Universität Karlsruhe (TH)
Research University · founded 1825

Accuracy Enhancements of the 802.11 Model and EDCA QoS Extensions in ns-3

Diploma Thesis at the Institute of Telematics
Prof. Dr. Hannes Hartenstein
Faculty of Computer Science
University of Karlsruhe (TH)

by

cand. inform.
Timo Bingmann

Supervisors:

Prof. Dr. Hannes Hartenstein
Dr. Jérôme Härri
Dipl.-Inform. Jens Mittag

Date of Registration: 30th of October 2008
Date of Completion: 29th of April 2009

Institute of Telematics
Decentralized Systems and Network Services Research Group

I hereby declare that this thesis is a work of my own, and that only sources cited in the bibliography have been used.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, 29th of April 2009

Acknowledgments

I would like to express my thankfulness to people, who supported me in preparation and creation of this thesis. Without their suggestions and contributions it would have been less enjoyable and successful.

First I would like to express my gratitude to Jens Mittag and Felix Schmidt-Eisenlohr from the DSN. Jens kept me on the right track and gave technical consultations leading to new ideas that would never have occurred to me. Felix took the time to competently answer some very detailed and uncomfortable inquires about ns-2.

Next I want to thank the ns-3 developers Mathieu Lacage and Tom Henderson for resisting and actually, even promptly answering my initially clueless emails. Their continuous leadership and contributions make ns-3 such a high quality simulation tool.

Extra thanks goes to my friends for helping me cope with the stress generated by this thesis. Furthermore I want to especially thank my family for support too often taken for granted, and for proof-reading a thesis filled with puzzling technical terms.

Lastly I want to thank the University Rechenzentrum (computing center) for providing DUKATH, the campus wireless LAN network. Over many years, my Internet access via DUKATH showed utterly unintelligible variations in signal quality and speed. These oddities triggered and continue to sustain my interest in 802.11 wireless technology and signal propagation effects.

Zusammenfassung

Simulation ist ein unverzichtbares Werkzeug zur Entwicklung neuer Applikationen für drahtlose Netzwerke wie mobile Ad-hoc- oder Fahrzeugnetze. ns-3 ist der designierte Nachfolger von ns-2, dem sehr bekannten und in der Forschung aktuell am häufigsten eingesetzten open-source Netzwerksimulator. Mit ns-3 wurde ein vollständig neu konstruiertes Simulatordesign entwickelt, in dem Erfahrungen von ns-2 eingehen und modernste Softwareentwicklungsmethoden und Programmiersprachen eingesetzt werden.

In dieser Arbeit wurde die vorhandene 802.11 Implementierung des ns-3 mit dem verbesserten ns-2 Modell der DSN Forschungsgruppe vergleichen und die neuesten Erweiterungen der Kanalmodellierung und physischen Schicht in ns-3 übertragen. Die daraus resultierenden neuen Modelle verhalten sich identisch zu den ns-2 Vorgaben und wurden durch Wiederverwendung existierender Komponenten nahtlos in das bestehende ns-3 Design integriert. Die vom DSN beigetragenen log-distance und Nakagami-m Wellenausbreitungsmodelle wurden restrukturiert und in ns-3 als separate Funkfelddämpfung- und Interferenzschwundmodelle portiert. Entsprechend wurde das erweiterte auf einem SINR Empfangskriterium basierende PHY-Schichtmodell von ns-2 nach ns-3 übertragen. Diese neue PHY Implementierung modelliert den „Capture Effect“ und liefert identische Werte wie das entsprechende ns-2 Modell. Sowohl das neue SINR als auch das vorhandene BER/PER Empfangskriterium werden im ersten Teil dieser Arbeit detailliert dargelegt.

Ein weiterer Schwerpunkt ist die Eingliederung der EDCA Erweiterungen in ns-3, durch die Experimente mit relativem, dezentralisiertem QoS in drahtlosen Ad-hoc-Netzen möglich werden. Die entwickelte EDCA Implementierung baut auf dem bestehenden DCF Design auf, welches in dieser Arbeit eingehend untersucht wird. Um den 802.11e Standard zu erfüllen, wurden entsprechende Basisänderungen vorgenommen und darüber hinaus TXOP Zeitschranken und „burst“ Paketfolgen implementiert. Die EDCA Erweiterungen wurden mit zwei Experimenten überprüft, wobei in der ersten Untersuchung simulativ gemessene maximale Durchsatzwerte mit rechnerisch ermittelten Referenzwerten verglichen wurden und im zweiten Szenario die von EDCA ermöglichte relative QoS an Hand einer steigenden Zahl von Paketflüssen unterschiedlicher Priorität besonders herausgestellt wurde.

Die in dieser Arbeit implementierten Modelle ermöglichen einen aussagekräftigen Geschwindigkeitsvergleich von Wireless LAN Experimenten in ns-2 und ns-3. In beiden Simulatoren wurde ein abstraktes Autobahnszenario identisch nachgebildet mit exakt denselben experimentellen Ergebnissen. Zur Messung wurden verschiedene Compiler, Optimierungsstufen und Linkertechniken angewandt und deren Auswirkungen auf die Ausführungszeit untersucht und begründet. Mögliche zukünftige Ansätze zur weiteren Verkürzung der Simulationslaufzeit werden kurz umrissen. Die Geschwindigkeitsmessungen ergaben erhebliche Laufzeitersparnisse von bis zu 59% mit ns-3 gegenüber identische ns-2 Simulationen.

Abstract

For studying wireless networks like mobile or vehicular ad-hoc networks, simulation is an indispensable tool. ns-3 is the designated successor of ns-2, the well-known open-source packet-level simulator widely used in network research. ns-3 is a new modern network simulator designed from scratch blending state-of-the-art software engineering methods with experiences gained from ns-2.

For this thesis, the 802.11 implementation in ns-3 was compared to the ns-2 models improved by the DSN research group in 2007. These contributed physical and channel layer enhancements were transferred to ns-3. The implemented model code yields equal results as in ns-2, while adapting applicable code modules and integrating cleanly in the existing ns-3 model design. The log-distance and Nakagami-m propagation loss models, as added to ns-2 by the DSN, were restructured and ported to ns-3 as separate path loss and fast fading components. Likewise the improved ns-2 PHY layer model, based on a SINR reception criterion, was transferred from ns-2 to ns-3. This new PHY implementation features the frame capture effect and is verified to produce equal results as the corresponding ns-2 code. Both the new SINR and existing BER/PER reception criteria are thoroughly discussed in the first part of this thesis.

The second focus is on EDCA extensions, which were added to ns-3 and enable experiments with relative, decentralized QoS in wireless ad-hoc networks. The developed EDCA implementation builds on the existing DCF design, which was minutely examined and is reviewed in this thesis. Necessary modifications thereof and further extensions as TXOP limits and burst sequences were added to fulfill 802.11e specifications. The added code is verified using maximum throughput experiments, which are compared against analytically determined reference results. Furthermore, relative QoS as provided by EDCA is spotlighted in a second simulation scenario with an increasing number of differently prioritized traffic streams.

Use of the 802.11 models added in this thesis allow a convincing speed comparison of wireless simulations in ns-2 and ns-3. For this purpose a complex abstract highway scenario was designed and identical experiments were created with both simulators. These were tested to produce exactly equal results by using components added for this thesis. Different compilers, optimization levels and build options were tested and their effects on simulation execution time are explained. Possible future work on ns-3 to further reduce wireless simulation run time is shortly sketched. In the speed test, execution time of ns-3 showed a reduction of up to 59% over identical ns-2 simulations.

Table of Contents

List of Figures

List of Tables

List of Acronyms

AC	access category. 20, 21
ACI	access category index. 20
ACK	acknowledgment. 17
ACM	admission control mandatory. 21, 23
AIFS	arbitration interframe space. 16, 20, 21, 76, 78
AIFSN	arbitration interframe space number. 20, 78
AP	access point. 6, 13, 18
API	application programming interface. 30, 32, 34
AWGN	additive white Gaussian noise. 36, 51, 52
BER	bit error rate. 60–62, 65
BPSK	binary phase shift keying. 11, 61, 62
BSS	basic service set. 13
CBR	constant bit rate. 34
CCK	complementary code keying. 6
CEPT	Conference of Postal and Telecommunication Administrations. 7
CFP	contention-free period. 18, 19, 23
CP	contention period. 18, 19
CSMA/CA	carrier sense multiple access with collision avoidance. 14
CSMA/CD	carrier sense multiple access with collision detection. 14
CTS	clear to send. 15
CW	contention window. 16, 17, 20, 21
DCF	distributed coordination function. 16–18, 75, 76
DFS	dynamic frequency selection. 6
DIFS	DCF interframe space. 15–17
DS	distribution system. 35
DSN	Decentralized Systems and Network Services Research Group. 26, 45
DSRC	Dedicated Short Range Communication. 8
DSSS	direct sequence spread spectrum. 6, 7
EDCA	enhanced distributed channel access. 6, 19–21, 23, 75, 78–80, 82–84
EDCAF	enhanced distributed channel access function. 20, 21, 75, 76, 78
EIFS	extended interframe space. 26
ERP	extended rate PHYs. 6
FCC	Federal Communications Commission. 7, 8

FHSS frequency hopping spread spectrum. 6, 7

GOT Global Offset Table. 92

HC hybrid coordinator. 23, 24
HCCA HCF controlled channel access. 6, 19, 23, 24
HCF hybrid coordination function. 6, 19, 20
HR/DSSS high rate direct sequence spread spectrum. 6, 7

IBSS independent basic service set. 13, 21
IEEE Institute of Electrical and Electronics Engineers. 5
IFS interframe space. 15, 16
ISM industrial, scientific and medical. 6, 8
ITU International Telecommunication Union. 7

LAN local area network. 5, 7
LL link layer. 88
LLC logical link control. 5, 35

MAC medium access control. 13
MANET mobile ad-hoc network. 75
MIMO multiple input and multiple output. 6, 96
MLME MAC sublayer management entity. 6
MPDU MAC protocol data unit. 81
MPI message passing interface. 30, 93
MSDU MAC service data unit. 82

NAV network allocation vector. 14, 15

OFDM orthogonal frequency division multiplexing. 6–8
OLSR Optimized Link State Routing. 34

PC point coordinator. 18, 19
PCF point coordination function. 18, 19
PER packet error rate. 60, 62, 65, 66
PHY physical layer. 7, 51
PIC position-independent code. 92
PIFS PCF interframe space. 15, 16, 18, 19, 23
PLCP physical layer convergence procedure. 6, 9
PLME physical layer management entity. 6
PLT Procedure Linkage Table. 92
PMD physical medium dependent. 6
PSK phase shift keying. 11

QAM quadrature amplitude modulation. 11, 62
QoS quality of service. 6, 16, 19, 20, 23
QPSK quadrature phase shift keying. 11, 61, 62

RTS request to send. 15
RTT round trip time. 34

SDL Specification and Description Language. 57
SIFS short interframe space. 15
SINR signal to interference and noise ratio. 52, 54–56
SME station management entity. 6
SNAP subnetwork access protocol. 35
STA station. 6, 12
STL standard template library. 29

TBTT target beacon transmission time. 19
TID traffic identifier. 78
TPC transmit power control. 6
TS traffic stream. 23, 24
TSID traffic stream identifier. 23
TSPEC traffic specification. 23
TXOP transmission opportunity. 19, 21, 23, 78, 79

U-NII unlicensed national information infrastructure. 6, 8
UML Unified Modeling Language. 47, 57, 75
UP user priority. 20

VANET vehicular ad-hoc network. 26, 75

WAVE wireless access in vehicular environments. 6, 12
WEP wired equivalent privacy. 6
WLAN wireless LAN. 5
WPA Wi-Fi protected access. 6

Chapter 1

Introduction

1.1 Motivation

Wireless communication in computer networks enables novel and fascinating applications and increasingly pervades everyday life. The IEEE 802.11 standard is the foundation of this omnipresent technology and future extensions will continue its success. These extensions allow operation of wireless devices in new environments and open fields for innovative research.

For studying wireless networks simulation is an indispensable tool. Design and development of network protocols and components can be done in simulation with less cost and time. Particularly for large-scale mobile or vehicular networks, deployment and management of new technology is very expensive, whereas flexible simulation allows rapid prototyping and evaluation of new protocols and ideas.

ns-3 is the designated successor of ns-2, which is currently the most popular open-source packet-level network simulator used in research. The new ns-3 project aims at developing a network simulator aligned with modern network research. It is written from scratch with a state-of-the-art design, which utilizes up-to-date software engineering methods and incorporates lessons learned from ns-2's aged code and development history.

First goal of this thesis is to compare the existing 802.11 model in ns-3 with the one contributed to ns-2 by the Decentralized Systems and Network Services Research Group (DSN). Enhancements made to ns-2's model will be transferred to ns-3, while reusing available components and embedding cleanly in the existing structure. Goal is to have both models produce equal results or show accountable differences.

To achieve this, an in-depth comparison of both simulators' propagation models and reception criteria is necessary. The Nakagami-m fast fading propagation model, as contributed by the DSN to ns-2, will be restructured and ported to ns-3. Both SINR threshold reception criterion and BER/PER calculations will be discussed and their different approaches highlighted. Cumulative noise and frame capture effects are relevant for both criteria and will be integral parts of these discussions.

While the first thesis goal focuses on models of medium and physical layers, the second goal addresses QoS in the MAC layer. In the beginning wireless LAN had a free-ride: it was a cool new technology operating in far-spread areas with little cooperation troubles. No one expected any reliable service in densely operated areas. However, increasingly wireless LAN is used for critical applications with high costs of failure, and dependable services must be delivered.

The 802.11e amendment addresses this need by adding QoS features, of which the EDCA extensions will be implemented in ns-3 as the second part of this thesis. These extensions will provide relative QoS using a distributed medium access coordination algorithm, which is based on DCF. To implement EDCA in ns-3, the existing, well designed DCF implementation will be modified and extended to provide multiple different priorities. Moreover, EDCA-TXOP limits and burst sequences will be added to the medium access control code.

Programming of simulation code requires a high level of accuracy and expertise, due to its own complexity and complexity of the entities modeled. Good and verified code is a necessity, because errors in simulation code often go unnoticed for long periods of time. Other researchers expect to be able to use the models without needing to check their implementations. Moreover, lower simulation layers are often used without

detailed knowledge of their workings. For this reason, all added components will be thoroughly tested with specially designed scenarios, which pinpoint crucial parts of the code. The contributed propagation loss models and reception criteria including frame capture will be tested using two reduced scenarios containing only two or three communication nodes. Maximum throughput of both the new EDCA and the existing DCF implementations will be checked against analytically determined values, and thereby verifying the tightest allowed sequence of frames and waiting intervals. The effects of relative QoS as provided by EDCA will be showcased in a simple scenario.

Finally, a speed comparison of ns-2 and ns-3 will be performed with an abstract highway scenario implemented in both simulators. Equal behavior and results for all speed test experiments can be achieved despite the disparate simulation platforms. Remaining differences between both simulators will be highlighted. Speed test results with the GNU and Intel C++ compilers with and without static linking will be put into relation and their causes discussed.

1.2 Contributions

The main contributions of this thesis can be grouped into two categories: comparison and augmenting of PHY and channel models, and enabling EDCA QoS simulations in ns-3.

Behavior of the PHY and channel models of ns-2 and ns-3 were compared and contributions to ns-2 by the DSN were transferred to ns-3. In this context the SINR-based reception criterion including frame capture effect was successfully integrated into existing ns-3 code. Furthermore, the Nakagami-m propagation loss model was added to ns-3 in form of two separate classes representing the path loss and fast fading components of the ns-2 model. All ported features were verified using experiments in ns-2 and ns-3 and produce equal results. Both the new SINR reception criterion and the existing BER/PER criterion, as implemented in ns-3, were explained with great detail and their different approaches discussed.

Moreover, EDCA was implemented in ns-3 with special focus on ad-hoc networks. By applying out-of-band access category identifiers to packets relative QoS, as defined by 802.11e, can be simulated. Multiple queues and parallel medium access coordination with different priorities enables interesting new research with ns-3.

The speed test between ns-2 and ns-3, comparing identical experiment setups on both platforms, showed a simulation run time reduction of up to 59% by employing ns-3 with full optimization and static linking.

