a very positive effect on low priority traffic since latency and jitter can be reduced by 50 %. Furthermore, the loss rate is reduced from 30 % to 25 % as shown in Figure 5.12. Since this timeout mechanism, in general, drops only real-time packets that are quickly out-dated, it has been expected that low priority traffic will benefit from queue timeouts. Due to the same reason high prioritry packets benefit, as well, because the available bandwidth can be used for up-to-date real-time packets that still have a chance to arrive in-time.

Backup Route

The backup route mechanism does not show any changes in the performance of high and low priority traffic and the values remain the same. This mechanisms, as well as broken link detection, that has not performed very well, attempt to improve the performance in case of higher mobility. As mentioned before, the mobility of players is very low (range of 15 m) in the employed schoolyard scenario and this might be a reason why it does not show any improvements in case of high priority traffic.

Local Repair

The built-in local repair mechanism does not work well in this scenario and slightly increases the loss rate of high and low priority traffic as illustrated in Figure 5.8 and Figure 5.12. Latency and jitter remain almost unchanged in both cases. One reason might be that repaired routes are very likely to be longer than the original route. In that case a routing control message is sent to the source node that will start the route discovery process on its own again and thus creates double routing overhead.

Rate Control

The rate control system relies on the priority queue as well and is compared to the performance results of the priority queue. Loss rate of real-time traffic is further reduced by 3 % in the average case whereas the loss rate of low priority traffic is increased by 1 % as shown in Figure 5.8 and Figure 5.12, respectively. Latency of low priority traffic remains unchanged at a very high level of 2500 ms and jitter has been increased by 100 ms from 2450 ms to 2550 ms. This outcome has been expected since the rate control mechanism limits low priority traffic.

Summary of Single Impact QoS Extensions

RTS/CTS adaptation, priority queueing, timeouts and rate control showed the highest gain in improving the loss rate of high priority traffic as illustrated in Figure 5.8. The AODV enhancements broken link detection based on link layer feedback and back up route were not convincing and did not improve the performance. Local repair even showed worse loss rate performance results. Regardless of the applied QoS mechanisms the loss rate highly depends on the number of hops. The higher

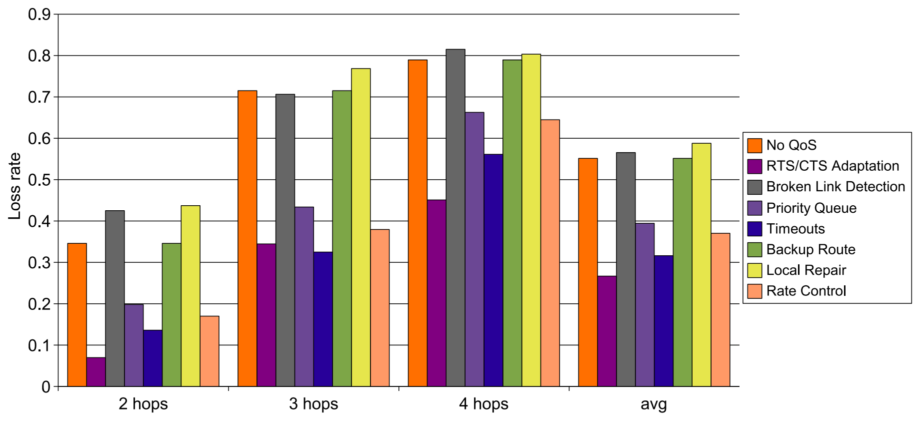

Figure 5.8: Loss Rate of Game Traffic wrt Hop Count

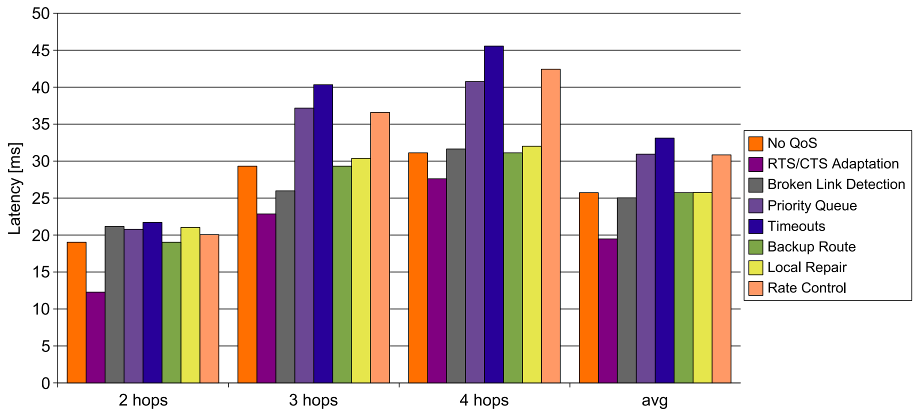

Figure 5.9: Latency of Game Traffic wrt Hop Count

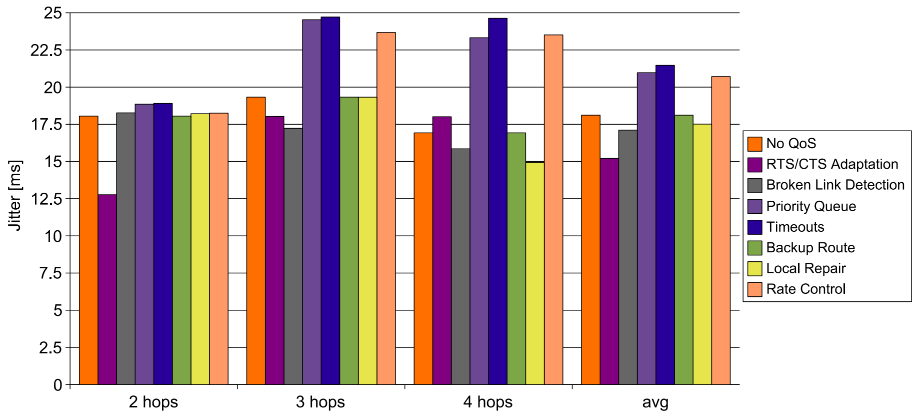

Figure 5.10: Jitter of Game Traffic wrt Hop Count

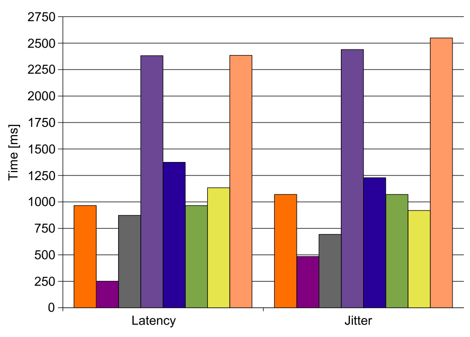

Figure 5.11: Latency and Jitter of Background Traffic

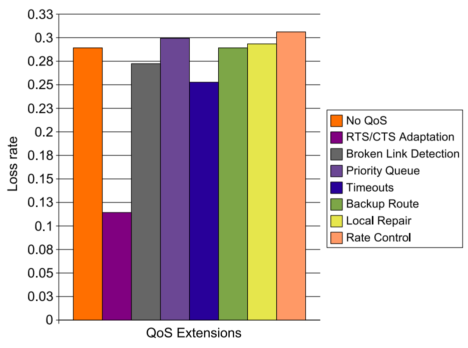

Figure 5.12: Loss Rate of Background Traffic

the hop count, the more high priority packets are discarded since they arrive too late at the destination.