Part I

Related Work

Chapter 2

IEEE 802.11

Wireless computer communication has become a nearly ubiquitous technology. Low cost hardware and high data rates have popularized wireless data networks in the recent years. Information processing is not longer bound to stationary computer systems and many new and interesting applications have been enabled by mobile data processing.

Basis for this success is the IEEE 802.11 standard for wireless LAN (WLAN). It is unquestionably today's most successful standard for networking computers wirelessly. This standard first enabled the wireless connectivity we often take for granted. Its popularity is based on simplicity and robustness against failures, both comparable to wired networks. Operation in unlicensed radio bands allows wide-spread public use and easy adoption, while at the same time requires data transmission to be resilient against interference and thus provide robust connectivity. Distributed medium access control allows uncomplicated deployment and smooth operation of multiple networks in the same area.

The initial 802.11 standard was approved and published by the Institute of Electrical and Electronics Engineers (IEEE) in 1997. Much of the fundamental service definition including the basic MAC functions, like DCF and AP association rules, was already part of the initial standard, is still valid and in general use today. Other parts were subsequently extended in different supplements and amendments. These provide, for example, faster data transmission, QoS extensions and adaptation to specific locations and environments. Table 2.1 shows a selection of currently approved or proposed 802.11 amendments and includes a short description of their individual aims.

Until 802.11 was completed in 1997, there was no common standard for wireless networking devices. Each vendor had his own set of protocols, transmission modes and parameters. Some examples of pre-802.11 wireless technology are NCR's WaveLAN, Proxim's RangeLAN and Aironet/Cisco's ARLAN [4]. Incompatibility between these solutions prohibited wide-spread adoption and lead to definition of 802.11, which is the prevalent standard for medium range packet radio networking today.

2.1 802: The Big Picture

The IEEE 802.11 standard is part of the IEEE 802 standards family, which deals with local and metropolitan area networks. The 802 standards define services and protocols for the lower two layers of the ISO/OSI layer reference model, the data link and physical layer. The number 802 was simply the next available IEEE standard number at that time.

Among the members of this standard family are 802.3 for Ethernet, 802.5 for Token Ring, 802.11 for Wireless LAN and the upcoming 802.16 for WiMAX. There are also some less successful or retired standards like 802.4 for Token Bus and 802.8 for Fiber Optic networks.

The 802 family also includes a set of base standards like 802.1 for network bridging and management, 802.2 for logical link control (LLC) and 802.10 for interoperable LAN security mechanisms.

The 802.11 wireless standard defines a service point compatibly with 802 medium access control (MAC) requirements, the same as 802.3 wired Ethernet fulfills. It therefore provides an interface to the upper layers

Standard	Year	Description
802.11-1997	1997	Initial standard. 1 or 2 Mb/s using FHSS or DSSS in 2.45 GHz band.
802.11a	1999	Up to 54 Mb/s using OFDM in 5 GHz band.
802.11b	1999	Up to 11 Mb/s using HR/DSSS with CCK in 2.45 GHz band.
802.11h	2003	TPC and DFS for 5 GHz band in Europe.
802.11g	2003	ERP using OFDM, CCK, DSSS and others for up to 54 Mb/s in 2.45 GHz band.
802.11i	2004	Security mechanisms WPA and WPA2 replacing broken WEP encryption.
802.11e	2005	QoS extensions: HCF with EDCA and HCCA.
802.11-2007	2007	Revised standard incorporating all preceding amendments.
802.11k	2008	Radio quality measurement and network information.
802.11r	2008	Fast handoff for transition between BSS.
802.11p	in work for 2009	WAVE – Wireless Access in Vehicular Environments.
802.11n	in work for 2009	MIMO, channel bonding and frame aggregation for higher throughput.
802.11s	in work	Mesh networking for infrastructure and ad-hoc, multi-hop connectivity.

Table 2.1: Selection of 802.11 standards and amendments

that is equivalent to that of other network technologies and ultimately yields a user experience very similar to wired networks.

2.2 Outline of 802 and 802.11 Layers

This section gives a very broad overview of the components defined in the 802.11 standard. Figure 2.1 illustrates these sublayers and how they build up the 802.11 architecture.

The medium access control (MAC) layer provides a service entry point for higher entities to exchange message packets between addressable wireless stations. To support this service the MAC utilizes the underlying physical layer (PHY) services provided by the pair of PLCP and PMD. The MAC layer defines a transmission frame format and different data, control and management frames for exchange between wireless stations (STAs) and access points (APs). To manage frame exchanges on the shared medium, the MAC defines the coordination functions DCF, PCF and HCF, which regulate how stations may access the shared wireless medium. Conceptually the MAC layer contains a sublayer called MAC sublayer management entity (MLME), which provides operational features like AP scanning and association, encryption set up and configuration.

The PHY layer is split up into two sublayers. The physical medium dependent (PMD) layer defines how a specific medium is accessed by the transceiver, while the physical layer convergence procedure (PLCP) provides adaptation to a common PHY interface. The 802.11 standard defines different PMD/PLCP pairs for communication in the 2.45 GHz ISM radio band, the 5 GHz U-NII radio band and using infrared light. Integrated into the PHY layer is another management layer, the physical layer management entity (PLME), which exports configurable aspects of the used transmission modes and other information.

Both management layers are accessed by the station management entity (SME) cross layer, which is not explicitly defined by the 802.11 standard. Its purpose is to provide an interface for higher level system management. Possible services of the SME are configuration of 802.11 components by the user, gathering of statistics or forwarding of user requests.

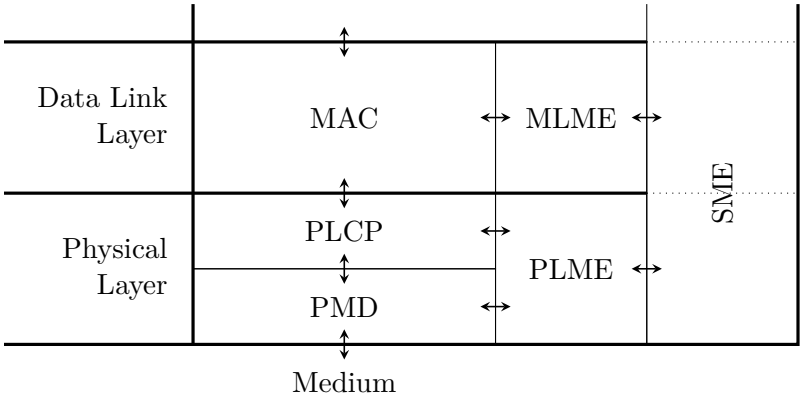


Figure 2.1: General 802.11 architecture as ISO/OSI layer diagram (adapted from [12, figure 5-10])

2.3 PHY Layers

The 802.11 standard specifies different PHY layers for transmitting data over different media. All PHYs have a common interface to the higher MAC layer, which coordinates packet transfer on the medium. Each PHY layer definition contains all details of the low-level aspects of wireless data transfer: how bits are encoded, modulated and how the wireless medium is multiplexed. Regulatory aspects of the 2.45 GHz, 5 GHz and 5.9 GHz radio bands used by the different PHY layers are discussed in the following section 2.3.1.

The original standard from 1997 defines two PHYs for the 2.45 GHz ISM radio band. Both PHYs operate at two data rates: 1 Mb/s or 2 Mb/s. Two different *spread spectrum* techniques are used to encounter the challenge of transmitting in unlicensed bands: frequency hopping spread spectrum (FHSS) and direct sequence spread spectrum (DSSS). The original standard also contained a PHY for infrared light, which was not as successful as the radio PHYs.

In 1999, the 802.11a amendment was approved by the IEEE and it introduced new specifications for up to 54 Mb/s in the 5 GHz U-NII bands. These high data rates were possible by employing orthogonal frequency division multiplexing (OFDM).

Later in the same year, the 802.11b amendment added higher data rates of up to 11 Mb/s in the 2.54 GHz band. These extend the original spread spectrum technique by using complementary code keying (CCK) for high rate DSSS (HR/DSSS).

The speed increase of 802.11a and 802.11b was a major break-through at that time and made wireless LAN a serious contender to wired connections for many scenarios.

By adapting the 802.11a PHY, operating with OFDM, for the 2.54 GHz radio band, the higher transmission rates of up to 54 Mb/s were made possible in all of Europe. This adaption, called extended rate PHYs (ERP), is focus of the 802.11g amendment.

With the future 802.11p, the same OFDM transmission schemes are being made available in the 5.9 GHz band.

2.3.1 The ISM and U-NII Bands

Wireless devices use parts of the radio spectrum as communication medium by sending electromagnetic signals. The used radio frequency band is the key resource to wireless communication. However, since the radio spectrum is a limited resource, most governments regulate its use by mandating frequency allocation and requiring operational limits of equipment. Different national regulatory bodies are commissioned for individual jurisdictions, like the Federal Communications Commission (FCC) for the United State, the Conference of Postal and Telecommunication Administrations (CEPT) for Europe and the Bundesnetzagentur for Germany. For purposes of harmonization and common equipment almost all countries follow the radio regulations issued by the International Telecommunication Union (ITU), nevertheless regional difference exist and must be dealt with.

Allocation No. 5.138	Allocation No. 5.150
6.765 – 6.795 MHz	13.553 – 13.567 MHz
433.05 – 434.79 MHz	26.957 – 27.283 MHz
61.0 – 61.5 GHz	40.66 – 40.70 MHz
122 – 123 GHz	902 – 928 MHz
244 – 246 GHz	2.400 – 2.500 GHz
	5.725 – 5.875 GHz
	24.00 – 24.25 GHz

(a) ISM bands defined by the ITU [36]

Frequency Range	U-NII
5.15 – 5.25 GHz	low band
5.25 – 5.35 GHz	middle band
5.470 – 5.725 GHz	world-wide band
5.725 – 5.825 GHz	high band

(b) U-NII bands defined by the FCC

Table 2.2: ISM and U-NII bands

Use of most portions of the regulated radio spectrum requires a broadcast license from the national regulatory body. Licenses are granted often for only a specific geographical area and usually contain restraints on broadcast power, modulation and antennas. Sales of these licenses are increasingly offered via public auctions and achieve high prices due to their economic value.

The industrial, scientific and medical (ISM) bands (see table 2.2(a)) are parts of the electromagnetic spectrum designated by the ITU for "industrial, scientific and medical applications", meaning non-communication applications. The maybe (still) most common ISM-band device is a microwave oven operating at 2.45 GHz, because radiation at that frequency is particularly well absorbed by water. The ISM is an unlicensed band and therefore operating radio devices (like microwave ovens) in those bands does not requires an expensive license. For communication applications this advantage, however, is a double-edged sword since unlicensed, secondary users must cope with interference from primary users. The whole idea behind regulation is to keep interference from other equipment at a minimum, like traditional AM/FM-radio broadcasting stations. To mitigate the interference problem, devices that use the 2.45 GHz ISM-band are required by regulation to limit power to 30 dBm (before the antenna) in the USA and to 20 dBm (at the antenna) in most of Europe.

Major drawback of the 2.45 GHz ISM band is that there are already many applications using this range. Bluetooth, various cordless mice and other devices also operate in these bands.

The second set of radio bands used by 802.11 are the four unlicensed national information infrastructure (U-NII) bands listed in table 2.2(b). These FCC band allocations were originally only usable within the US. Equipment operating in the U-NII bands is subject to specific power limitations of 40 mW, 200 mW and 800 mW respectively. In 2005, use for these blocks has been re-regulated for Europe and operation of 802.11 devices in the bands 5.15 – 5.35 GHz and 5.470 – 5.725 GHz is now allowed in all European countries.

In 1999, the FCC allocated the 5.850 – 5.925 GHz frequency range, in short called the 5.9 GHz band, for a variety of Dedicated Short Range Communication (DSRC) uses in the transportation system. Envisioned applications include increased traffic safety, traffic monitoring with congestion detection and avoidance, and traffic light control and possible preemption by emergency vehicles. An example for a system operating in the 5.9 GHz band, that is already in wide use, is the drive-by truck toll collection system in large parts of Europe. The upcoming 802.11p amendment adapts the 802.11 standard for the 5.9 GHz band.

2.3.2 802.11a – OFDM

Since the 802.11a communication specifications are of special interest in this thesis, they are reviewed and explained in great detail in the following subsections. In section 6.6 error rate calculations for the following coding techniques are discussed.

802.11a supports eight different transmission modes. The different schemes' data rates depend on how the used 5 GHz band is partitioned into channels. Table 2.3 shows the data rates in Mb/s for 20, 10 and 5 MHz channels. Obviously the data rates for 20 MHz channels is twice as large as for 10 MHz channels, but then e.g. the U-NII low range 5.15 – 5.25 GHz can be partitioned into only five channels.

To cope with interference in the unlicensed bands, 802.11a employs OFDM. For OFDM the used frequency range is split up into 48 subcarriers and each subcarrier transmits 1, 2, 4 or 6 coded bits, depending on the

Modulation	Coding rate (R)	Coded bits per subcarrier N_{BPSC}	Coded bits per OFDM symbol N_{CBPS}	Data bits per OFDM symbol N_{DBPS}	Data rate (Mb/s) for 20 MHz channels	Data rate (Mb/s) for 10 MHz channels	Data rate (Mb/s) for 5 MHz channels
BPSK	1/2	1	48	24	6	3	1.5
BPSK	3/4	1	48	36	9	4.5	2.25
QPSK	1/2	2	96	48	12	6	3
QPSK	3/4	2	96	72	18	9	4.5
16-QAM	1/2	4	192	96	24	12	6
16-QAM	3/4	4	192	144	36	18	9
64-QAM	2/3	6	288	192	48	24	12
64-QAM	3/4	6	288	216	54	27	13.5

Table 2.3: Data rates and modulation parameters for 802.11a (from [12])

keying modulation used (see table 2.3). Furthermore, convolutional encoding is used to make transmission more robust against bit errors by adding redundancy. And finally, because data bits tend to contain long sequences of zeros or ones, the data bits must be scrambled prior to encoding.

This series of transformations is detailed step by step in the following sections. At the beginning there is a sequence of plain data bits waiting for transfer. Furthermore, the selected transfer scheme from table 2.3 is known.

Payload Whitening

From viewpoint of the PHY the sequence of data bits is the payload of the frame. Since this payload is bound to contain long sequences of ones and zeros, it must be whitened to eliminate an unwanted direct current.

Before scrambling the data bits, they are wrapped into the PLCP *data* frame portion illustrated in figure 2.2. The payload is prefixed with a 16 bits *service* field and suffixed with 6 *tail* bits, all initialized with zeros. Furthermore the whole byte sequence is padded with zeros to the length of a full OFDM symbol at the desired data rate, i.e. N_{DBPS} in table 2.3.

Then the complete *data* part of the frame is whitened by running the bit sequence through the linear feedback shift register generated by $x^7 + x^4 + 1$. The shift register is shown in figure 2.3 and is used for both scrambling and descrambling. For scrambling the sender initializes the registers with pseudo-random bits.

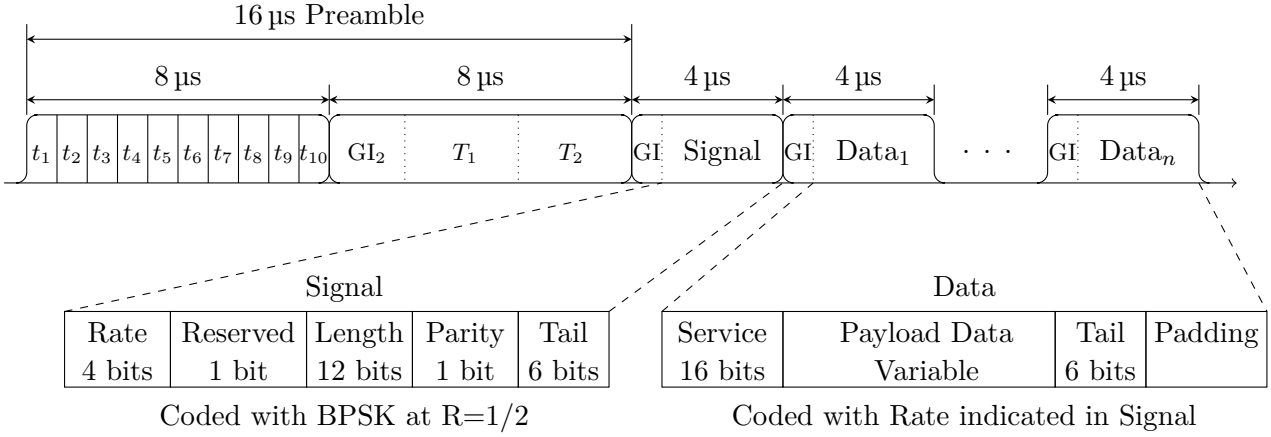


Figure 2.2: 802.11a frame with OFDM preamble and PLCP header using 20 MHz channels (based on [12])

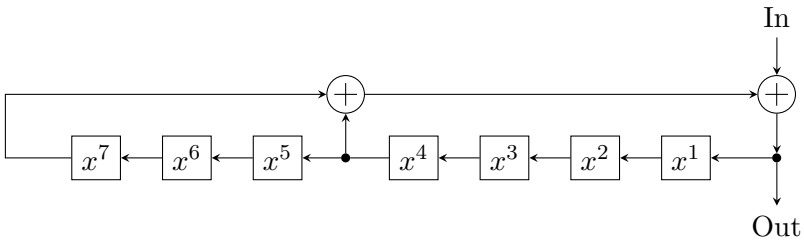


Figure 2.3: 802.11a scrambler as linear feedback shift register (adapted from [12, figure 17-7])

The receiver can derive the initial state from the first seven bits of the *service* field, because they were set to zero prior to scrambling.

Convolutional Encoding

The whitened *data* portion of the frame is then prefixed with the *signal* header. This frame header, shown in figure 2.2, contains the length of the payload data and the transfer rate employed.

To allow fast data transfer rates, the data portion may be modulated at any transmission mode from table 2.3 that is supported by both sender and receiver. The *signal* header, however, is always modulated separately with the lowest data rate, BPSK and $R = 1/2$, because it contains the *rate* field specifying the payload mode and must be readable at a common rate. The complete *signal* header fits into one OFDM symbol at the fixed data rate.

The two parts of a frame, header and whitened data portions, are encoded separately using a convolutional encoder with coding rate $R = 1/2$. Purpose of the convolutional encoding is to add redundancy, which is used for forward error correction during decoding.

The convolutional encoder uses the function generators $g_{\mathrm{A}} = [133]_8 = [1011011]_2$ and $g_{\mathrm{B}} = [171]_8 = [1111001]_2$ where $[\,]_8$ is the octal and $[\,]_2$ the binary representation. Figure 2.4 shows the encoder as a linear feedback shift register.

There are other possible representations of the convolutional encoder like a trellis diagram or a state machine. However, due to the $2^6 = 64$ possible register states, both representations are large, complex and not very useful. A state diagram of the encoder is shown in figure B.1 in the appendix.

For each input bit two output bits are calculated, yielding a coding rate of $R = 1/2$. The higher coding rates 3/4 and 2/3 are derived from the lower rate 1/2 by puncturing, that is exchanging and omitting some of the generated bits. For $R = 3/4$, 6 out of 18 encoded bits are removed and for $R = 2/3$ only 3 out of 12 are removed from the bit sequence.

The decoding process is efficiently done at the receiver with the Viterbi algorithm, which is described in section 6.6.2. For correct decoding, the six *tail* bits in both header and data parts of the PLCP frame are required to be zero prior to encoding. These must be reset to zero after the previous whitening phase.

So the output at this stage is a bit sequence inflated by the convolutional encoder to 2, 4/3 or 3/2 times the original size.

Bit Interleaving - Mapping to Subcarriers

Next step is to determine how the bit sequence is mapped to OFDM subcarriers.

802.11a always uses 48 data OFDM subcarriers, which are explained later. Each subcarrier transmits $N_{\mathrm{BPSC}} \in \{1, 2, 4, 6\}$ code bits, where N_{BPSC} depends on the transmission mode employed. Multiplying $N_{\mathrm{BPSC}} \cdot 48 = N_{\mathrm{CBPS}}$ directly yields the amount of code bits transmitted in one OFDM symbol. An OFDM symbol is the transmission state of all subcarriers in one fixed time interval.

So the whitened, convoluted bit sequence is divided into blocks of $N_{\mathrm{CBPS}} \in \{48, 96, 192, 288\}$ bits, which all go into one OFDM symbol. These bits are then mapped onto the 48 subcarriers, with N_{BPSC} code bits per

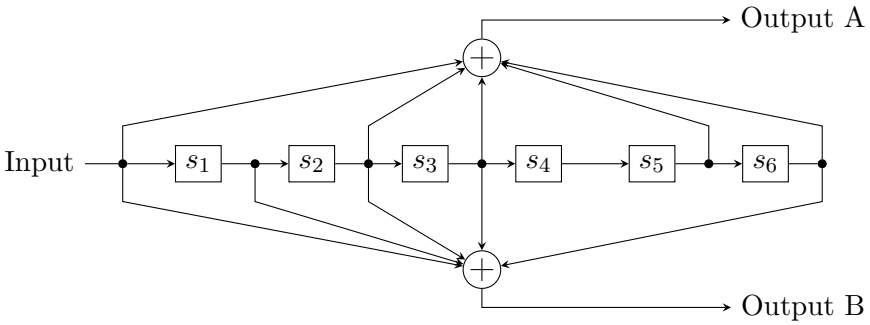


Figure 2.4: 802.11a convolutional encoder as linear shift register (adapted from [12, figure 17-8])

subcarrier. This mapping is defined as a two-step permutation defined in the standard [12, section 17.3.5.6]. Goal of the mapping is to interleave the individual bits so error detection and correction can work efficiently.

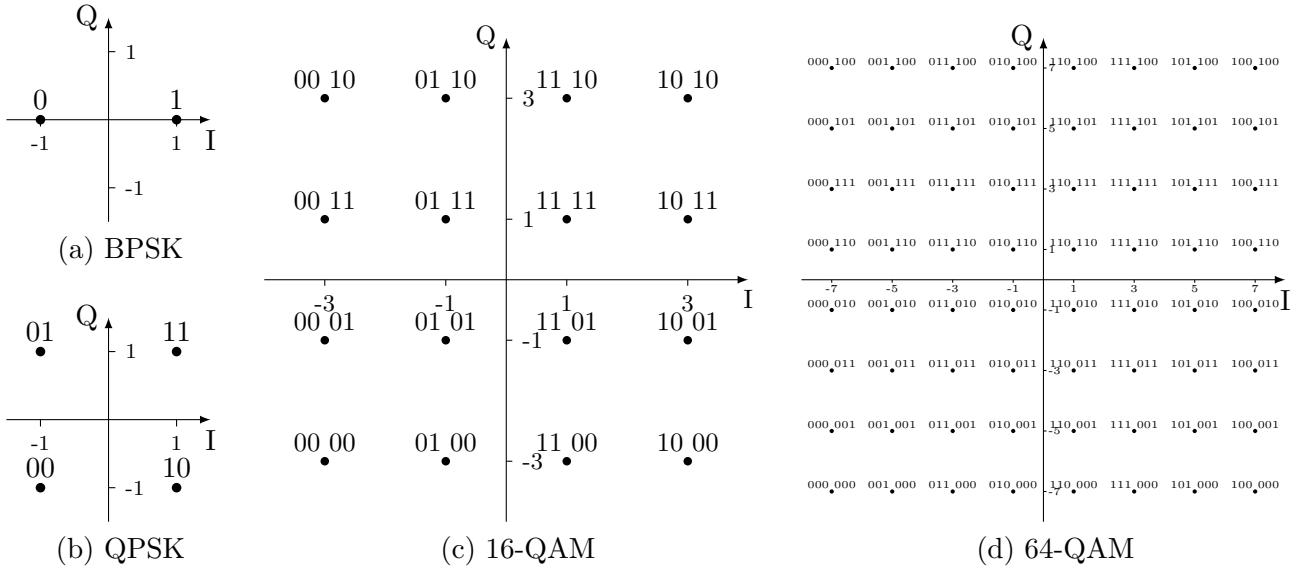
For example when transfering with 24 Mb/s in a 20 MHz channel, the input bit sequence is divided into blocks of 192 bits. These bits are already convolutional encoded and actually contain only 96 original data bits. Each OFDM subcarrier transmits 4 bits using a modulation described in the next section. So the 192 code bits are mapped onto the 48 subcarriers creating groups of 4 bits.

Signals on Subcarriers

The previous block interleaving stage has mapped $k := N_{\mathrm{BPSC}} \in \{1, 2, 4, 6\}$ bits into each of the 48 subcarriers. These k digit bits are represented as changes of a carrier signal, which is a sinoid function. The bit keyings employed are BPSK, QPSK, 16-QAM or 64-QAM, depending on the transmission mode desired.

Binary phase shift keying (BPSK) and quadrature phase shift keying (QPSK) belong to the family of phase shift keyings (PSKs). For PSK the bit value is encoded in the phase of the underlying carrier frequency. In BPSK a zero bit is represented by sending the carrier signal with a $-\frac{\pi}{2}$ and a one bit by sending it with $+\frac{\pi}{2}$ phase. For QPSK the two bits are encoded in four phase offsets of the carrier sinoid, for example with phase shifts of 0, $\frac{\pi}{2}$, π and $3\frac{\pi}{2}$.

For $k > 2$ the bits are encoded with quadrature amplitude modulation (QAM), which means that the information is in both the phase and amplitude of the data signal. These two orthogonal elements conveyed in the signal function are conveniently expressed as a complex number $(I + jQ)$ with j the imaginary unit. Figure 2.5 shows how each of the 2^k possible k-tuples of input bits are mapped onto a single complex number



(a) BPSK (b) QPSK (c) 16-QAM (d) 64-QAM

Figure 2.5: Keying constellation bit encoding (from [12, figure 17-10])

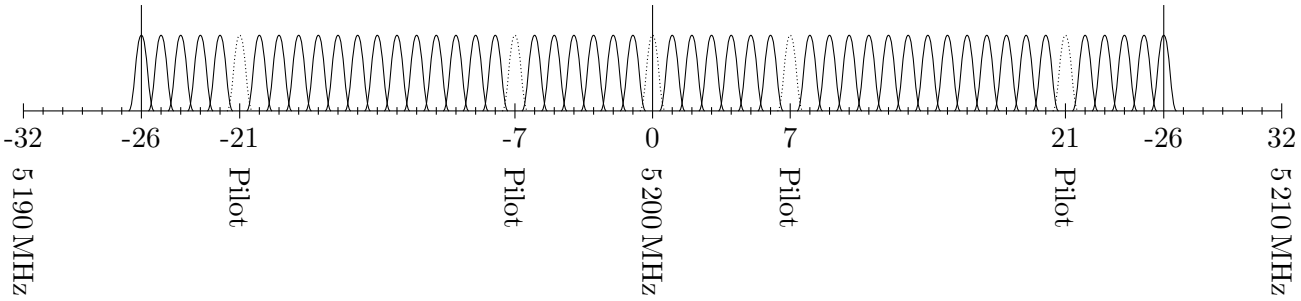


Figure 2.6: OFDM subcarrier layout

z. The norm $|z|$ corresponds to the amplitude, real and imaginary parts are sine/cosine components of the signal (without a normalizing factor).

As suggested by figures 2.5(a) and 2.5(b) BPSK and QPSK can also be represented by the QAM complex numbers. QPSK is equivalent to one variant of 4-QAM.

OFDM Subcarriers

To achieve high data rates, 802.11a uses orthogonal frequency division multiplexing (OFDM) in the used frequency range. OFDM is very popular for broadband communication and also used in ADSL for wired Internet access, DVB-T for digital television and 802.16 WiMAX.

The used frequency range is divided into 64 subcarriers. Figure 2.6 shows this partitioning for an example 20 MHz channel centered at 5.2 GHz. Of the 64 subcarriers only 48 are used for data transmission. To allow a receiver to correctly tune to the OFDM signal, 4 pilot carriers are added at fixed positions.

Each of the 64 subcarriers is modulated using BPSK, QPSK, 16-QAM or 64-QAM and carries 1, 2, 4 or 6 bits.

OFDM is a complex technique [35] and can cope with severe channel conditions like those expected in unlicensed bands. Insertion of a *guard interval* between symbols makes it possible to reduce inter-symbol interference. The guard interval contains a copy of the last portion of the symbol and this information can be used to reduce multi-path interference.

To allow a receiver to detect and tune to the OFDM signal, 802.11a defines a PLCP preamble used for synchronization [12, section 17.3.3]. It contains a fixed sequence of training symbols and has a duration of 16 µs as shown in figure 2.2.

802.11a enables very high data rates by employing the combination of convolutional encoding to protect data and OFDM for robust, highly efficient signal modulation.

2.3.3 802.11p – WAVE

The 802.11p amendment [14] is currently still a draft and aims to extend 802.11 to vehicular ad-hoc networks (VANETs). VANETs are different from the traditional applications of 802.11, because they are highly mobile and devices should operate in a dedicated, licensed frequency band to enable dependable services. 802.11p's goal is to provide wireless access in vehicular environments (WAVE).

To cope with highly mobile nodes, 802.11p contains extensions for communication between STA without establishing a basic service set (BSS), i.e. ad-hoc network set up or even AP association. This is called communication outside the context of a BSS (see section 2.4.1).

Beyond these communication context definitions, the 802.11p also amends the PHY defined in 802.11a for communication in the 5.850 – 5.925 GHz band. This requires only few changes to allowed frequency bands, because the base modulations scheme stays the same. See section 2.3.2 for details on 802.11a OFDM-based communication.

For the focus of this thesis, the most important specifications contained in the 802.11p amendment are the default EDCA parameters for use outside the context of a BSS. These parameters are specially geared for

vehicular communication and allow many interesting new applications. See section 2.4.7 for figures and a discussion of the parameter set in 802.11p.

2.4 MAC Layer

The purpose of the MAC layer is, as its name suggests, to coordinate access to the medium. However, the MAC layer in a single wireless device can, in effect, only control packets sent by itself. To prohibit another device from sending packets is not in its direct control.

Therefore, a group of independent wireless stations must collaboratively coordinate packet transfer by following a set of rules. This set of distributed rules is put down in the 802.11 standard and all equipment must follow them for smooth operation. At the core, wireless packet-based medium access is about agreeing on distributed algorithms or protocols that determine when and which station may send a packet.

How the packets themselves are actually sent on the medium is determined by the PHY layer below. The packet sending interface of the PHY has only broadcasting semantics: a transmitted packet is heard by all receivers within range.

The MAC layer, however, attempts to provide an interface to higher layers that is indistinguishable from other wired 802 layers like Ethernet or Token Ring. Due to the differences of wireless communication this aim can not (and should not) be completely fulfilled, but a common interface that enables wireless devices to be used side-by-side with other network devices on a system is possible.

This network interface has many attributes commonly known from Ethernet devices. For example, each wireless devices has a 48 bit MAC address, which is globally unique and allocated from the same pool as Ethernet devices. Using the MAC addressing schema, unicast packet delivery semantics are defined within a BSS just like with Ethernet.

However, the medium used by the PHY layers of wireless devices has characteristics fundamentally different from traditional wired media. The MAC layer must incorporate advanced functionality to deal with these differences. Some of the differences and mechanisms to overcome them are listed below.

- Communication using a wireless medium is significantly less reliable than wired communication. Therefore, the MAC uses positive acknowledgement for most packet transfers.
- Full connectivity between all stations cannot be assumed, e.g. due to obstacles. Packets are therefore relayed through a central access point.
- Radio propagation between two stations is generally asymmetric.
- Communication is unprotected from other signals on the medium. Particularly in unlicensed bands, the grade of quality of service (QoS) guarantees that can be provided are severely limited.

2.4.1 Communication Context

Wireless stations are grouped together to form a communication context called a basic service set (BSS). All data transfer between stations takes place within a BSS, except if the new upcoming 802.11p standard is used.

There are two kinds of BSS: independent BSS (IBSS) and infrastructure BSS with an AP.

Independent BSS are more commonly referred to as ad-hoc networks and communication within an IBSS is limited to direct passing of packets between stations.