5.3.3 Combined Impact of the QoS Extensions

In the following, the effects of step-wise applying the QoS extensions are investigated and the simulation results are presented. The basic AODV protocol is extended by the different QoS mechanisms such as RTS/CTS adaptation, broken link detection, priority queue, timeouts, backup routes, local repair and rate control as in the previous section. If an extension does not show the expected improvement and even degrades the performance it is not applied for the simulation of further QoS extensions. The results of the real-time traffic packet loss rate in case of end-to-end connections with different hop counts and the average loss rate is illustrated in Figure 5.13. Since latency and jitter are in the range of 25 ms-35 ms and 15 ms-22 ms, as shown in the previous section, the main focus is on reducing the loss rate of high priority traffic and keeping track of the effects of the QoS extensions on low priority traffic as depicted in Figure 5.16 and Figure 5.17.

RTS/CTS Adaptation

Since the simulation settings are the same as in the previous simulation please refer to Section 5.3.2 for a discussion of RTS/CTS adaptation. In the following simulations, RTS/CTS has been disabled.

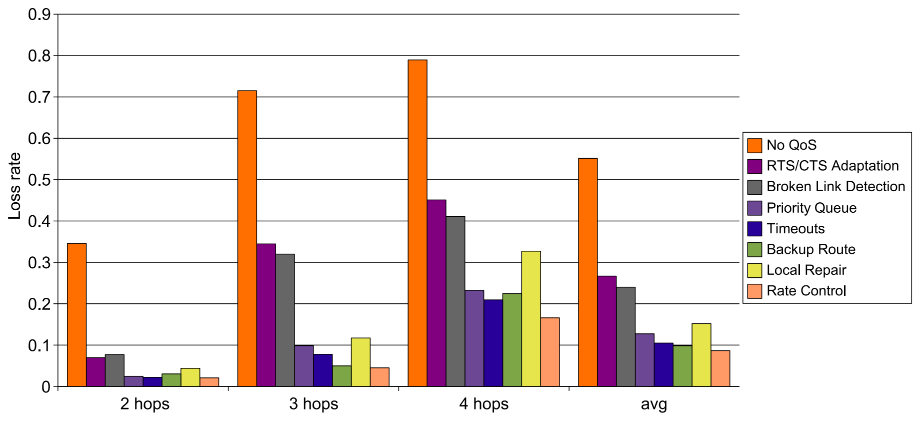

Figure 5.13: Loss Rate of Game Traffic wrt Hop Count - Combined Impact

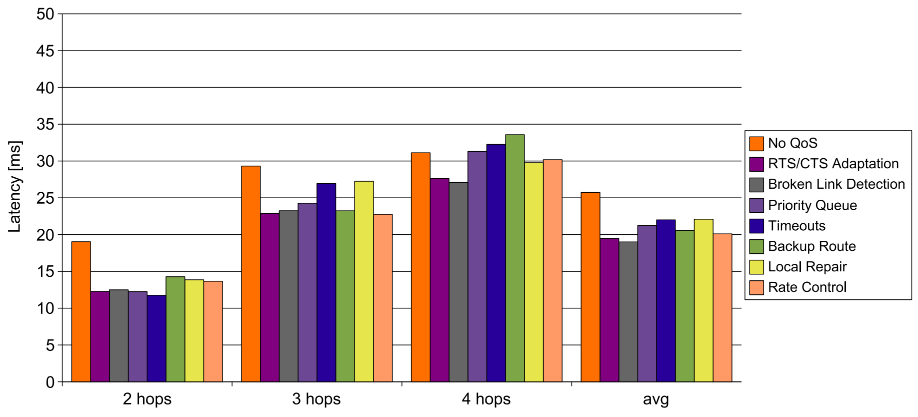

Figure 5.14: Latency of Game Traffic wrt Hop Count - Combined Impact

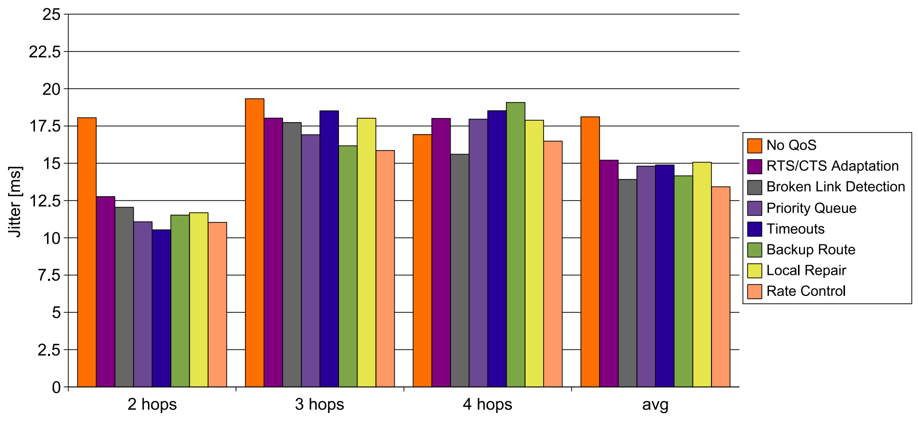

Figure 5.15: Jitter of Game Traffic wrt Hop Count - Combined Impact

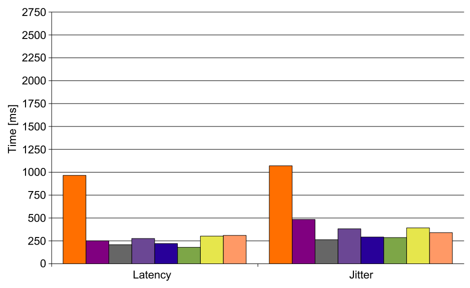

Figure 5.16: Latency and Jitter of Background Traffic - Combined Impact

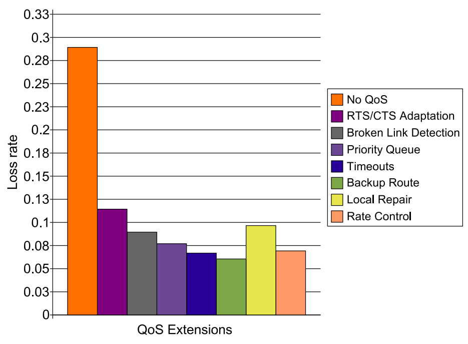

Figure 5.17: Loss Rate of Background Traffic - Combined Impact

Broken Link Detection

Using the broken link detection mechanism based on *Link Layer Feedback* (LLF) of the MAC layer, broken links can be detected very quickly. Although, the performance could not be improved significantly when broken link detection has been simulated independently, as discussed in Section 5.3.2, it reduces the loss rate of real-time game traffic by 3 % from 27 % to 24 %, as illustrated in Figure 5.13. This is rather surprising since the movement pattern of the nodes is the same in both simulations. However, the previous simulation with RTS/CTS and broken link detection (refer to Section 5.3.2) has been heavily loaded and congested. In that case the link layer might falsely report more broken links than actually exist. This can be researched by increasing the number of allowed retransmissions at the MAC layer and monitoring the amount of broken links. Figure 5.16 and Figure 5.17 show that latency, jitter and loss rate of low priority traffic have been improved as well. Still, broken link detection does not show the expected level of improvement which might be due to the limited mobility of player nodes.