An infrastructure BSS on the other hand contains an AP, which is connected to further networks usually via a wired network interface. The AP operates as a bridge between wired and wireless communication by forwarding packets as necessary. Even packets sent between wireless stations are relayed via the AP.

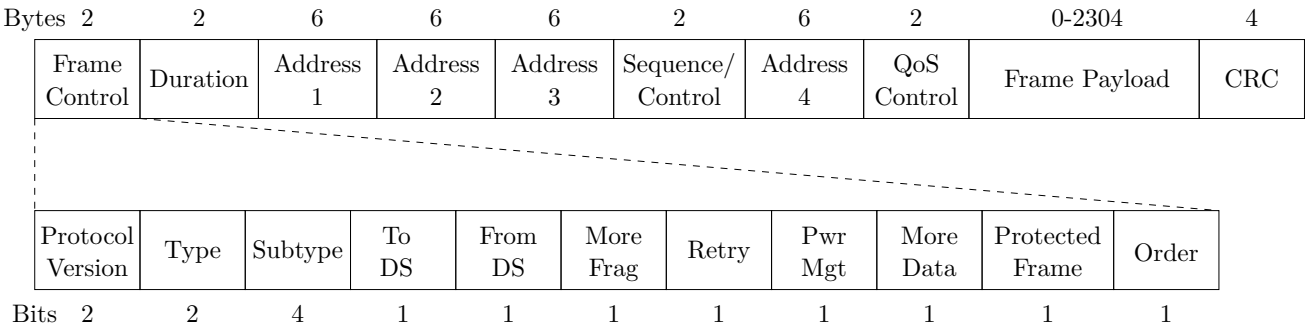


Figure 2.7: 802.11 MAC header (adapted from [12])

2.4.2 CSMA/CA using CS and NAV

802.11 uses a carrier sense multiple access with collision avoidance (CSMA/CA) scheme to coordinate access to the wireless medium. This mechanism is similar to carrier sense multiple access with collision detection (CSMA/CD), which is used by Ethernet. Both employ carrier (medium) sensing to detect multiple access to the medium.

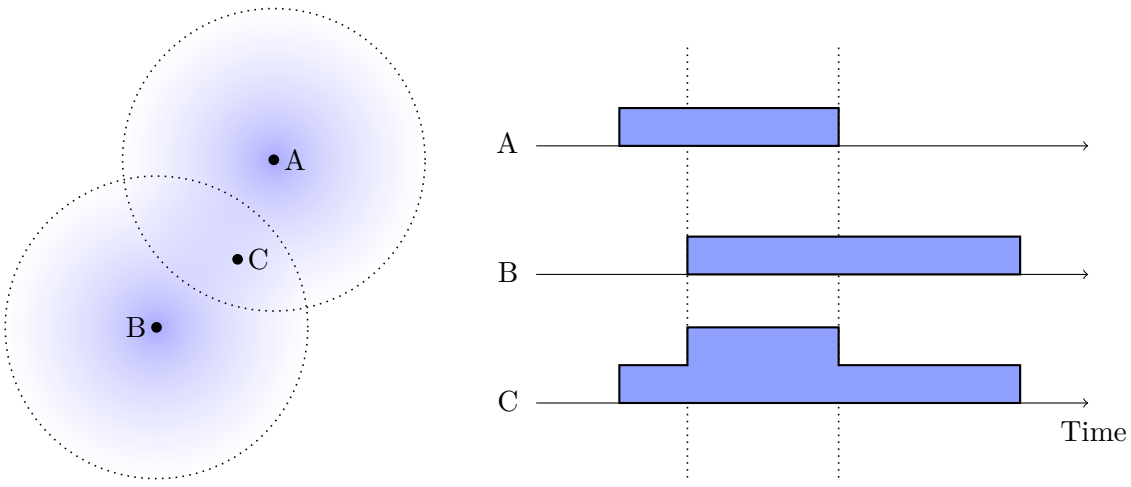
However, since it is difficult and thus expensive to built radio transceivers that can both listen and send simultaneously, 802.11 devices are not required to be full-duplex. With half-duplex transceivers, however, collision detection as employed by Ethernet while sending is not possible. Instead, collision avoidance is attempted through distributed coordination protocols.

There are two different carrier sensing mechanisms used by 802.11: physical carrier sense and virtual carrier sense.

A physical carrier sense indication is raised when another signal is detected on the medium. However, this signal detection is rather difficult because most wireless modulation schemes are only distinguishable from noise if the receiver is properly synchronized and has the capabilities required to decode the signal. The different PHYs in 802.11 require individual carrier sensing mechanisms and the standard leaves device manufacturers great flexibility in designing these mechanisms.

Virtual carrier sense is defined by the *duration* field in the MAC header, which is attached to every packet sent (see figure 2.7). The *duration* field contain a time value (in microseconds) for which the medium is reserved after the current packet. This medium reservation is called the network allocation vector (NAV).

As a first basic coordination rule, a station may not send a packet if either physical or virtual carrier sense indicates that the medium is busy. Using this rule, collisions are already avoided in most cases. However,



(a) Geometric layout (b) Interference sequence

Figure 2.8: Hidden terminal scenario

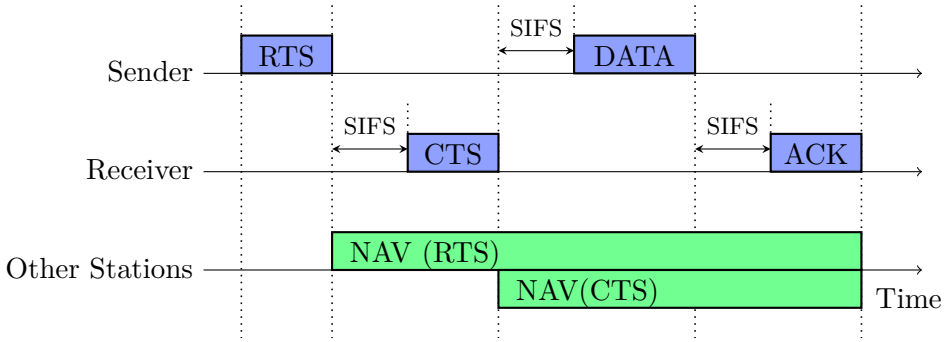


Figure 2.9: RTS/CTS/ACK frame protection sequence with NAV (adapted from [12, figure 9-7])

two important collision scenarios remain:

The first one occurs when two stations wish to send a packet and both detect the medium as idle. The issue is to determine when a station is allowed to start sending. 802.11 uses this to prioritize access to the medium for different transmission methods. Section 2.4.3 discusses these access priorities in detail.

And the second collision scenario, which is a special case of the first, is called the "hidden terminal" constellation. This happens when two stations are too far away from each other to sense their transmission and thus both detect the medium as idle. If both attempt to send to a third station located in between, their transmissions interfere and packets will be lost.

Figure 2.8 shows the hidden terminal scenario. If A and B attempt to send a packet to C at nearly the same time, both packets are superimposed at the receiver. Under most circumstances both packets are decoded incorrectly and are dropped. If however the receive power of one signal is small relative to the other, the stronger signal can still be correctly decoded. In section 6.4 these effects are discussed in more detail.

RTS/CTS Exchange

To avoid interference in the hidden terminal scenario, 802.11 uses a protection scheme. First a RTS control frame is sent, which must be answered by the destination station with a CTS frame. By setting the duration fields in both control frames, the two stations set up a NAV that prohibits other stations to send for a time interval that is used to transmit the data frame and return its acknowledgement. Figure 2.9 illustrates the frame exchange sequence.

Important in this scenario is that all other stations within range receive the RTS, the CTS or both frames. Thus all interferers in range of both stations are prohibited from sending during the data frame.

However, the RTS/CTS protection sequence comes at the cost of two additional frames. These introduce delay and reduce the maximum throughput by adding overhead. Hence the RTS/CTS sequence should only be used to large data frames. For large data frames the cost of collision is reduced, because the RTS frame is much shorter and thus medium recovery time is shorter. RTS/CTS does not reduce the probability of collision.

2.4.3 Interframe Space

The time a station has to wait after it detects the medium to be idle is used by 802.11 to prioritize access. These time intervals following the last detected frame are called interframe space (IFS). Figure 2.10 illustrates the IFS by showing the waiting intervals as required by different priority mechanisms.

In the original standard there are three different priority classes and three corresponding *interframe spaces*: SIFS, PIFS and DIFS.

The shortest interval is the short interframe space (SIFS). It is used in direct frame sequences like RTS/CTS shown in figure 2.9. The length of SIFS is determined by PHY characteristics: it is the maximum time allotted to the electronics of a station for decoding the incoming signal, processing the request and issuing an answer. Table 2.4 shows the actual values for all PHY.

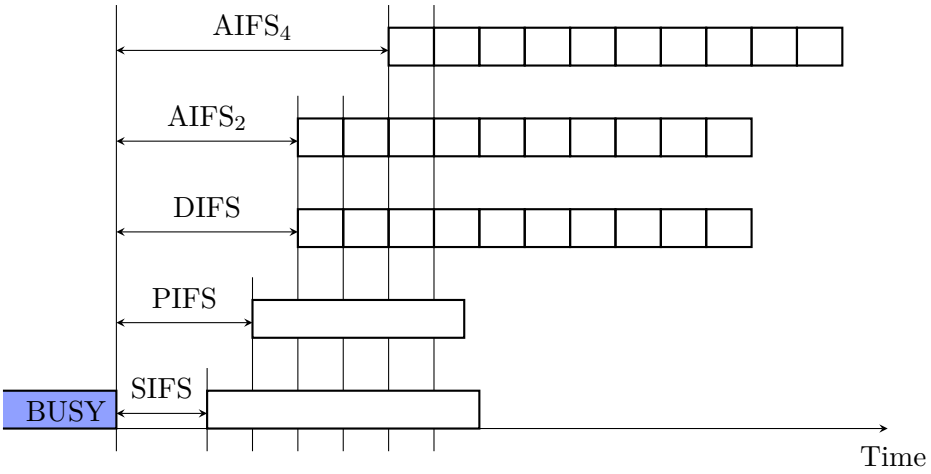


Figure 2.10: Interframe space (IFS)

The second longest interval is PCF interframe space (PIFS) used, as its name says, by the point coordination function (PCF) operation mode. PCF is described in section 2.4.5.

The longest of the original three intervals is DCF interframe space (DIFS), used by the distributed coordination function (DCF) mode. DCF is the most commonly used operation mode today and is described in section 2.4.4.

For IFS calculations the smallest amount that can be added to a shorter IFS is called *SlotTime*. This time interval depends on the used PHY and is defined by the time required by another station to hear a packet sent by another. It includes required hardware processing time but also propagation time of a packet on the medium.

The values for PIFS and DIFS are derived from SIFS and SlotTime by setting

$$\text{PIFS} := \text{SIFS} + 1 \cdot \text{SlotTime}$$
$$\text{DIFS} := \text{SIFS} + 2 \cdot \text{SlotTime}$$

By specifying SIFS < PIFS < DIFS, the MAC rules grant priority access to answers in direct sequences. If no station needs to answer directly, a station (actually only an AP) operating in PCF mode may seize the medium after PIFS. And then if after DIFS still no station has sent a packet, stations operating in DCF may send.

For enhanced distributed channel access (EDCA) mode, a new set of interframe spaces called arbitration interframe space (AIFS) is introduced in 802.11e. For each QoS class n the corresponding time interval is denoted by AIFS[n]. The actual interval lengths are defined by the EDCA parameter set using an integer called arbitration interframe space number (AIFSN). From this integer the time interval AIFS is calculated as follows:

$$\text{AIFS}[n] := \text{SIFS} + \text{AIFSN}[n] \cdot \text{SlotTime} \tag{2.1}$$

Since the standard requires AIFSN[n] ≥ 2 for stations and AIFSN[n] ≥ 1, the values of AIFS[n] are always greater than SIFS and may be equal to PIFS, DIFS or larger. More about the AIFS is found in section 2.4.7 on EDCA.

2.4.4 DCF

The most commonly used coordination mode today is distributed coordination function (DCF). As explained in the previous section, a station operating in DCF must always wait DIFS time after a frame. However, multiple stations might have been waiting, so further rules are needed to decided which of the stations gets to send first. This must be done using a distributed algorithm that provides fairness between all stations. So the time a station must wait after DIFS is uniformly randomized. The range from which the random waiting time is chosen is called the contention window (CW).

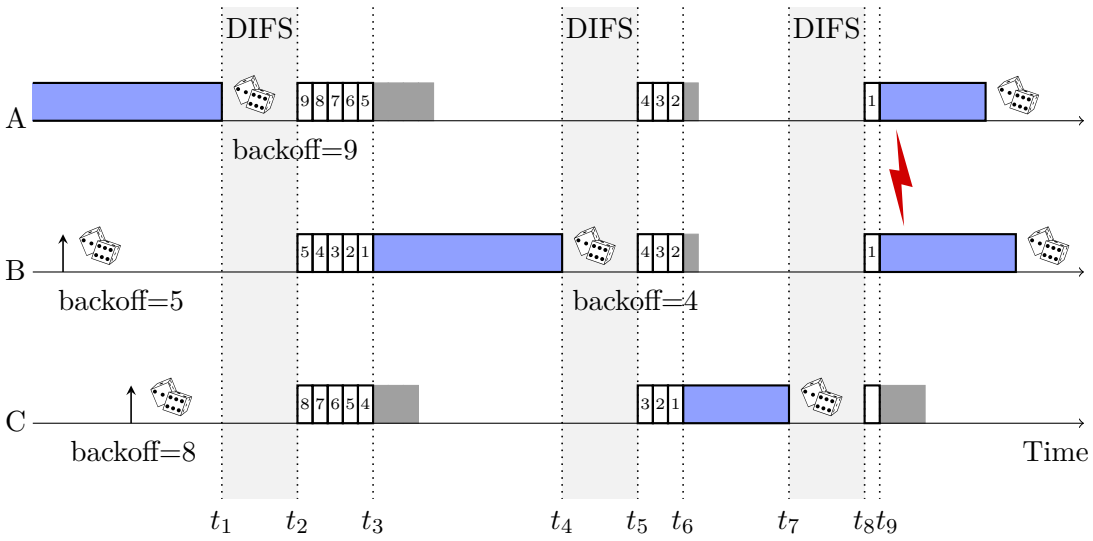


Figure 2.11: Backoff coordination example in DCF

The rules of DCF are following: if a station is newly initialized then it may send the first frame immediately after DIFS if the medium is free. If the medium is not idle, then a *backoff* time is uniform randomly chosen from the interval [0 . . . CW], where CW := CWmin. Once the medium is idle and DIFS elapsed, the station must still wait further and decrement the backoff counter for each SlotTime interval. If the medium is seized by a different station in that time, the backoff counter decrementing is stopped. Decrementing proceeds once the medium is idle again for DIFS. When the backoff counter reaches zero, the station may send the waiting frame.

If sending of the frame does not succeed, then a further backoff is started. The station can only determine whether a frame was correctly received if the frame required an ACK answer. If the corresponding ACK answer does not arrived after a specific timeout, the frame must be retransmitted. For this retransmission the backoff is randomly selected from an enlarged interval [0 . . . CW]. The initial CW is CWmin, which is determined by PHY specifics (see table 2.4). For subsequent retransmissions, the new CW is calculated by

$$\mathrm{CW}_{\mathrm{new}} := \min\{2 \cdot (\mathrm{CW}_{\mathrm{old}} + 1) - 1,\ \mathrm{CWmax}\} \tag{2.2}$$

If the frame transmission was successful or no ACK was required, then the station must always start a new backoff procedure, regardless if it has another packet waiting or not.

Figure 2.11 shows an example time line of DCF's packet coordination. The example contains three stations all within range of each other and initially station A is sending a packet. While A is sending, the higher layers in both B and C queue a packet for transmission. B and C determine that the medium is busy and must pick a random backoff. In the example, B picks 5 and C's backoff is randomized to 8. When A finishes its broadcast frame at t_1, it too must start a backoff timer in this case with initial value 9.

After DIFS all stations start decrementing their backoff counters. The small rectangles each depict an interval of SlotTime. After 5 SlotTimes, station B wins the distributed competition and is allowed to send its frame at t_3. Because all other stations sense that the medium is busy, they stop their backoff counters. Once B has finished its frame, it starts a new backoff to transmit an additional frame and this time must wait 4 slots.