Priority Queue

When employing the priority queue, the average loss rate of high priority packets has been substantially reduced from 24 % to 13 %, as illustrated in Figure 5.13, while the loss rate of two-hop connections is already below 5 %. As in the single impact simulation, three-hop connections benefit most from priority queueing and the loss rate is 22 % lower. Latency and jitter of low priority traffic are increased by 70 ms and 120 ms, respectively, as shown in Figure 5.16 and are not as much affected as in the single impact simulation where latency and jitter have been more than doubled. One reason for this is that priority queueing in this simulation does not suffer from congestion caused by the RTS/CTS mechanism in the single impact simulation (refer to Section 5.3.2). The loss rate has been reduced by 1 % as depicted in Figure 5.17 which is very similar to the single impact simulations.

Queue Timeouts

When adding timeouts to the priority queue, the loss rate of high priority traffic is slightly reduced further. Connections with up to two or three hops have a loss rate of 2 % and 8 %, respectively. 4-hop connections still have a very high loss rate over 20 % and the average loss rate is over 10 %, as illustrated in Figure 5.13. In addition, jitter and latency of low priority traffic can be also reduced. The behaviour is anlog to the single impact simulation. Again, high priority as well as low priority traffic can benefit from queue timeouts.

Backup Route

The backup route mechanism increases the loss rate of high priority traffic for two-hop connections slightly to 3 %. On the other hand, the loss rate of three-hop connections is now reduced to 5 % while the average latency and jitter are below 20 ms. Thus, two- and three-hop connections can be employed for real-time multiplayer games. The average loss rate of high priority traffic is just slightly below 10 %. Connections with four hops still suffer from a very high loss rate about 23 % and therefore cannot be used for real-time applications. The reason is that on the average high priority packets that are transmitted over four hops cannot be queued for more than 25 ms in each queue to be still able to arrive within the 100 ms time constraint. Furthermore, a four-hop connection covers a large part of the simulation area when taking the transmission ranges of the nodes on that path into account. If there is congestion somewhere in the ad hoc network the probability that the four-hop connection will be affected is high. Using the route backup mechanism, low priority traffic yields the lowest latency in all scenarios below 250 ms as depicted in Figure 5.16. The loss rate of low priority traffic is reduced as well. Although backup route did not show convincing performance results in the previous single impact simulation, it improves the average loss rate of high priority traffic and yields the best values for low priority traffic. These simulation results are interesting and further investigations have to be carried out to find out the reason why the backup route mechanism does not show any improvements in the single impact simulation and how potential interactions with other QoS extensions might look like.

Local Repair

The local repair mechanism degrades the packet loss rate of high and low priority traffic as in the single impact simulation. In particular, the average loss rate of high priority connections is increased by 5 %, as illustrated in Figure 5.13. First, this is due to the fact that the local repair process takes too much time for delivering real-time packets. Second, repaired routes tend to be longer than the original route and in this case the node sends an additional route error RERR message to the source which itself will restart the route discovery process again. Therefore, it is not recommended to use the built-in local repair mechanism of AODV in case of real-time traffic and the following QoS extension does not employ local repair.

Real-Time Neighbour Aware Rate Control

Applying the rate control mechanism the loss rate of high priority traffic has been reduced to 2 % and 4.5 % for two-hop and three-hop connections, respectively, as shown in Figure 5.13. Four-hop connections show a loss rate of 17 %, which is still too high for real-time applications. The loss rate, latency and jitter of low priority traffic are slightly affected, as depicted in Figure 5.16 and Figure 5.17. The gained improvements for high priority traffic, however, are of more importance.

5.3.4 Summary of the Overall Impact Using QoS Extensions

Interestingly, the well-known mechanisms like RTS/CTS to cope with the hidden terminal problem and local repair to handle broken links degrade the overall performance of both real-time and low priority traffic. Gradually applying the discussed QoS extensions, except AODV's local repair and the default RTS/CTS mechanism, the packet loss rate for connections up to three hops can be reduced below 5 % while the end-to-end delay and delay jitter remain in the range of 25 - 30 ms and 15 - 20 ms, respectively. Hence, for connections up to three hops the demands of real-time multiplayer games can be met by applying the QoS extensions. With the help of these QoS extensions the average loss rate of high priority traffic is more than six times lower. Deactivation of the default RTS/CTS mechanism and the use of priority queueing with timeouts showed the greatest impact in reducing the loss rate of high priority traffic. Moreover, the effect on the background traffic is also tolerable and it has not been punished drastically in favour of real-time traffic. Thus, other, not real-time applications can still use the network at the same time.

5.4 Further Simulation Results

In this section further observations of the behaviour of broken link detection in a different mobility scenario is presented.

5.4.1 Broken Link Detection

As mentioned in Section 5.3, the broken link detection mechanism did not show the expected impact. This is due to the employed simulation scenario as shown in this section. The same simulation settings as depicted in Table 5.1 have been used to test the behaviour of broken link detection in a simulation scenario with higher mobility. Since 15 short-living unidirectional UDP connections are employed the probability of experiencing broken links is much higher.

Figure 5.19 and Figure 5.18 depict the performance of broken link detection with *Link Layer Feedback* (LLF) enabled and disabled, respectively. Latency can be decreased by 50 ms and jitter is more than three times lower if LLF is activated. In addition, the loss rate can be reduced from 7 % to below 5 %. Moreover, the routing overhead is much smaller if LLF is used. In particular, the packet overhead can be reduced by four times to 3 %. The byte overhead is even below 3 %. Since AODV does not require HELLO messages for broken link detection any more if LLF is applied, the packet overhead is much lower. Latency is more than two and half times higher and jitter is almost four times higher without link layer feedback as illustrated in Figure 5.19. The behaviour of loss rate and overhead are very similar. The lost rate is almost doubled and the routing overhead is almost five times higher if link layer feedback is not activated, as shown in Figure 5.18. The higher routing overhead is due to HELLO messages that are periodically exchanged by the nodes. As a result, it is crucial to provide an implementation that supports link layer feedback when latency and jitter need to be minimised. Link layer feedback, though only simulations with AODV-UU have been carried out, is beneficial for all on-demand routing protocols and to some extent for proactive protocols as well. However, in proactive routing protocols the routing overhead cannot be reduced as significantly because the basic routing principle relies on periodic advertisements. Moreover, the amount of routing traffic is much smaller if LLF is used since AODV does not require HELLO messages for broken link detection any more.

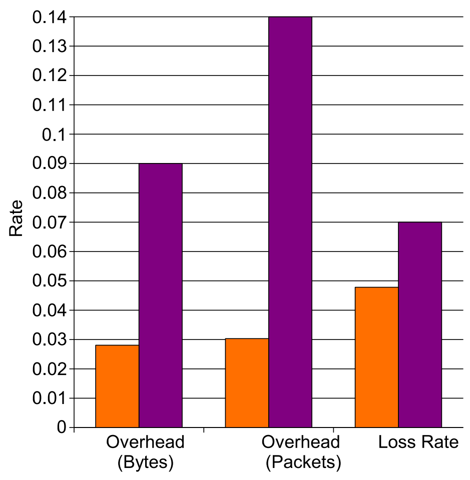

Figure 5.18: Comparison of Overhead and Loss Rate wrt Link Layer Feedback

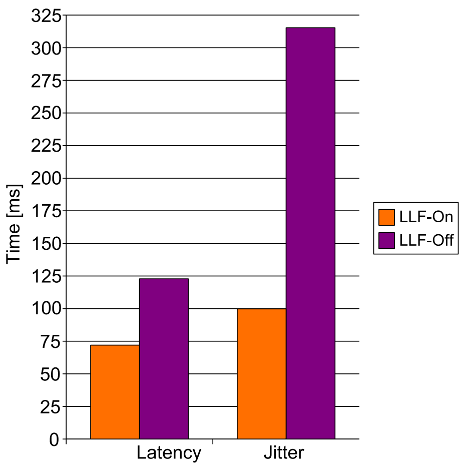

Figure 5.19: Comparison of Latency and Jitter wrt Link Layer Feedback

5.5 Test Environment

The QoS extensions BEACON based neighbour detection, broken link detection, signal strength monitoring and priority queueing with timeouts have been tested in a real environment consisting of two PCs and two laptops running Linux. Two laptops have been equipped with the modified Madwifi network driver to be able to test the QoS extensions. In case of neighbour detection, AODV-UU currently cannot translate remote MAC addresses into IP addresses and the *Reverse Address Resolution Protocol* (RARP) should be used in order to detect new nodes that do not yet have an entry in the local ARP table. For management of existing neighbours this mechanism works fine. The other mechanisms work as expected but the test environment needs to be extended to be able to carry out reproducible performance measurements. In order to test the broken link detection mechanism one of the laptops sends packets to the other laptop. When the wireless lan card of the destination laptop is disabled the sender receives a broken link detection message.