At t_5 all station again start decrementing and this time C's counter, which still contained 3, first reaches zero. The frame transmitted by C finishes at t_7 and a new backoff is started for C. After DIFS all stations again start decrementing. Because both A's and B's counters contain a remaining backoff of 1 slot, both transmit their packets after 1 SlotTime at t_9. Due to the interference of both packets, they will probably not be received correctly and a retransmission might be necessary.

As the example shows, DCF does not prohibit collisions, it only avoids them using random backoffs. By increasing the CW interval, DCF automatically adapts to higher network load.

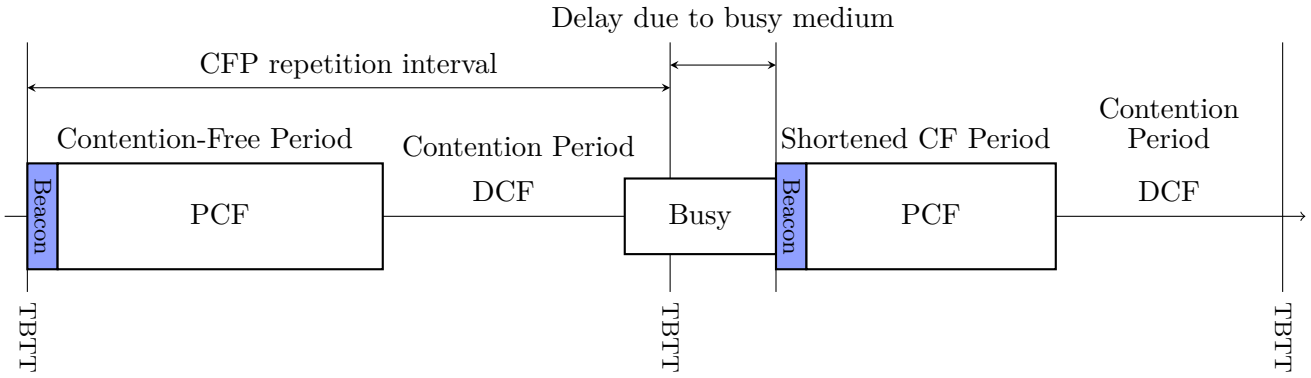


Figure 2.12: CFP/CP alteration (adapted from [12, figure 9-13])

DCF is statistically fair to all stations with respect to the time they have to wait before sending. However, it is not fair with respect to the amount of bytes transmitted by each station; packet payload size or transmission time is not taken into account by DCF. Thus byte throughput is not fairly coordinated by DCF. Obviously DCF is not designed to provide QoS: all traffic is delivered with best effort semantics.

2.4.5 PCF

The second of the original operation algorithms of 802.11 is the point coordination function (PCF). It provides an enforced "fair" access mode during a contention-free period (CFP). The CFP is initiated and coordinated by a so called point coordinator (PC), which is a subfunction of an AP.

Contention-free period (CFP) and contention periods (CPs) alternate, with PCF controlling transmission in the CFP and DCF in the CP (see figure 2.12). A CFP is started by the PC with a beacon frame. In this beacon frame the NAV duration is set to the maximum value and thus stations operating in DCF are not allow to send.

The PC uses the shorter interval PIFS to send the beacon frame with higher priority than usual DCF traffic. Having seized the medium, the PC keeps control until the end of the CFP.

Frame transmission is coordinated by the PC in the CFP with an algorithm similar to token-passing: a station may only send a frame if polled by the PC. A typical PCF frame sequence is shown in figure 2.13. The frames used in PCF can have two different flags: CF-Poll, CF-ACK. Both flags can be set independently and the frame may or may not contain payload data. All eight combinations are possible.

The PC polls each station from its polling list with a CF-Poll. The polled station must answer with one CF-ACK. If the PC/AP has data queued for the polled stations, a data frame with CF-Poll flag is transmitted, otherwise an empty frame with the flag is sent. Likewise the station, which must answer with CF-ACK, transmits a data or empty frame. The PC receiving a frame from a station also must answer with a CF-ACK. This CF-ACK can be combined with a new CF-Poll to a different station and may also contain data for that station.

PHY	SlotTime	SIFS	DIFS	CWmin	CWmax
HR/DSSS	20 µs	10 µs	30 µs	31	1 023
OFDM 5 GHz					
with 20 MHz channels	9 µs	16 µs	34 µs	15	1 023
with 10 MHz channels	13 µs	32 µs	58 µs	15	1 023
with 5 MHz channels	21 µs	64 µs	106 µs	15	1023
ERP-OFDM 2.45 GHz					
with short preamble	9 µs	10 µs	28 µs	31 / 15[1]	1 023
with long preamble	20 µs	10 µs	50 µs	31 / 15[1]	1 023

[1] Value depends on data rate set used.

Table 2.4: Timing and DCF parameters for different PHYs

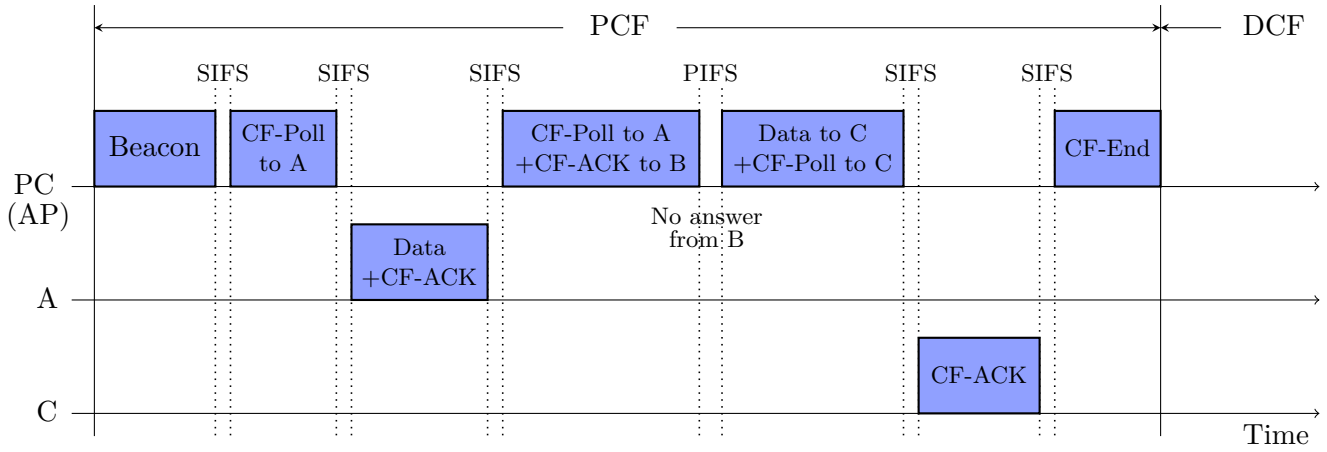


Figure 2.13: Example PCF contention-free period

If a polled station does not answer, then the PC continues its polling list after PIFS. If it would wait longer, then a station that operates in DCF and overheard the NAV could seize the medium. Therefore, during the CFP no idle period longer than PIFS occurs.

The CFP is terminated by the PC using a packet flagged with CF-End. With the CF-End packet, the NAV is reset and usual DCF traffic operation commences. The duration of the following CP must allow at least one complete frame transmission by a station operating in DCF.

Beacons are transmitted in periodic intervals. The interval is announced by the AP in a parameter set attached to previous beacons. From this interval the next target beacon transmission time (TBTT) can be determined. However, because traffic operating in DCF may be holding the medium at the TBTT, a beacon frame may be delayed. If it is delayed the maximum duration of the following CFP is shortened.

QoS using PCF

The PCF is a mechanism for the PC to manage frame transmission. However, the level of QoS that can be provided using PCF is very limited. The two main problems with PCF are listed below, see [22] for details.

To send a beacon at the TBTT, the medium must be idle for PIFS. However, due to DCF traffic the beacon can be delayed up to about 4.9 ms [22]. The worst case is a DCF transmission using RTS/CTS, the longest payload size, slowest modulation and coding scheme and high fragmentation.

Second major problem is that the PC cannot enforce frame sizes transmitted by polled stations. Stations are free to send very long frame sequences, up to the maximum of 2 304 bytes payload with the slowest transmission scheme. This makes any attempt to provide high grade QoS impossible.

2.4.6 HCF

The shortcomings of DCF and PCF motivated development of a new coordination function in 802.11e [13]. The enhanced function, named hybrid coordination function (HCF), includes provisioning of high QoS guarantees. HCF manages both contention-free and contention-based access to the wireless medium, which is why it is called *hybrid*.

Central to providing QoS support is the concept of a transmission opportunity (TXOP). A TXOP is an allotment of access time to the medium, which can be used to transmit a single frame or multiple frames in a sequence. TXOPs are defined by start time and duration. A station sending on a TXOP must complete its frame transmission including acknowledgements within the defined time interval, it must not utilize the medium for longer than the allocated duration. Due to this hard rule, QoS can be provided by assigning TXOPs in a coordinated fashion.

The HCF defines two medium access mechanisms: enhanced distributed channel access (EDCA) and HCF controlled channel access (HCCA). EDCA is used only in the contention-based periods, while HCCA used in

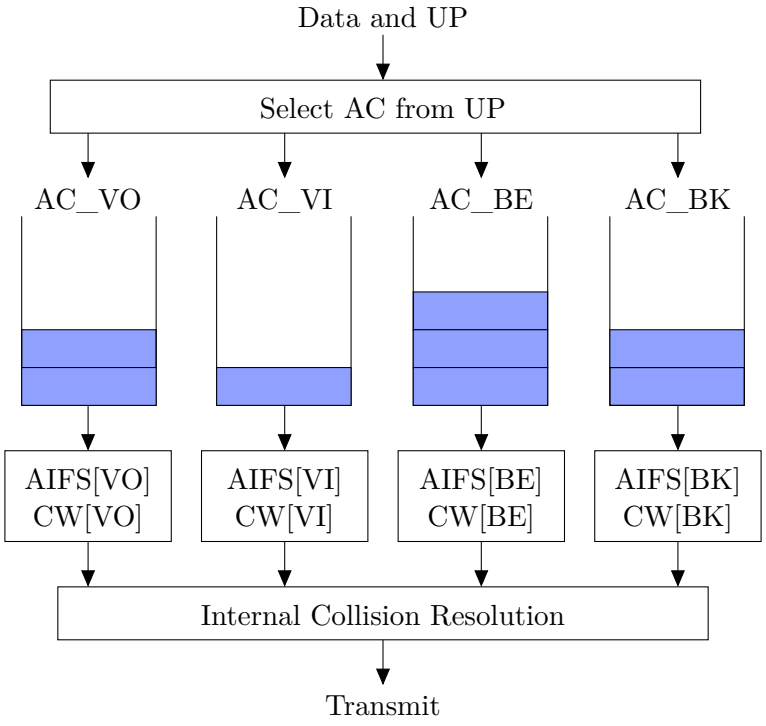


Figure 2.14: EDCA queues

both contention and contention-free periods. Both are enhancements of the original coordination functions DCF and PCF and will be described in the next two sections.

2.4.7 EDCA

In contention periods (CPs) of HCF, TXOPs are allocated by EDCA using a set of distributed rules, which are based on those defined in DCF. Similar to DCF, EDCA is very likely to become the most widely used channel access mechanism, because it is simple, easy to implement and hassle-free to employ due to its distributed nature.

EDCA defines four access categories (ACs) with different medium access priorities. The four AC are labeled AC_VO for "voice" class traffic, AC_VI for "video" class traffic, AC_BE for best effort traffic and AC_BK for background traffic. For packets received from higher layers that are tagged with user prioritys (UPs) defined in 802.1D [15], a mapping to ACs is defined in the standard [12, table 9-1]. The four classes are enumerated with an access category index (ACI): 3 for AC_VO, 2 for AC_VI, 0 for AC_BE and 1 for AC_BK.

Each AC has a separate queue and enhanced distributed channel access function (EDCAF) managing backoff (see figure 2.14). Note the difference between EDCA as a submechanism of HCF and an EDCAF, which defines when packet transmission is allowed for each AC. Thus there are four EDCAF in a QoS station.

An EDCAF is very similar to the original DCF (review section 2.4.4). The differences can be summarize with following modifications:

Like DCF, the medium must be idle for a specific time interval before the EDCAF is granted access or starts decrementing the backoff counter. This interval, denoted as AIFS[AC], corresponds to DIFS in DCF. It is defined by the integer AIFSN[AC] through the equation described in section 2.4.3. For backwards compatibility with PCF, the value of AIFSN[AC] must be greater or equal to 2 for non-AP stations. See figure 2.10 for an illustration of the interframe spaces. The figure shows two AIFS with AIFSN set to 2 and 4.

If a transmission needs to be delayed, a random backoff is chosen from $[0 \ldots \mathrm{CW}]$. As with DCF, the CW value starts at CWmin[AC] and is increased for each retransmission, up until CWmax[AC] is reached. Each

	802.11e				802.11p/D4.02				DCF
AC	VO	VI	BE	BK	VO	VI	BE	BK	
AIFSN[AC]	2	2	3	7	2	3	6	9	2
CWmin[AC]	3	7	15	15	3	3	7	15	15
CWmax[AC]	7	15	1 023	1 023	7	7	15	1 023	1 023
TXOPLimit[AC]	1 504 µs	3 008 µs	0	0	0	0	0	0	0

Table 2.5: Default EDCA parameters using OFDM PHY in 5 GHz bands

EDCAF has a separate CW variable. The backoff counter is decremented only after the medium is idle for AIFS[AC].

The previous two modifications adapted variables of DCF to differentiate medium access for different ACs. The following third change is very important to provide QoS, for reasons that first unfold in the next section on HCCA.

When the backoff counter reaches zero, the AC is granted an EDCA-TXOP. This TXOP has a maximum duration specified by the parameter TXOPLimit[AC]. A TXOPLimit = 0 allows a single frame at any data rate and an addition RTS/CTS exchange if desired. Note that the TXOP is granted to the AC, not to the station.

Last modification is the addition of a admission control mandatory (ACM) parameter, which specifies whether a station needs permission from the AP to use the AC. This is important in HCCA controlled BSSs.

A QoS-enabled station holds four EDCAFs, so it can occur that two or more backoff counters reach zero simultaneously. This is called an internal collision and is resolved by granting access only to the EDCAF with highest priority. The other EDCAF must initiate the usual backoff procedure just as if an "external" collision on the medium is detected.

Default EDCA Parameters

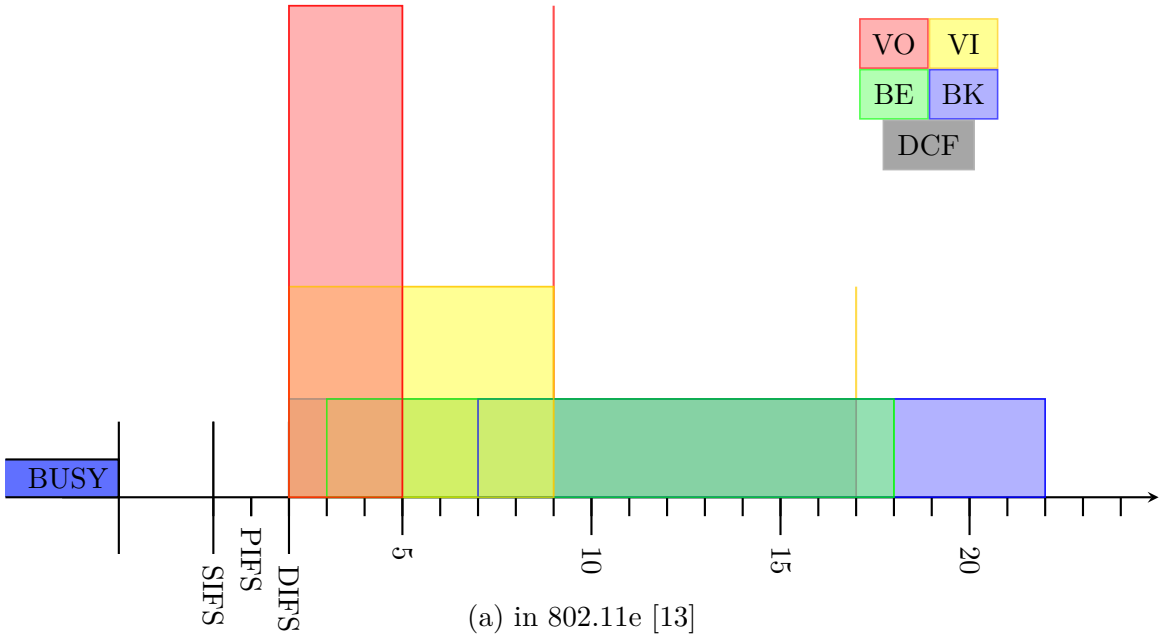
EDCA can be used with or without an AP as central controller. If EDCA is used within a BSS controlled by an AP supporting QoS, the EDCA parameters used by all associated stations are defined in the beacon. This allows the AP to tune and limit the QoS provided by EDCA depending on administrative requirements and other QoS constraints.