5.6 Summary

In this chapter the performance of four popular ad hoc routing protocols have been compared and evaluated. AODV has shown the best performance and has been used for the evaluation of the QoS mechanisms. First the QoS mechanisms have been simulated independently and RTS/CTS adaptation, priority queueing, timeouts and rate control showed the highest impact on improving the loss rate of high priority traffic. Next, the QoS mechanisms have been gradually applied yielding an average loss rate of real-time game traffic that is more than six times lower. Two- and three-hop real-time connections provide a packet loss rate that is below 5 % while the end-to-end delay and delay jitter remain in the range of 25 - 30 ms and 15 - 20 ms, respectively. Thus, the goal to meet the demands of real-time multiplayer games has been achieved for connections taking up to three hops. Deactivating the default RTS/CTS mechanism and the use of priority queueing with timeouts showed the most impact in reducing the loss rate of high priority traffic as in the simulation before. Moreover, the effect on the background traffic is also tolerable and it has not been punished drastically in favour of real-time traffic. Thus, other, not real-time applications can still use the network at the same time. In the last part of this chapter, it has been shown that the use of link layer feedback based broken link detection makes a huge difference in latency, jitter and packet loss in scenarios with higher mobility. Finally, the results of the QoS extensions in the real test environment have been described briefly.

Chapter 6

Conclusions and Outlook

This chapter concludes the thesis with the presentation of the achieved results and gives some ideas and proposals for future work and following projects.

6.1 Conclusion

Quality of Service provisioning in mobile wireless ad hoc networks for real-time applications, such as multiplayer games, has been investigated and evaluated in this thesis. Real-time multiplayer games demand low latency and jitter connections with a maximum round-trip delay of 150 ms and packet loss rate of 5 %. Packets that have an end-to-end delay of up to 100 ms can be still useful to prediction mechanisms. It has been shown in network simulations employing standard routing protocols that meeting these demands in a mobile wireless ad hoc network is a challenging task.

Four popular routing protocols, AODV, DSR, DSDV and OLSR particularly targeting at MANETs have been compared and evaluated in the network simulator NS-2. The simulation results show that the reactive routing protocol AODV outperforms the other routing protocols in the employed multiplayer game scenario and provides the lowest end-to-end delay of 125 ms or 250 ms round-trip delay assuming a symmetric connection, respectively. Together with the reactive protocol DSR, AODV also shows the lowest loss rate of 7 %. The proactive routing protocols DSDV and OLSR do not cope as well with mobility and show much higher loss rates of 19 % and 13 %. Since AODV provides the lowest latency it has been employed as the basic routing protocol throughout this thesis for further investigation.

AODV's average end-to-end latency of 125 ms is still too high to meet the demands of real-time multiplayer games. This is mainly due to four challenges real-time applications are facing in MANETs: (1) mobility, (2) congestion, (3) shared medium and (4) wireless signal. In order to tackle specifically these challenges, AODV enhancements, traffic management and MAC layer support QoS extensions have been analysed. In addition to the built-in mechanisms of NS-2 and AODV, the QoS extensions backup route, neighbour list, priority queue with hop dependent queueing timeouts and real-time neighbour aware rate control have been implemented in the network simulator NS-2. Since BEACON based neighbour detection requires major changes to the NS-2 IEEE 802.11 MAC layer it could not be implemented in NS-2 within the time constraints of this thesis. To the same reason signal strength monitoring has not been used in the simulations since the signal strength values provided by the employed two-ray ground propagation model are not accurate enough.

The results of the QoS extension simulations show how the different QoS mechanisms influence the performance of real-time multiplayer games. Packets that do not arrive at the destination within 100 ms are regarded as dropped as they are not meaningful to the multiplayer game anymore. Thus, in all QoS extension simulations latency and jitter are in the range of 20 - 35 ms and 15 - 22 ms, respectively, in cost of the loss rate that is in the average case more than 55 %. Therefore, the main focus has been on reducing the loss rate to below 5 % to meet the demands of real-time multiplayer games. When comparing the QoS extensions separately, RTS/CTS adaption, that is, disabling RTS/CTS in the case of small real-time packets, has the biggest impact and the average loss rate of real-time traffic has been reduced by more than 50 %. Another huge impact makes priority queueing that reduces the loss rate by 15 %, followed by queueing timeouts with 10 % and neighbour aware rate control with 3 %. Backup route and broken link detection do not reduce the loss rate in the expected way. This is due to the employed simulation scenario. Nodes that act as game players and send real-time traffic do not experience much mobility in the employed scenario. Since the average hop count is two hops, the influence of other nodes with a higher mobility rate is low. In a different simulation scenario with higher mobility it has been shown that broken link detection significantly improves latency and loss rate.

By gradually applying RTS/CTS adaption, priority queueing with timeouts and real-time neighbour aware rate control combined with broken link detection and backup routes, the average loss rate has been reduced significantly from more than 55 % to below 10 %. Again, the biggest impact was shown by RTS/CTS adaption and priority queueing with timeouts. However, broken link detection has reduced the loss rate as well this time. Since there is less network congestion the probability of interference and collisions is lower and fewer links are falsely regarded as broken. For connections up to three hops AODV with the applied QoS mechanisms meets the demands of real-time multiplayer games and the loss rate has been scaled down more than 15 times from 35 % to below 3 % for 2-hop and from 70 % to 5 % for 3-hop connections, respectively. Longer connections taking more than three hops have been also improved significantly. The loss rate is four times lower in the case of 4-hop connections but they still suffer from a very high loss rate of about 17 %. Furthermore, it has been shown that priority queueing and rate control do not excessively affect low priority background traffic. The average latency and jitter of low priority traffic is around 300 ms and the loss rate of 7 %. Moreover, it has been shown that common mechanisms like local repair of AODV and using RTS/CTS for small real-time packets degrade the overall performance and should not be employed in case of real-time traffic.

Signal strength monitoring, BEACON based neighbour detection, broken link detection and priority queueing with fixed timeouts have been implemented in a real test environment running Linux. To accomplish this, the Madwifi wireless card Linux device driver has been extended with support for BEACON based neighbour detection, broken link detection and signal strength monitoring. For communication between the device driver and the routing protocol AODV-UU the Netlink interface has been used. Furthermore, the priority queueing system extended with queueing timeouts has been implemented as a Linux kernel module. These QoS extensions have been installed and tested in a test environment and can be employed by real-time applications or by the service provisioning framework SIRAMON.