If EDCA is used in an ad-hoc BSS (IBSS) or outside the context of a BSS (802.11p), then a default predefined parameter set is used. This default parameter set is different for each PHY layer and is defined in the standard by calculations from other parameters. Table 2.5 shows the default parameters from 802.11e and 802.11p for one specific PHY. See section B.2 in the appendix for default parameters of other PHYs. The default parameters may be changed by the SME to adapt the prioritization required.

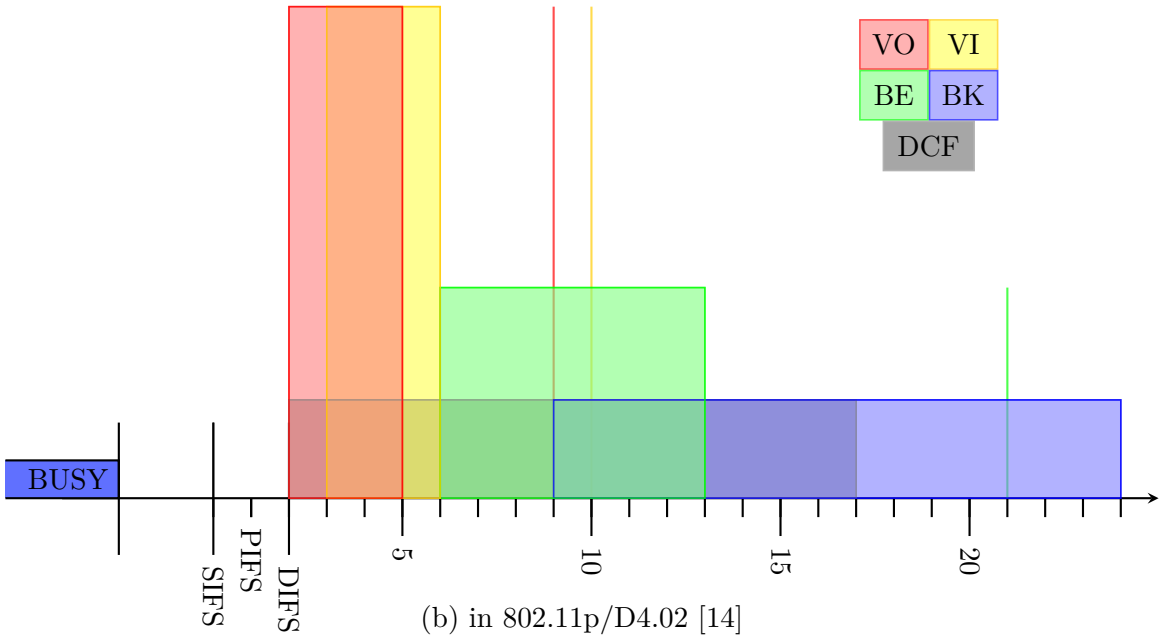
A better understanding of the parameter values can be gained from figure 2.15. These two diagrams visualize the default EDCA parameters of 802.11e [13] and 802.11p/D4.02 [14]. Each tick on the x-axis represents one backoff slot. The colored rectangles mark the time a frame must wait for the first transmission attempt: each rectangle's left border is at AIFS[AC] and the right border at AIFS[AC] + CWmin[AC]. All rectangles have the same area. For each AC, the CWmax[AC] is displayed as a single vertical line in the corresponding color.

Because all rectangles have equal areas, the rectangle's height symbolizes the probability of transmission at the slot boundary. Summing up all rectangle heights at a slot boundary yields the probability of collision at that point in time.

Note that the illustration only shows waiting time for the *first transmission attempt*, in absence of other contenting stations. Other stations may seize the medium and thus delay backoff or even cause a collision. Subsequent attempts due to transmission errors have different waiting times. However, it is only possible to detect a transmission error if positive ACK is used for the packet. This is not the case for broadcast, multicast and frames marked with NoACK.



(a) in 802.11e [13]



(b) in 802.11p/D4.02 [14]

Figure 2.15: First transmission attempt with default EDCA parameter

The parameters of 802.11p [14], illustrated in figure 2.15(b), show two very interesting and useful properties:

- The first observation that can be made is that packets from AC_VO and AC_VI are always transmitted before frames from AC_BE and AC_BK *in the first transmission attempt*. This is due to

  $$\text{AIFSN[AC\_VO]} + \text{CWmin[AC\_VO]} \leq \text{AIFSN[AC\_BE]} \leq \text{AIFSN[AC\_BK]}$$
  $$\text{AIFSN[AC\_VI]} + \text{CWmin[AC\_VI]} \leq \text{AIFSN[AC\_BE]} \leq \text{AIFSN[AC\_BK]}$$

  This holds even at the slot time boundary 6, where an internal collision occurs if the packet from AC_VI starts with a maximum backoff of CWmin[AC_VI] and the packet from AC_BE chooses backoff 0. The internal collision is resolved by granting access to AC_VI.

- The second observation is that AC_VO frames are *always* transmitted before frames from AC_BK *at any retransmission attempt*. The follows from the equation

  $$\text{AIFSN[AC\_VO]} + \text{CWmax[AC\_VO]} \leq \text{AIFSN[AC\_BK]}$$

  Again this also holds at the time boundary, due to internal collision resolution.

In many scenarios legacy DCF stations will coexist with EDCA QoS-enabled stations. DCF allows transmission after DIFS, which is equal to an AC with AIFSN = 2. This severely intervenes with the prioritization provided by EDCA and therefore the relative QoS guarantees are only valid in pure EDCA setups.

2.4.8 HCCA

The second new medium access mechanism defined in 802.11e is HCF controlled channel access (HCCA). It is derived from PCF (review section 2.4.5) and keeps its fundamental polling nature. However, many modifications are made which enable HCF to provide true QoS in wireless LANs. Review of HCCA is included in this thesis because it is the sibling medium access mechanism of EDCA in HCF; it is not discussed beyond this section.

A BSS operating in HCF with HCCA is managed by the hybrid coordinator (HC). Similar to the PC in PCF, the HC may access the medium after only PIFS idle time and thus can seize the medium before stations operating in DCF or EDCA. Using this priority access, the HC can send beacons, poll stations or transmit data frames.

Unlike PCF the periods in which the medium is controlled by the HC using HCCA need not strictly follow beacon frames. Instead, periods controlled using distributed EDCA and central controlled HCCA can be intermixed as shown in the example figure 2.16. After each beacon there may still be a longer sequence of HCCA, which is called a contention-free period (CFP) like in PCF.

The arbitrary mixing of HCCA and EDCA periods is the first important change required for high grade QoS: the HC may seize the medium during the contention period to enforce QoS guarantees.

Stations may request a QoS reservation through a management frame holding a traffic specification (TSPEC). The major TSPEC parameters are stream direction, mean data rate, delay bound, maximum service interval, nominal MAC service data unit (MSDU) size and minimum PHY rate. If the HC determines that the reservation can be granted, a traffic stream (TS) is created with unique traffic stream identifier (TSID). Up to eight upstream and eight downstream TS are supported simultaneously.

In HCCA phases, the HC then sends special QoS-CF-Poll management frames to stations holding a QoS reservation. This QoS-CF-Poll grants the station a polled HCCA-TXOP for the allocated TS that is used for upstream traffic. For downstream traffic in direction of the STA, the HC can gain access directly after PIFS.

The HC must determine if the QoS parameters given in a TSPEC can be assured. By changing the EDCA parameters broadcasted in each beacon, the HC can limit medium access of other stations operating outside the TS allocations. By setting the ACM[AC] it may even deny stations access to high priority ACs.

Beyond requesting TSs, all stations can use a new field in each QoS-data frame to request a further TXOP. Depending on current network load, the HC can grant this piggy-backed request or not.

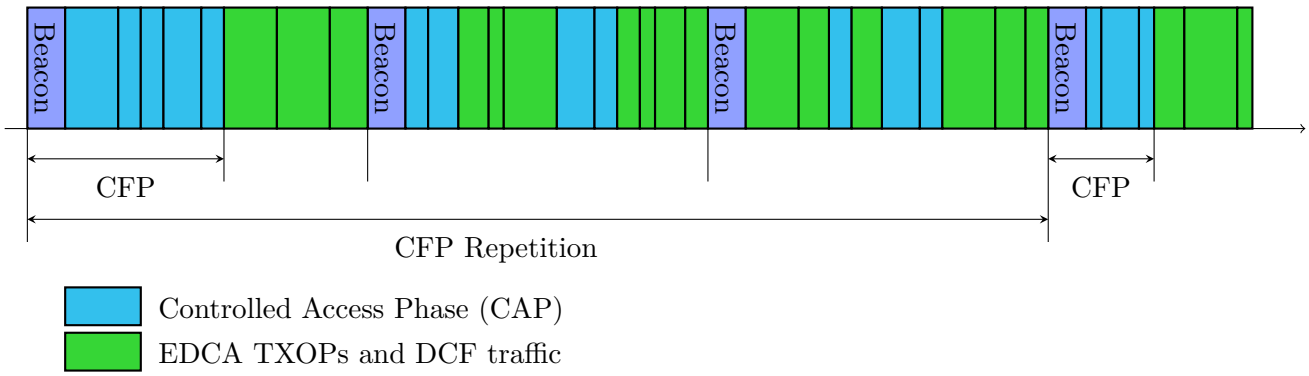


Figure 2.16: HCCA CAP/CF/CP periods (adapted from [12, figure 9-10])

Coordination and guarantee of QoS is burdened on the HC as controlling instance in the BSS. Admission control of new TS, adaptation of EDCA parameters and HCCA coordination is a challenging problem and many algorithms and protocols have been developed [8].

Using HCCA, QoS can be guaranteed within the inherit limits of wireless communications. In unlicensed bands these guarantees are severely limited due to unpredictable interference. However, in licensed bands and controlled environments high grade QoS guarantees can be given.

Chapter 3

The ns-2 Network Simulator

ns-2 is a discrete event network simulator specially geared for scientific research in packet-switched computer networks. It is doubtlessly the most popular open-source network simulator today.

Work on ns-2's ancestor, the LBL Network Simulator (sometimes called ns-1), began in May 1990 at Lawrence Berkeley Nation Laboratory as a variant of the REAL network simulator. S. Keshav's REAL network simulator was originally designed for research on dynamic behavior of flow and congestion control schemes [19]. In 1994, the original ns changed its simulation description language to Tcl [24].

The first version of ns-2 was released in 1995. At that time ns had gained support from DARPA (Defense Advanced Research Projects Agency) and the VINT (Virtual Inter Network Testbed) project at LBL. Further backing was given by Xerox PARC, UCB (University of California, Berkeley) and USC/ISI (Universitry of Southern California, Information Sciences Institute) [17]. Important contributions have come from Sun Microsystems, the UCB Daedelus and Carnegie Mellon Monarch projects. Until 2004 the NSF (National Science Foundation) CONSER (Collaborative Simulation for Education and Research) and DARPA SAMAN (Simulation Augmented by Measurement and Analysis for Networks) projects funded development. Currently another NSF funded project is maintaining ns-2 in collaboration with the University of Washington, Georgia Institute of Technology, ICSI and the Planete group at INRIA.

Today ns-2 is developed in an open-source manner by a number of different researchers and institutions with individual focuses. The code base is maintained by a NSF funded collaboration and periodically new versions are released.

3.1 Overview

This section gives a very short overview of ns-2's basic architecture and properties. More instructive tutorials are best found on ns-2's official web site [25].

ns-2 is a discrete event simulator and as such simulated time in ns-2 does not progress continuously. Instead, the simulated world changes state at specific points in time called events. These changes are made instantaneously with respect to simulated time by event handlers, which may schedule further events at any later point in time. The simulation stops when no more events are to be processed or by user intervention.

To simulate computer networks, ns-2 is designed to imitate the flow of packets on a network. The simulated world contains a set of communication nodes and each node contains a collection of network objects, which can represent applications, ISO/OSI network layers and more. These network objects interact with each other by passing packets around. Just like in real networks, packets are first augmented with header and trailer, when traveling down a stack of network layers. And then, after traveling through an emulated medium, the information from header and trailer is used to disassemble and process the packet as it travels up the stack.

Two programming languages are used in ns-2: OTcl (MIT Object Tcl) and C++ (see figure 3.1). The rationale behind this dual language approach is to make writing simulation scripts easy and flexible in Tcl, while implementing the performance critical code in C++. Because Tcl is an interpreted language, changes to a simulation script do not require lengthy compilations. However, this flexibility comes at the

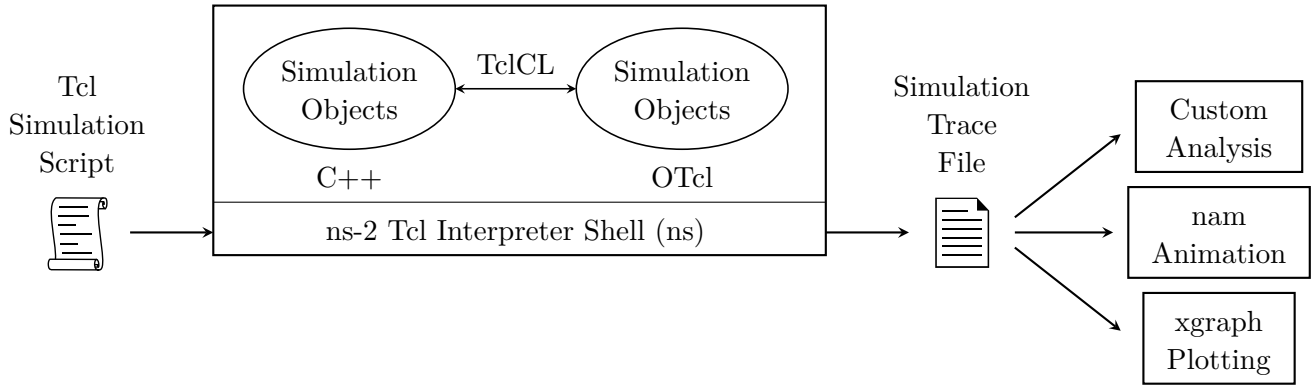


Figure 3.1: Basic ns-2 simulation architecture (based on [17, figure 2.1])

cost of slower execution speed. The heavy simulation core and simulation models close to the hardware are implemented in C++ classes.

However, due to this dual language design, all objects which need to be available in both programming domains must provide dual interfaces in both C++ and OTcl. These dual interfaces are maintained by TclCL bindings and shadow objects in C++.

Through the years ns-2 has grown very large, currently amounting to about 300 000 lines of code, and many models at different network layers were implemented. Among the highlights are many variants of TCP, special queues and classifiers, the SCTP and XCP protocols, configurable traffic generators with different behaviors and HTTP, FTP, UDP, PLM, SRM and RTP application simulators. Support for research in mobile wireless networks, satellite networks and differentiated services (DiffServ) is also included in the base release. Beyond the official code many special patches are available from groups doing research on specific topics.

3.2 Problems with 802.11 and Overhaul by DSN

IEEE 802.11 wireless and mobile node source code was first added in 1997. It was contributed by Sun Microsystems, the UCB Daedelus and Carnegie Mellon Monarch projects. Many research papers have been published with simulation results using these 802.11 modules in ns-2.