In this thesis, it has been shown in the network simulator NS-2 that with the proposed QoS extensions the demands of real-time multiplayer games employing standard IEEE 802.11 technology can be met for short connections up to three hops. Due to the probabilistic nature of IEEE 802.11 this still does not allow to give any hard QoS guarantees. Furthermore, the results have been achieved for particular game scenarios with 20 nodes including three game players. In a different scenario with more game players it is likely that the QoS demands cannot be met anymore. Furthermore, it is not possible to transfer these results directly to the real world due to the simplifications that have been used throughout the simulations within the network simulator NS-2 (mobility model, traffic model, wireless channel). Nevertheless, this thesis shows which QoS extensions are promising to improve Quality of Service and gives some hints how the mechanisms can be implemented in a real wireless network environment.

6.2 Outlook

As faster wireless network technologies will be available soon, for instance, IEEE 802.11n aiming at a theoretical bandwidth of 540 Mbit/s and *Multiple Input Multiple Output* (MIMO) technology providing separate antennas in a wireless network card to be able to send and receive data at the same time more players will be able to play a real-time multiplayer game concurrently. Nevertheless, without modifying the actual contention based medium access mechanism of IEEE 802.11 MAC layer, the same problems that have been discussed in this thesis remain and fundamental changes to the MAC layer are required to overcome the problems of today's IEEE 802.11 distributed medium access.

There are various interesting directions to extend the work and further investigate Quality of Service in mobile wireless ad hoc networks based on the results achieved in this thesis. Some of them are presented in the following:

AODV / NS-2 Improvements: The reactive routing protocol DSR benefits from source routing during the route discovery process. AODV can be extended using AODV option headers to employ source routing in RREQ and RREP messages, as well. First, this will reduce the amount of required routing control messages and second, other mechanisms like backup route can also benefit since a node is aware of more routes. Furthermore, it has been shown that the reactive protocols AODV and DSR outperform proactive routing protocols in scenarios with higher mobility. However, for high priority connections in combination with signal strength monitoring, periodic dissemination of low priority RREQ can proactively find stable routes before a route breaks. Thereby, the queueing system has to be extended to provide for a low priority queue with short timeouts. BEACON based neighbour detection and management showed excellent results in the test environment. This feature should be implemented into the NS-2 MAC layer to be able to simulate it under various conditions. In addition to that, the IEEE 802.11e QoS enabled MAC protocol which provides some QoS directly on the MAC layer should be investigated and integrated into the existing QoS extensions in NS-2.

Simulation Scenarios: As it has been shown in the case of broken link detection the actual performance of the QoS extensions highly depend on the employed simulation scenario. The QoS extensions should be simulated under various scenarios with different mobility and traffic pattern to find out which QoS extensions are most effective under particular simulation conditions. Thereby, it is important to apply as realistic simulation scenarios as possible.

QoS Guarantees: In order to give any QoS guarantees the system has to be able to detect QoS violations. Either this is handled by every application which might be rather inefficient or the system provides a means to do this. An interesting approach that does not require additional signalling is to measure the delay between sending a frame on the MAC layer and the reception of the corresponding acknowledgement frame as it is done in the SWAN framework. If the delay is constantly increasing and still some real-time packets are sent over that particular link the routing protocol can try to proactively find a better route.

Prediction Models: Based on some measurements from the MAC layer such as the signal strength and the delay between transmission of a frame and the reception of the ACK frame mobility prediction models and congestion prediction models can be investigated. With the help of such prediction models the routing protocol can proactively cope with the situation and for instance, discover a better route or notify the application to reduce the bit rate of high priority traffic and so forth.

Test Environment: Although, network simulation is currently the only practical tool to investigate and evaluate large mobile wireless networks, a proper test environment is mandatory to make sure that the simulation and the real environment show confirming results. A test-bed needs to be set up that allows to easily configure and measure different mobility and traffic scenarios in a reproducible manner.

Appendix

Appendix A

Network Simulator NS-2

NS-2 is a discrete event network simulator that is used by many researchers. It is freely available and open source. In this thesis, the *ns-allinone* package release 2.28 (released February 3, 2005) as it is available on the NS-2 homepage [5] has been employed. This package contains all required components used for running NS-2 and is available for several platforms and computer architectures. The core of NS-2 is written in C++ while it provides an object oriented version of the script language Tcl, called OTcl, as user interface for configuration scripts. In addition to that, there is an interface that allows to access C++ objects from OTcl scripts and vice versa. The combination of both languages provide a powerful means in sense of performance and ease of use of the simulator. However, in some cases the distinction between C++ and OTcl is not consistent any more and fundamental parts of the network simulator are implemented in OTcl, as well. In general, wireless simulations in NS-2 involve the following steps:

1. Implementation or modification of a protocol, for instance, routing protocol in C++ and OTcl

2. Specification of the simulation settings, mobility model and traffic model in OTcl

3. Running the simulations

4. Analysis of the trace files and visualisation in the *Network Animator* (NAM) [73]

A.1 OTcl Simulation Scripts

Listing A.1 shows a typical OTcl simulation template script that defines the simulation settings and includes the mobility and traffic models that are itself written in OTcl (refer to Section A.1.1 and A.1.2).

```
# ns-2/sim/wifi-sim.tcl
# create a new ns simulator object
set ns [new Simulator]
# enable tracing
set  nstrace [open $opt(tr) w]
$ns trace-all $nstrace
# create a rectangle like area
set topo [new Topography]
$topo load_flatgrid $opt(x) $opt(y) 1
# create a wireless channel
set channel [new $opt(chan)]
# basic node configuration
$ns node-config -adhocRouting  $opt(rp)\    ;# AODV-UU
                 -llType       $opt(ll)\
                 -ifqType      $opt(ifq)\   ;# QoSQueue
                 -ifqLen       $opt(ifqlen)\;#
                 -macType      $opt(mac)\   ;# Mac/802_11
                 -antType      $opt(ant)\   ;# OmniAntenna
                 -propType     $opt(prop)\  ;# TwoRayGround
                 -phyType      $opt(netif)\ ;# WirelessPhy
                 -topoInstance $topo\
                 -channel      $channel
# creating the nodes
for {set i 0} {$i < $opt(nn) } {incr i} {
    set node($i) [$ns node]
}
# include mobility model
source $opt(scene)
# include traffic model
source $opt(traffic)
# add an event to shutdown the simulation
$ns at $opt(stop)1 "stopSimulation"
# start the actual simulation
$ns run
```

Listing A.1: NS-2 Wireless Simulation OTcl Script

A.1.1 Node Mobility

The OTcl mobility model file describes at which time and speed nodes move from one point in the simulation field to another. Listing A.2 shows an exemplary movement pattern for `node(1)`. First, the start position of `node(1)` is determined. At simulation time 5.0 and 130.0 `node(1)` moves towards a new position. More complex mobility patterns have to be modelled using the basic primitive `setdest`.

```
# initial position of node(1)
$node(1)  set X_ 320
$node(1)  set Y_ 620
$node(1)  set Z_ 0   ;# z-axis is always 0 in NS-2

# event to move to location (18,356) at a speed of 1.0 m/s
$ns at 5.0 " $node(1) setdest 18 356 1.0"

# event to move to location (572,525) at a speed of 0.5 m/s
$ns at 130.0 " $node(1) setdest 572 525 0.5"
```

Listing A.2: OTcl Node Mobility Script

A.1.2 Traffic Generation

NS-2 provides several application protocols, such as *Constant Bit Rate* (CBR) and *File Transfer Protocol* (FTP) and the transport protocols UDP and TCP among others. Listing A.3 illustrates a OTcl script to set up a UDP connection sending CBR traffic from `node(1)` to `node(2)`. In order to generate mobility and traffic files for different scenarios a Java tool, called Mobigen, has been developed.

```
# create a UDP agent
set udp(1) [new Agent/UDP]
# mark the flow with an id
$udp(1) set fid_ 1
# mark the flow as high priority
$udp(1) set prio_ 2
# attach the UDP agent to node(1)
$ns attach-agent  $node(1) $udp(1)

# create a UDP sink object
set null(1) [new Agent/Null]
# attach the UDP sink to node(2)
$ns attach-agent  $node(2) $null(1)

# create traffic at a constant bit rate
set cbr(1) [new Application/Traffic/CBR]
# 64 byte packets
$cbr(1)  set packetSize_ 64
# send 20 packets / second
$cbr(1)  set interval_ 0.05
# attach the traffic generator to the UDP agent
$cbr(1)  attach-agent $udp(1)
# transmit 3000 packets in total
$cbr(1)  set maxpkts_ 3000

# connection between node(1) and node(2)
$ns connect $udp(1) $null(1)

# start the transmission at 5.0
$ns at 5.0 " $cbr(1)  start "
```

Listing A.3: OTcl UDP Traffic Script

Appendix B

AODV-UU Overview

AODV-UU [58] has been developed at the University of Uppsala and is available for NS-2 and Linux, both sharing the same code for the core routing algorithm. Thus, the routing algorithm can be extended and verified in NS-2 and then tested in a real test-bed using Linux. The original version of AODV-UU has been implemented in C for Linux and later ported to NS-2 using C++. Still the same code base is used. The Linux kernel specific part of the code resides in the `lnx` sub-directory whereas the code that contains NS-2 specific parts can be found in the `ns` sub-directory. The main directory contains the core routing algorithm that is employed in both versions. In this thesis, AODV-UU 0.9 has been employed and later manually upgraded to AODV-UU 0.91.

B.1 Important Macros

In order to achieve a concurrent implementation for Linux in C and NS-2 in C++, AODV-UU makes extensive use of conditional compilation. The main macros are defined in `defs.h`:

NS_PORT: this macro is used to indicate NS-2 specific code and provides a glue layer to map the networking system in NS-2 to Linux.

NS_CLASS: stands for `AODV-UU::` in NS-2 and is used to provide a C++ version of AODV-UU. If AODV-UU is compiled for Linux this macro is just empty.

B.2 Packet Processing of AODV-UU in NS-2

Every time the AODV-UU routing agent receives a packet the function `recv` in the class `AODV-UU` is called. Either the packet has been sent by a different node or by an application at the same node. In both cases the `recv` function is called. Depending on the packet type either `recvAODVUUPacket` in the case of AODV routing control packets or `processPacket` for data packets is called. In the first case, the packet is converted from the NS-2 packet format into the Linux format. Next, the function `aodv_socket_process_packet` is called which delegates the AODV message depending on its type to one of the processing routines. In the latter case, the data packet has either reached the destination node and can be handed to the application layer `target_->recv()` or if a route is available the packet is forwarded to the next hop calling `sendPacket`. Otherwise, the route discovery process is started. AODV's packet processing steps are described in more detail in the following sections whereas the sender of a *route request* RREQ message is always the *source* and the sender of a *route reply* (RREP) message is always the *destination* node.

B.2.1 Route Request Processing

If the packet is a RREQ message the function `rreq_process` is executed. At the beginning several consistency checks are performed to make sure that the size of the packet is valid, the originator is not the local node and the RREQ has not been processed before. If the message passes these checks the RREQ is stored to avoid repetitively processing the same RREQ. Afterwards, potential RREQ extensions are processed. Next, if the routing entry to the originator of the RREQ is unknown, a new routing table entry is inserted into the routing table using the previous hop of the RREQ as the next hop on the route to the originator of the RREQ.

An existing routing entry timer `rt->rt_timer` is updated either if the RREQ has a more up-to-date sequence number of the originator than the existing routing table entry or when they are equal or the hop count of the RREQ is lower or the routing table entry was marked `INVALID`. To update the routing entry and the timer `rt_table_update` and `rt_table_update_timeout` are called, respectively. The lifetime of the new entry depends on the hop count of the received RREQ. If the hop count is low the lifetime is higher and vice versa. If the route has become `VALID` and some packets are waiting for delivery to that destination they are dequeued from the routing queue and forwarded to the destination in the function `packet_queue_set_verdict`.

If the node is the destination of the RREQ, a RREP is created (`rrep_create`) and unicast to the originator (`rrep_send`). If the node is an intermediate node all along the way from the source to the destination the node can reply to the RREQ if it knows an active route and the sequence number is at least as fresh as the one in the RREQ. Additionally, the destination only `d-flag` in the header of the RREQ must not be set. The intermediate node sends a RREP to the originator and if the `g-flag` is set it sends a RREP to the destination, as well. Finally, if the node does not know the route it reduces the `ttl` field and broadcasts the RREQ to its neighbours (`rreq_forward`).

B.2.2 Route Reply Processing

If the packet is a RREP message the function `rrep_process` is executed. At the beginning, the hop count field of the RREP is incremented to account for the additional hop and optional RREP extensions are processed. Next, if no *forward route* to the destination exists, a new route is created. Otherwise, an existing route is updated. If the A-RREP flag is set an ACK-RREP message is send back to the destination. If the current node is not the source the RREP is forwarded along the reverse route towards the source.

B.2.3 Route Discovery

If currently no route to a particular node is available and packets have to be delivered to that node, the route discovery process is started in `rreq_route_discovery`. Before sending a RREQ the `seek_list` is searched for any on-going route discovery processes to that particular destination. If it finds a valid entry another route discovery is not started again. If no entry can be found, a new RREQ message is created in `rreq_create` and then the node broadcasts the RREQ to its neighbours (`aodv_socket_send`). If expanding ring search is disabled the `ttl` field is set to `NET_DIAMETER`. Otherwise the first RREQ is sent applying either the last known hop distance as `ttl` to avoid multiple timeouts or `ttl` is set to `TTL_START`. The `tll` is increased by `TTL_INCREMENT` each time if the RREQ could not discover a route within the timeout `RING_TRAVERSAL_TIME`. If the route discovery times out `route_discovery_timeout` is called and the discovery process is started again with a higher `ttl` and higher timeout. The new timeout is calculated using an exponential backoff algorithm. If the `ttl` is higher than `TTL_THRESHOLD`, `NET_DIAMETER` is used as the maximum `ttl`. The route discovery process is started `RREQ_RETRIES` times with `ttl` set to `NET_DIAMETER`, until an error messages is presented to the application layer and all packets for that destination are removed and dropped from the routing queue.

B.2.4 Timer Management

AODV relies heavily on timers to maintain the routing table. Active routes timeout if they have not been used for a while and are marked as `INVALID`. After another timeout the entry is finally removed from the routing table to save memory. The routing table timer `rt->rttimer` is updated by the function `rt_table_update_timeout` every time a data packet has been received or an AODV message has been processed.

Appendix C

CD-ROM

Figure C.1 shows the directory tree of the CD-ROM. Every directory contains a `README` file that describes the packages and programs in the particular directory.