The 802.11 implementation has had some major shortcomings, oversimplifications and even errors in the past. These shortcomings and their duration in the main code base are surprising given the amount of research papers based on ns-2 wireless simulations.

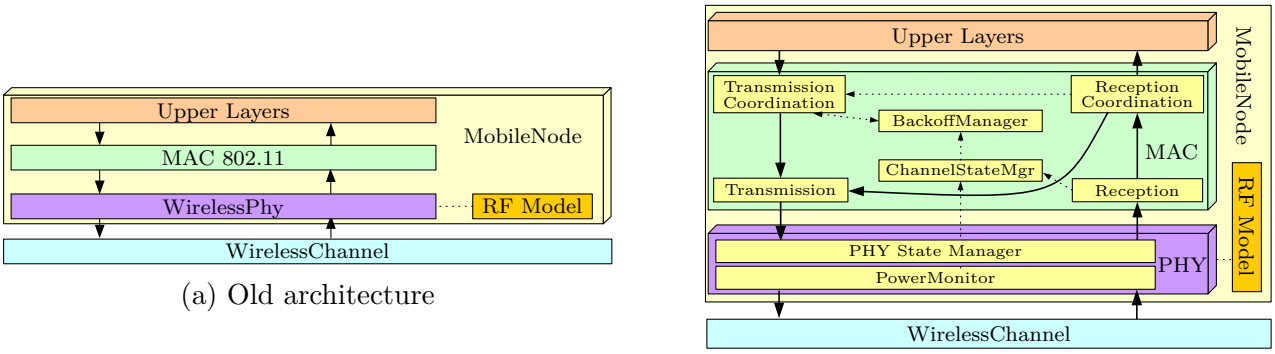
Only the basic DCF operation mode is implemented in the mainstream code. Many patches to the wireless code were developed by different research groups. Among them are EDCA support by the Telecommunication Networks Group at the Technical University of Berlin [37] and HCCA support by the Computer Networking Group of the University of Pisa [3].

Infrastructure mode with AP beacons and station association has only recently been added in ns-2.33, which was released in March 2008.

On a whole the 802.11 implementation in ns-2 is not well designed and poorly documented. This has been known for a long time and ns-2 lacked behind other network simulators, which are more specifically geared towards wireless networks.

To reliably simulate large, highly mobile VANETs, the 802.11 implementation has been minutely studied by the DSN research group at the University of Karlsruhe and Daimler Research California. In 2006, many bugs in the 802.11 modules were identified and a patch for these was published by the DSN [34]. Some of the bugs and improvements are listed below.

- Incorrect simulation of the extended interframe space (EIFS) by setting the NAV instead of actual backoff. Correction of this bug required large structural changes in the NAV and backoff implementation.



(a) Old architecture

(b) Overhauled architecture

Figure 3.2: 802.11 simulation modules and architecture in ns-2 (adapted from [2, figures 2 and 3])

- Sensing of incoming frames while transmitting, a feature not available to 802.11 hardware.
- Extended frame capture as supported by newer wireless chipsets. Frame capture is described in section 6.4.

Not fully satisfied with patching up existing code, the DSN group and Daimler Research published an overhauled implementation of 802.11 MAC and PHY for ns-2 in 2007 [2]. Main goal of this revised implementation was to improve overall code structure and design. Key features of the redesigned architecture (figure 3.2) are

- a much more accurate modeling of the PHY layers, including signal to interference and noise ratio (SINR) computation with cumulative noise and correct distinction between PLCP preamble, header and payload,
- optional, extended frame capture as described in section 6.4 and
- the probabilistic Nakagami propagation loss model, which includes Rayleigh fast fading and enables realistic simulations of radio propagation in urban environments (see section 5.4).
- Moreover the MAC layer was remade and modularized into transmission and reception coordination classes, DCF backoff management and a channel monitor.

This completely revised architecture was included in the newest ns-2.33 mainstream code. The two base classes modeling MAC and PHY are called `Mac802_11Ext` and `WirelessPhyExt`.

In this thesis the revised 802.11 architecture is compared with the new ns-3 simulator, described in the following section. The improvements listed above, that were not contained in ns-3 were ported and verified extensively.

Chapter 4

The ns-3 Network Simulator

The new ns-3 network simulator is, as its name suggests, the designated successor of the popular ns-2 simulator described in section 3. Its development initially began in 2006, the first public pre-alpha version was released in March 2007 and the first stable milestone was reached in June 2008. Maintenance and initial development are funded by a four-year NSF grant.

The goals of ns-3 [11] are set very high: to create a new network simulator aligned with modern research needs and develop it in an open-source community. This section describes both plans for the project and its current development state as of ns-3.4, released in April 2009.

In the initial project goals, it was decided that the new ns-3 simulation architecture was to be redesigned from scratch. In this design, historic experiences gained from ns-2 should be blended with advances in programming languages and software engineering. Strict backwards compatibility to ns-2 was explicitly declared to not be a project goal. This frees ns-3 from historic burdens and enables construction of a simulator that is "correctly designed" from the beginning.

There apparently was unanimous consent that OTcl should be replaced by a modern scripting language. OTcl was cited as one of the great obstacles in using and teaching ns-2. The dual language approach of ns-2 was also abandoned, for multiple reasons. It makes model development and debugging more difficult due to two programming language domains. Furthermore the C++/OTcl binding code is not well documented and introduces some subtle and unsolvable difficulties in connecting models written in the disparate languages.

The core of ns-3 and all models are written in C++. Front-ends to the C++ code should be created for various scripting languages like Python, Perl and also Tcl using specialized binding generators. Similar to ns-2's approach, these front-ends allow a user to create simulation objects in a flexible script language without compilation, and at the same time have the performance critical code in C++. Currently a Python front-end is automatically generated from C++ class files and is already used by example simulation scripts. Support for a wider variety of script languages through SWIG is planned.

Due to its age, C++ code in ns-2 was limited in its use of the language. At the time ns-2's foundation code was created, the standard template library (STL) was not wide-spread yet and compiler support for complex templates still poor. Consequently these modern C++ features and the STL were not allowed to be used in ns-2 for a long time. Because of these limitations, large parts of ns-2's code appear to today's developers ancient, overly complex and "hacky".

The core of ns-3 heavily utilizes advances in C++ compiler technology and software engineering. The fundamental scaffolding done in ns-3's core is described in the following section 4.1. The core framework contains a hierarchical object system with attributes, callbacks and integrated tracing. Built upon these foundations is a memory-efficient packet handling system and discrete event simulation algorithms, which drive progress in each simulation program.

One major project goal is to allow simulation of "large" networks with a reasonable level of detail. By heavily optimizing the simulator core code, scalability on a single computer can be improved to a limited degree. However, ns-3's goals go beyond simulations with a single, sequential event processing loop. Using parallel and distributed simulation technology, simulations should scale further, utilizing the memory and processing resources of a whole computation cluster. Experience gained from implementing the Georgia Tech

Network Simulator (GTNetS) [30], which uses "ghost nodes" [31], and conservative look-ahead techniques from Parallel/Distributed NS (PDNS) [27] should flow into ns-3's design. This goal has not yet been achieved. Some advances using a node federate-based approach with MPI were developed in a project in 2008.

To enable easy integration of new models into ns-3, its networking architecture is based on real world hardware and software. The interfaces in this architecture closely resemble those found in real operating systems and hardware: for example the BSD sockets application programming interface (API) and Linux network devices interfaces are emulated. The packets passed around in the simulation are required to be exactly like real network packets, down to the last bit.

By use of these well-known interfaces, ns-3 aims at leveraging existing model or real implementations. Three different binding techniques are devised to integrate existing models or code into ns-3:

First is to integrate models from ns-2 or other network simulators by adapting the code directly to ns-3 interfaces. This is the traditional approach used in most simulators and requires manual coding work by experts.

A second method aims at integrating existing open-source libraries implementing network protocols. For example the quagga routing protocol suite can be integrated by attaching it to the generic, well-known APIs provided by ns-3. This method envisions easy reuse of the large amount of open-source networking software, an opportunity missed by most other simulators.

Using the third set up, real operating system code can be integrated into ns-3. Such integration has already been done with the Network Simulation Cradle (NSC), which has successfully been ported to ns-3. Further work is planned to interface real applications with a slim emulation layer, e.g. using the sockets or libc API, and run them completely within ns-3.

Stepping outside the reuse of existing software, ns-3 also integrates real hardware into simulations. Because packets created and used by the simulation are indistinguishable from real packets, they can be exchanged with real networks. ns-3 already provides an implementation to do this on Linux.

One further goal of the ns-3 project is to maintain tools for educational use, a field in which ns-2 has not had much impact. The vision includes animations and educational courseware for undergraduate networking classes. By providing these example scripts, classes on networking can be enhanced with live simulations and evaluation.

As progress continues, further goals of ns-3 are to be added and existing ones revised. Until the first stable release in June 2008, main focus of development was on the core architecture. Current work is more wide-spread and includes enhancing the simulator with more models and interfaces to external software.

4.1 Design Overview

In this section an overview of the principle components of ns-3 is given. Each component's role in the ns-3 simulation architecture and main classes are reviewed. Most classes described in the following overview are documented more extensively using doxygen comments in the source code.

The components in figure 4.1 form a layered software architecture with the most basic concepts provided by the *core*, *common* and *simulator* components. The *helper* component contains cross-layer API enhancements and thus cuts through the usual stacking. Classes completely outside of the layer architecture are grouped in the *contrib* component.

All components rely only on lower software layers. Thus, for example, the *simulator* and *core* components could be used independently for a discrete-event simulator not focused on packet networks. All C++ classes in ns-3 are enclosed in the namespace `ns3`, which is usually omitted in this thesis for legibility.

Core This module forms the basis of every ns-3 program by creating a hierarchical C++ class structure deriving from the base `Object`. Almost all objects in ns-3 are derived from `Object`, with some notable exceptions. This object hierarchy has many features specially geared for simulation purposes. Among these are smart `Ptr`s, object aggregation, `TypeId`s with `Attribute`s, `Callback`s and `TracedCallback`s. These concepts are explained in section C.2.

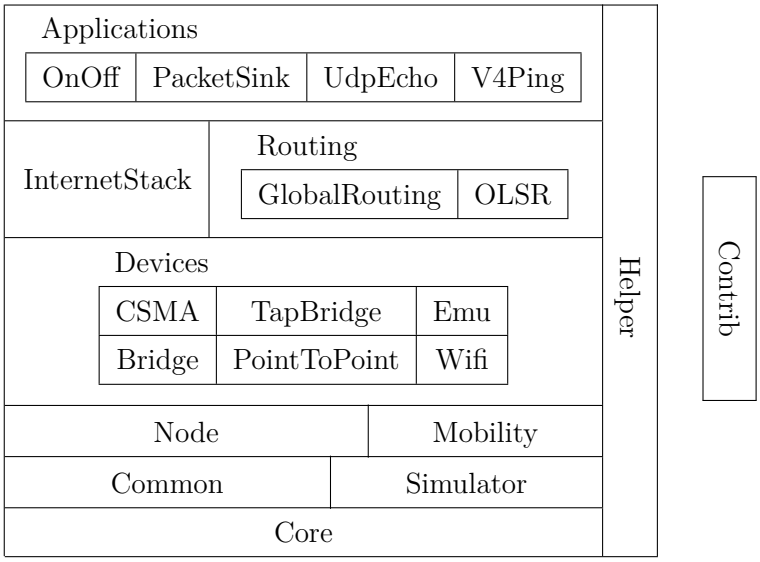


Figure 4.1: ns-3 main components

Furthermore the *core* also contains a `RngStream` "combined multiple-recursive" random generator called "MRG32k3a", upon which a number of `RandomVariable`s like `UniformVariable`, `ExponentialVariable` and `NormalVariable` are based.

Assertion (`NS_ASSERT()` and `NS_ASSERT_MSG()`) and complex logging macros (`NS_LOG_ERROR()`, `NS_LOG_WARN()` etc) are also defined in this component.

Common Most of this component is centered on the `Packet` class used for packet-level network simulation. The current `Packet` implementation employs a *copy-on-write* technique: copying packet objects is cheap because the packet data is not duplicated. Only on the first modification the complete packet is deeply copied. Another optimization is included for packets in which only the size but not the payload is important to the simulation, e.g. like packets from traffic generators. These unused bytes are not allocated in memory; the `Packet` class pretends to contain a zero-filled buffer, until it is really needed.

`Packet`s are manipulated using classes derived from `Header` and `Trailer`. When adding a specific `Header` to a packet, the header fields are serialized and prepended to the packet byte buffer. Later in the simulation, the `Header` can be removed again and automatically deserialized into the specific header's fields.

Aside from augmenting packets with `Header`s and `Trailer`s, layers in ns-3 can also attach `Tag`s to packets. Each `Tag` stores an arbitrary amount of bytes with a packet, but does not add to the packet's actual byte buffer. This implies that, if the packet is sent over a real network, any attached `Tag`s are lost. There are many uses for `Tag`s; the most common ones are cross-layer signaling like QoS tags and end-to-end application measurements.

Two more network-related classes in this component are `DataRate` and `ErrorModel`. `DataRate` allows easy specification of byte or bit rates as strings like "54MB/s" or "400kbps". And the `ErrorModel` class can be used to simulate random packet corruption.

Simulator This module enables discrete event simulation. A very important part of the simulation foundation is laid by the `Time` class. It represents high-precision simulated time points with a 128-bit integer value. The default smallest non-empty time interval in ns-3 is one nanosecond. Note that in contrast to using *double* variables, time computations using integers are exact. Particularly this means that all time values are equally spaced, all nanosecond values exist and are pairwise different. This is very unlike time represented with a *double* variable, because its values are not spaced equally and resolution depends on the absolute value represented. Fast computation of 128-bit time values requires 64-bit CPUs, as investigated in section 8.

Using this representation of simulated time, a `Simulator` coordinating scheduling of `Event`s is built.

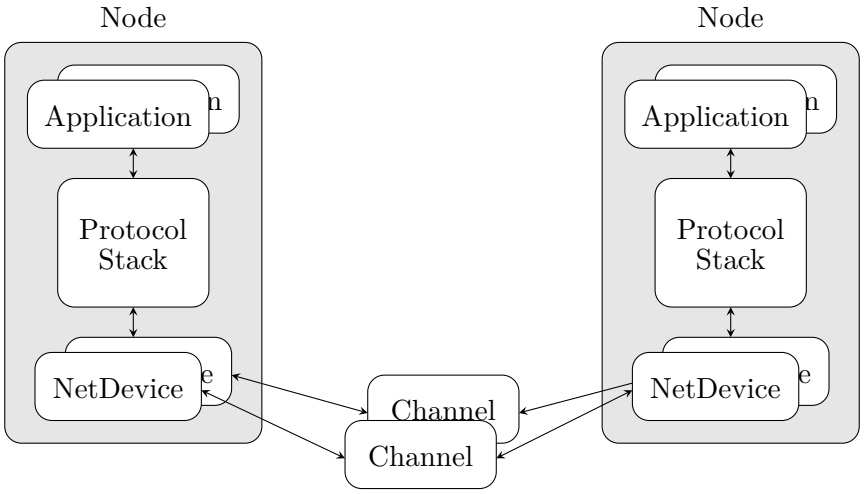


Figure 4.2: ns-3 node architecture (adapted from [10])

There are currently five actual implementations of the scheduling data structure: `MapScheduler` (current default), `HeapScheduler`, `ListScheduler`, `CalendarScheduler` and `Ns2CalendarScheduler`. There are also two `Simulator` implementations: standard virtual time progression (`DefaultSimulatorImpl`) and time progression synchronized to real system clock time (`RealtimeSimulatorImpl`).

The function of `Simulator` most often referenced by other modules is `Schedule()`, which inserts a new `Event` in the queue. Different from ns-2, the called event handler need not be an object derived from a class `Event`, which doesn't exist in ns-3. By using callback facilities provided by the *core* module, any class or basic function can be called directly with a list of provided parameters. An example on using callbacks can be found in section C.1.

Node The *node* module layouts the simulated network objects into network nodes containing protocols stacks. This layout closely resembles that of real operating systems.

It defines a `Node` class which can contain multiple `NetDevice`s. Each `NetDevice` is attached to a `Channel`, via which it sends and receives packets. Figure 4.2 illustrates this architecture for two nodes. Both `NetDevice` and `Channel` remain abstract classes in the *node* module; specific implementations are found in the *devices* component. One basic implementation, however, is included and described below.