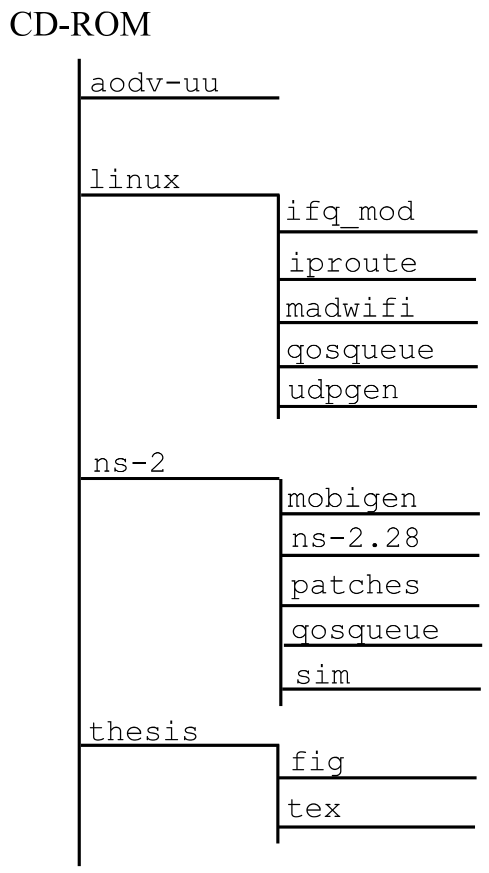

Figure C.1: Directory Structure of the CD-ROM

Bibliography

[1] K. Farkas. SIRAMON - Service provIsioning fRAMework for self-Organized Networks. ETH Zurich, January 2005. http://www.csg.ethz.ch/research/projects/siramon/.

[2] ETH Zurich. Computer Engineering and Networks Laboratory. http://www.tik.ee.ethz.ch.

[3] IEEE-SA Standards Boards. Part11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. 12 June 2003. http://standards.ieee.org/getieee802/802.11.html.

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

[5] Information Sciences Institute ISI. The Network Simulator ns-2, April 2005. http://www.isi.edu/nsnam/ns/.

[6] Andrew S. Tanenbaum. *Computer networks (4th ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.

[7] S. Herzog. RSVP Extensions for Policy Control. RFC 2750 (Proposed Standard), January 2000.

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

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

[10] IEEE-SA Standards Boards. Part16: Air Interface for Fixed Broadband Wireless Access Systems. 13 May 2004.
http://standards.ieee.org/getieee802/802.16.html.

[11] IEEE-SA Standards Boards. Part 15.1: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Wireless Personal Area Networks (WPANs(tm). 14 June 2002.
http://standards.ieee.org/getieee802/802.15.html.

[12] A. Jayasuriya, S. Perreau, A. Dadej, and S. Gordon. Hidden vs. Exposed Terminal Problem in Ad Hoc Networks. 2004.

[13] Z. Kong, D. Tsang, and B. Bensaou. Adaptive RTS/CTS Mechanism for IEEE 802.11 WLANs to Achieve Optimal Performance. In *Proceedings of IEEE International Conference on Communications*, volume 1, 2004.

[14] M. Heusse, F. Rousseau, G. Berger-Sabbatel, and A. Duda. Performance Anomaly of 802.11b. In *Proceedings of IEEE INFOCOM 2003*, San Francisco, USA, March-April 2003.

[15] IEEE 802.11 WG. Draft Amendment to Standard Information Technology-Telecommunications and Information Exchange Between Systems- Local and Metropolitan Area Networks- Specific Requirements- Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Medium Access Control (MAC) Enhancements for Quality of Service (QoS). February 2004. IEEE 802.11e/Draft 8.0.

[16] Priyank Garg, Rushabh Doshi, Russell Greene, Marry baker, Majid Baker, Majid Malek, and Xiaoyan Cheng. Using IEEE 802.11e MAC for QoS over Wireless. In *22nd IEEE International Performance Computing and Communications Conference (IPCCC)*, Phoenix, Arizona, USA, April 2003.

[17] Henrik Lundgren, Erik Nordström, and Christian Tschudin. Coping with communication gray zones in IEEE 802.11b based ad hoc networks. In *WOWMOM '02: Proceedings of the 5th ACM international workshop on Wireless mobile multimedia*, pages 49–55, New York, NY, USA, 2002. ACM Press.

[18] Charles E. Perkins. *Ad Hoc Networking*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[19] Charles E. Perkins and Pravin Bhagwat. Highly dynamic Destination-Sequenced Distance-Vector routing (DSDV) for mobile computers. In *SIGCOMM '94: Proceedings of the conference on Communications architectures, protocols and applications*, pages 234–244, New York, NY, USA, 1994. ACM Press.

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

[21] David B. Johnson, David A. Maltz, and Yih-Chun Hu. The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR). Internet-draft, IETF MANET Working Group, July 2004. Expiration: January 2005.

[22] Zygmunt J. Haas and Marc R. Pearlman. The Zone Routing Protocol (ZRP) for Ad Hoc Networks. Internet-draft, IETF MANET Working Group, November 1997. Expiration: May, 1998.

[23] Zygmunt J. Haas. A New Routing Protocol For The Reconfigurable Wireless Networks. In *Proceedings of 6th IEEE International Conference on Universal Personal Communications, IEEE ICUPC'97, October 12-16, 1997, San Diego, California, USA*, volume 2, pages 562–566. IEEE, IEEE, October 1997.

[24] Prasun Sinha, Raghupathy Sivakumar, and Vaduvur Bharghavan. CEDAR: A Core-Extraction Distributed Ad Hoc Routing Algorithm. In *Proceedings IEEE INFOCOM 1999*, pages 202–209, March 1999.

[25] Guangyu Pei, Mario Gerla, and Xiaoyan Hong. LANMAR: Landmark Routing for Large Scale Wireless Ad Hoc Networks With Group Mobility. In *MobiHoc*, pages 11–18, 2000.

[26] Dimitri P. Bertsekas and Robert Gallager. *Data Networks*. Prentice Hall, 1991.

[27] C.L. Hedrick. Routing Information Protocol. RFC 1058 (Historic), June 1988. Updated by RFCs 1388, 1723.

[28] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[29] L. Viennot. Complexity Results on Election of Multipoint Relays in Wireless Networks. Technical report, INRIA, 1998.

[30] S. Guha and S. Khuller. Approximation Algorithms for Connected Dominating Sets. *Algorithmica*, 20(4):374–387, 1998.

[31] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, New York, USA, 1983.

[32] David B. Johnson and David A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. In Thomasz Imielinski and Hank Korth, editors, *Mobile Computing*, volume 353, chapter 5, pages 153–181. Kluwer Academic Publishers, 1996.

[33] Johannes Faerber. Network Game Traffic Modelling. In *Proceedings Workshop on Network and System Support for Games*, pages 53–57, April 2002.

[34] C. E. Perkins and E. M. Belding-Royer. QoS for Ad hoc On-Demandn Distance Vector Routing. Internet-draft, IETF MANET Working Group, October 2003. Expiration: April, 2004.

[35] I. Jawhar and J. Wu. *M. Cardei, I. Cardei, D.-Z. Du (Eds.), Resource Management in Wireless Networking*, chapter :Quality of Service Routing in Mobile Ad Hoc Networks. Kluwer, 2004.

[36] Z. Wang and J. Crowcroft. QoS Routing for Supporting Resource Reservation. *IEEE Journal on Selected Areas in Communications*, 14:1228–1234, 1996.

[37] Qi Xue and Aura Ganz. Ad Hoc QoS On-Demand Routing (AQOR) in Mobile Ad Hoc Networks. *J. Parallel Distrib. Comput.*, 63(2):154–165, 2003.

[38] S. Mueller, R. P. Tsang, and Dipak Ghosal. Multipath Routing in Mobile Ad Hoc Networks: Issues and Challenges. volume 2965, pages 209–234, April 2004.

[39] S. Yusuke. AODV Multipath Extension Using Source Route Lists with Optimized Route Establishment. In *Proceedings of International Workshop on Wireless Ad-hoc Networks (IWWAN '04)*, May 2004.

[40] C. Siva Ram Murthy and B. S. Manoj. *Ad Hoc Wireless Networks*. Prentice Hall International, 2004.

[41] Geunhwi Lim. Link Stability and Route Lifetime in Ad-hoc Wireless Networks. In *ICPPW '02: Proceedings of the 2002 International Conference on Parallel Processing Workshops*, Washington, DC, USA, 2002. IEEE Computer Society.

[42] Gahng-Seop Ahn and Andrew T. Campbell and Andras Veres and Li-Hsiang Sun. SWAN: Service Differentiation in Stateless Wireless Ad Hoc Networks. In *Proceedings IEEE INFOCOM 2002*, pages 457–466, June 2002.

[43] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), September 2001.

[44] H. Arora, L. Greenwald, U. Rao, and J. Novatnack. Performance Comparison and Analysis of two QoS Schemes: SWAN and DiffServ. In *Drexel Universitiy Research Day*, April 2003.

[45] Xiaowei Zhang and Seoung-Bum Lee and Gahn-Seop Ahn and Andrew T. Campbell. INSIGNIA: An IP-based Quality of Service Framework for Mobile Ad Hoc Networks. In *Journal of Parallel and Distributed Computing, Special Issue on Wireless and Mobile Computing and Communications*, volume 60, pages 374–406. Academic Press, Inc., April 2000.

[46] H. Xiao, W. Seah, A. Lo, and K. Chua. A Flexible Quality of Service Model for Mobile Ad-Hoc Networks. In *Proceedings IEEE Vehicular Technology Conference*, pages 445–449, May 2000.

[47] Christian Bonnet and Navid Nikaein. A Glance at Quality of Service Models for Mobile Ad Hoc Betworks. In *16eme Congrès DNAC (De Nouvelles Architectures pour les Communications), December 2-4, 2002, Paris, France*, Dec 2002.

[48] Kai Chen, Samarth H. Shah, and Klara Nahrstedt. Cross-Layer Design for Data Accessibility in Mobile Ad Hoc Networks. *Wirel. Pers. Commun.*, 21(1):49–76, 2002.

[49] S.M. Riera, O. Wellnitz, and L. Wolf. A Zone-based Gaming Architecture for Ad-Hoc Networks. In *Proceedings of the Workshop on Network and System Support for Games (NetGames2003)*, Redwood City, USA, May 2003.

[50] F. Maurer. Service Management Procedures Supporting Distributed Services in Mobile Ad Hoc Networks. Master's thesis, ETH Zurich, Computer Engineering and Networks Laboratory, 31st August 2005. MA-2005-14.

[51] Nathan Sheldon, Eric Girard, Seth Borg, Mark Claypool, and Emmanuel Agu. The Effect of Latency on User Performance in Warcraft III. In *NETGAMES '03: Proceedings Workshop on Network and System Support for Games*, May 2003.

[52] Tom Beigbeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. The Effects of Loss and Latency on User Performance in Unreal Tournament 2003. In *Proceedings Workshop on Network and System Support for Games*, pages 144–151, August 2004.

[53] Lothar Pantel and Lars C. Wolf. On the Impact of Delay on Real-Time Multiplayer Games. In *NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 23–29, New York, NY, USA, 2002. ACM Press.

[54] Sudhir Aggarwal, Hemant Banavar, Amit Khandelwal, Sarit Mukherjee, and Sampath Rangarajan. Accuracy in Dead-Reckoning Based Distributed Multi-Player Games. In *SIGCOMM 2004 Workshops: Proceedings of ACM SIGCOMM 2004 workshops on NetGames '04*, pages 161–165, New York, NY, USA, 2004. ACM Press.

[55] Marcel Busse, Bernd Lamparter, Martin Mauve, and Wolfgang Effelsberg. Lightweight QoS-Support for Networked Mobile Gaming. In *SIGCOMM 2004 Workshops: Proceedings of ACM SIGCOMM 2004 workshops on NetGames '04*, pages 85–92, New York, NY, USA, 2004. ACM Press.

[56] Grenville Armitage and Lawrence Stewart. Limitations of using Real-World, Public Servers to Estimate Jitter Tolerance of First Person Shooter Games. In *Proceedings ACM SIGCHI ACE2004*, pages 257–262, 2004.

[57] Matthias Dick, Oliver Wellnitz, and Lars Wolf. Analysis of Factors Affecting Players' Performance and Perception in Multiplayer Games. In *Proceedings*

Workshop on Network and System Support for Games, Hawthorne, USA, October 2005.

[58] Erik Nordström, Henrik Lundgren, and Björn Wiberg. AODV-UU Webpage. http://core.it.uu.se/AdHoc/AodvUUImpl.

[59] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[60] T. S. Rappaport. *Wireless Communications, Principles and Practice*. Prentice Hall International, 1996.

[61] Sam Leffler. Multiband Atheros Driver for WiFi (MADWIFI), 2005. http://sourceforge.net/projects/madwifi/.

[62] Bert Hubert, Gregory Maxwell, Remco van Mook, Martijn van Oosterhout, Paul B Schroeder, and Jasper Spaans. *Linux Advanced Routing and Traffic Control HOWTO*. Linux Documentation Project, 7 2002.

[63] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly, 2005. ISBN=0596005903.

[64] R. Finlayson, T. Mann, J.C. Mogul, and M. Theimer. Reverse Address Resolution Protocol. RFC 903 (Standard), June 1984.

[65] J. Bardwell. Converting Signal Strength Percentage to dBm Values, November 2002.
http://web.archive.org/web/20040405102601/http://www.wildpackets.com/-elements/whitepapers/Converting_Signal_Strength.pdf.

[66] David A. Maltz. The CMU Monarch Project's Wireless and Mobility Extensions to NS-2, August 1999.
http://www.monarch.cs.cmu.edu/.

[67] J. Broch, D. A. Maltz, D. B. Johnson, Y-C. Hu, and J. Jetcheva. A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. In *Proceedings of ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, pages 85–97, October 1998.

[68] Q. Zheng, X. Hong, and S. Ray. Recent Advances in Mobility Modeling for Mobile Ad Hoc Network Research. In *ACM-SE 42: Proceedings of the 42nd*

annual Southeast regional conference, pages 70–75, New York, NY, USA, 2004. ACM Press.

[69] T. Camp, J. Boleng, and V. Davies. A Survey of Mobility Models for Ad Hoc Network Research. *Wireless Communications & Mobile Computing (WCMC): Special issue on Mobile Ad Hoc Networking: Research, Trends and Applications*, 2(5):483–502, 2002.

[70] P. Ruiz and F. Ros. UM-OLSR Webpage. http://ants.dif.um.es/masimum/.

[71] Samir Ranjan Das, Charles E. Perkins, and Elizabeth M. Belding-Royer. Performance Comparison of Two On-demand Routing Protocols for Ad Hoc Networks. In *INFOCOM*, pages 3–12, 2000.

[72] M. Natkaniec and A. Pach. An Analysis of the Influence of the Threshold Parameter on the IEEE 802.11 Network Performance. In *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC)*, volume 2, pages 819–823, Chicago, IL, USA, September 2000.

[73] Information Sciences Institute ISI. Nam: Network Animator, July 2003. http://www.isi.edu/nsnam/nam/.