A node may contain multiple protocol handlers, which accept packets received by the `NetDevice`s. Real Ethernet protocol identifiers (EtherType) are used to switch between handlers.

To initiate packet transmission, each node can also contain a list of `Application`s. Again `Application` remains an abstract class at this level.

In the *node* component, the binding interfaces between `Application`s, protocols and `NetDevice`s is defined. These interfaces are designed to be similar to those in real operating systems. Some `Queue`s are also defined between the layers.

The interface between `Application`s and protocols is based on the BSD sockets API. However, instead of C-style `sockaddr` structs, ns-3 uses a polymorphic `Address` class. Only non-blocking calls are supported by the API. Three `Socket`s are currently implemented in ns-3: `PacketSocket`, `TcpSocket` and `UdpSocket`. The simplest is `PacketSocket`, which sends packet data without extra headers to a destination via the underlying `NetDevice`.

And the second interface, between protocols and `NetDevice`s, resembles the basic 802.2 LLC interface. Packets may be sent to other stations identified by a MAC address and decoded there using an attached EtherType identifier. The `LlcSnapHeader`, containing LLC and SNAP encapsulation headers, is added by the `NetDevice`.

One actual `NetDevice` implementation called `SimpleNetDevice` is included in this module. Together

with `SimpleChannel` the `SimpleNetDevice` form a basic packet duplication and distribution with no particular properties modeled.

Mobility To enable position-aware nodes, this component defines a world geometry. To each node in the simulation a `MobilityModel` is aggregated, which maintains world position and velocity of the node. Currently ns-3 contains seven mobility models: `ConstantPositionMobilityModel`, `RandomWalk2dMobilityModel`, `RandomWaypointMobilityModel` and more. Only `ConstantPositionMobilityModel`, which defines fixed positions, is used in this thesis.

Devices Each device defined by these modules must fulfill the `NetDevice` interface and can be attached to a node. Most network devices transmit packets via a virtual `Channel` to other instances of the same `NetDevice` class. Five `NetDevice`s are implemented in ns-3.4:

CSMA The `CsmaNetDevice` class implements a CSMA bus resembling IEEE 802.3 Ethernet. Packets are transmitted via a `CsmaChannel` to which other `CsmaNetDevice`s are attached. However, no specific PHY implementation of 802.3 is used, the supported data rate is defined by setting a channel attribute. No Ethernet-like collision detection is used or implemented: all devices instantaneously sense when transmission on the channels starts. If the channel is detected to be busy upon transmission attempt, an exponentially increasing random backoff algorithm is invoked.

PointToPoint A simple point-to-point link is implemented using exactly two `PointToPointNetDevice`, which are connected via a `PointToPointChannel`. This link is a PPP-like direct link on which packets are encapsulated with `PppHeader`.

Emu Using an `EmuNetDevice`, a ns-3 simulation can communicate with the real outside world via a network device in the simulation computer. The emulated device fakes MAC addresses using raw sockets and sets the device to promiscuous mode. Thus the network node simulated in ns-3 appears to other real network members just like another attached device, different from the simulation host. Because `EmuNetDevice` uses raw sockets, simulations using them require root rights.

TapBridge Similar to `EmuNetDevice` the `TapBridge` allows ns-3 to communicate with a real network. For this the Linux "tap" device is used, which allows an application to drive a virtual network device in the operating system. Contrary to raw sockets, this does not require root rights. However, because a "tap" device is an internal network device, communication with the real world network is only possible via a bridge in the simulation host.

Bridge This component implements a 802.1D `BridgeNetDevice` to which multiple `NetDevice` can be attached. These are grouped and can be used by upper layers like a normal `NetDevice`. A learning bridge algorithm forwards incoming packets to the destination host via one of the outgoing net devices.

Similarly a `BridgeChannel` class is available, which aggregates multiple `Channel`s and forwards packets sent on them.

Wifi The *wifi* module is currently by far the largest network device component. It implements IEEE 802.11 wireless LAN and is the focus of this thesis. Section 4.2 contains a detailed discussion of the architecture of this model.

In short a `WifiNetDevice` is defined with resembles a wireless card. Different 802.11 MAC and PHY layers can be used to simulate the properties of wireless communication protocol and hardware.

InternetStack Classes in this module define ISO/OSI stack layer three and four protocols. Currently only IPv4 is supported, an IPv6 implementation is currently in progress.

Main class for IPv4 is `Ipv4L3Protocol`, performing protocol demultiplexing and packet forwarding between devices. A high level, Linux-like network device interface is provided by `Ipv4Interface`. Routing protocols are plugged in via the abstract class `Ipv4RoutingProtocol`. Layer 4 transport

protocols are attached via the interface `Ipv4L4Protocol`, which then receive packets tagged with their IANA-assigned protocol number.

Simplest `Ipv4L4Protocol` is `Icmpv4L4Protocol` for ICMPv4 signaling using packets constructed with `Icmpv4Header`, `Icmpv4Echo`, `Icmpv4DestinationUnreachable` etc.

The TCP implementation in ns-3 is ported from GTNetS [30] and implements the Tahoe TCP variant. Main class is `TcpSocketImpl`, which contains TCP handshaking, fragmentation, reordering, retransmission, acknowledgements and all other aspects. Packets are received from the IPv4 layer via the protocol handler `TcpL4Protocol`, which uses `Ipv4EndPointDemux` to demultiplex by IP port numbers.

Similarly the UDP implementation has a IPv4 protocol handler `UdpL4Protocol` and a socket communication context `UdpSocketImpl`.

Both `TcpSocketImpl` and `UdpSocketImpl` have abstract super classes `TcpSocket` and `UdpSocket`, which can be used to add other variants. These super classes are themselves descendants of `Socket`, which should be regarded as the object-oriented equivalent to the file descriptor in the sockets API.

ARP resolution is managed by the `ArpL3Protocol` with `ArpCache` and `ArpHeader`.

Routing Two routing algorithms are available in ns-3. The first, called `GlobalRouter`, uses static routes precomputed using Dijkstra SPF calculations from globally collected link information. The second implements the Optimized Link State Routing (OLSR) protocol for dynamic ad-hoc networks.

Applications Four `Application`s are available in ns-3.4. These are installed in `Node`s and started/stopped at specific times in the simulation. An application can create one or more `Socket`s of different protocol types. Using these sockets, packets can be sent to other destination sockets, at which the packet reception is delivered to an application via attached socket callbacks.

The `OnOffApplication` generates constant bit rate (CBR) traffic to a single destination according to an alternating *on*/*off* pattern. *On*/*Off* state durations are determined by two `RandomVariable`s. During *on* state, packets of one configured size are generated at a defined data rate. The packets are sent *after* a delay of packet-size divided by data-rate. The duration of *off* state has no effect on the packet rate in the *on* state.

At the destination of an `OnOffApplication`'s packet stream, a `PacketSink` application is commonly used to receive packets. Incoming packets can be processed by a trace callback and are subsequently discarded. Requested TCP connection are accepted and all received data handled like incoming packets.

The two classes `UdpEchoClient` and `UdpEchoServer` form a pair of applications, which send UDP packets back and forth. The `UdpEchoClient` generates UDP packets at a configured rate and sends them to a destination. At that destination a `UdpEchoServer` can be used to echo the packet back to the sender.

`V4Ping` is both a ping program and a ping reflector. It constructs ICMPv4 ping-request packets and sends them to a destination host. If it receives a ping-request, it will answer with a ping-echo ICMP packet. Using this exchange, the ping round trip time (RTT) between hosts can be calculated.

Helper Purpose of classes from the *helper* module is to make building complex simulation scenarios easier. As it does not define a specific networking layer it stands aside of the previous components, but contains utilities referencing each of them.

An example use case of the helper API is to create 20 nodes, add a TCP/IP stack and a wifi net device to each of them. This can be done in about ten lines of code, without losing the option to modify individual simulation attributes.

Starting point of the *helper* setup is the `NodeContainer` class, which keeps track of a list of `Node`s.

Each of the network devices listed in the *devices* module has an associated helper class, which can be used to create objects in batch mode. For example applying `CsmaHelper::Install()` to a `NodeContainer` creates a `CsmaNetDevice` on each node. The created `NetDevice`s can in turn be grouped into a `NetDeviceContainer` for further batch management.

By using the `InternetStackHelper`, a complete TCP/IP protocol stack can be added to a set of nodes with one line of code. Set up of IPv4 addresses can be automated using `Ipv4AddressHelper`, which will allocate addresses from configured subnets.

Similarly, routing protocols and mobility models can easily be aggregated to nodes using appropriate helpers.

Creating `Application`s is also simplified by different helpers like `OnOffHelper`, `PacketSinkHelper` and `PacketSocketHelper`. These helpers create an `ApplicationContainer` set, which can be configured to start and stop in batch.

Contrib All contributed code not yet incorporated into the main tree is collected in the *contrib* modules. It contains code that possibly will become a part of the main tree, if it proves useful enough. In ns-3.4 the *contrib* model contains following utilities.

The author of this thesis contributed a greatly enhanced collection of `Gnuplot` classes, which make generation of gnuplot scripts and datasets from within simulations very easy. Both 2d and 3d plots are supported. Most plots in this thesis were created using these classes.

A larger framework for collecting and processing statistics from simulation runs is currently in `contrib/stats`. Its goals is to flexibly set up simulation control mechanisms, while simultaneously recording and evaluating statistics from ns-3 simulations. Collecting of events is closely integrated into the ns-3 tracing system. This enables online calculation of statistical values and makes memory intensive writing of trace logs needless.

A class called `ConfigStore` can, in conjunction with `FileConfig`, be used to conveniently configure attributes of simulated objects. Configuration values can be loaded from plain text files or special XML documents. There is also a simple, preliminary gtk+ interface available to browse and set configuration values with a graphical user interface.

4.2 Architecture of 802.11 Implementation

In this section an overview of the `WifiNetDevice` architecture is given. It implements 802.11 wireless LAN and is currently the most complex network device in ns-3.

The implementation of 802.11 was ported to ns-3 from yans [20]. Yans (Yet Another Network Simulator) was a prototype project by Mathieu Lacage and Tom Henderson. It was developed in conjunction with the 802.11 PHY/MAC model, which was originally destined for ns-2. Many features and design issues were tested in yans and resulted in ns-3's architecture.

The *wifi* component of ns-3 contains a lot of modules and subclasses, in ns-3.4 their total count is 75. A very condensed summary of the wifi architecture of ns-3 is shown in figure 4.3. The `WifiNetDevice` is built up of a number of classes coordinating packet transmission and reception.

In the figure the path a packet takes when traveling through 802.11 device and channel is marked by the solid lines with arrows. Other dashed lines indicate functional class relationships, however, many dependencies and class associations are omitted. The following discussion will focus primarily on a single packet's journey through the architecture. To decrease complexity, many intermediate class bindings have been omitted from the discussion.

A new packet enters the wifi layers from upper layers via `WifiNetDevice`'s generic interface. The packet is augmented with a standard LLC and SNAP header and sent to `MacHigh`.

The abstract class `MacHigh` (actually named `WifiMac`) is responsible for high level MAC management functions like probing and AP association. There are currently three classes derived from `MacHigh`, `AdhocWifiMac`, `NqstaWifiMac` and `NqapWifiMac`, each providing different management capabilities. `AdhocWifiMac` has no management coordination for ad-hoc networks. `NqstaWifiMac` represents non-QoS wireless STAs, which send probes and attempt to associate with an AP. APs are simulated by `NqapWifiMac` and control a set of associated stations. When attached to another network, the AP can forward packets to and from the distribution system (DS). In section 7.2, a new QoS-enabled implementation called `QosAdhocWifiMac` is added to support EDCA.

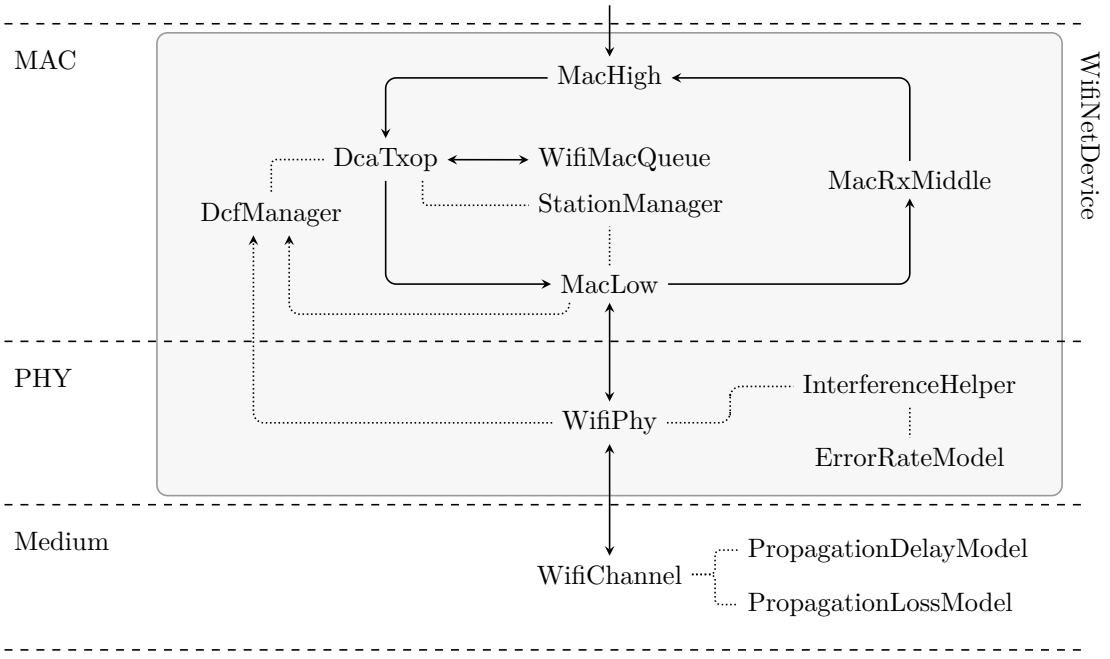


Figure 4.3: ns-3 802.11 wifi module architecture

All three currently implemented `MacHigh`s forward incoming data packets to a single `DcaTxop`. Together with `DcfManager`, the `DcaTxop` implements 802.11 DCF, as the names suggest. `DcaTxop` contains the backoff counter and requests access to the medium from `DcfManager`. When granted, it transmits the packet via `MacLow`. Prior to transmission, the packet is tagged with a sequence number generated by `MacTxMiddle` (not shown in the figure) and possibly fragmented. Transmission parameters as fragmentation and RTS/CTS exchange threshold are determined by the associated `StationManager`.

`DcaTxop` holds one "current" packet until it is acknowledged. If further packets are received from `MacHigh`, they are stored in a `WifiMacQueue`.

All transmission parameters like payload data rate, RTS/CTS exchange, short and long retransmission counters are controlled by `StationManager`. Each `WifiNetDevice` contains only one `StationManager` instance, which gets signaled about each correct and incorrect packet transmission or reception. Based on this information, payload data rate is determined according to different rate control algorithms. In ns-3.4 the rate control algorithms ARF, AARF, AMRR, Onoe and RRAA are implemented together with a constant and an "ideal" rate manager. Transmission parameters are requested by `DcaTxop` and `MacLow` when needed.

`DcfManager` controls multiple `DcaTxop`s and grants access to the medium according to DCF rules (see section 2.4.4). The manager processes both physical carrier sense from `WifiPhy` and virtual carrier sense indications from `MacLow`. Accordingly backoff is started and postponed in the associated `DcaTxop`s.

Data packets are pushed to `MacLow` for transmission. Depending on transmission parameters, `MacLow` initiates a RTS/CTS exchange prior to sending data and waits for an ACK if required. It signals completion to both `DcaTxop` and `StationManager`. Retransmission is not handled by `MacLow` due to the backoff required.

The class `WifiPhy` models the wireless transceiver. In ns-3.4 there is one implementation of `WifiPhy`: `YansWifiPhy`, which models an additive white Gaussian noise (AWGN) channel with cumulative noise using `InterferenceHelper`. In `InterferenceHelper` all incoming packets are recorded, whether they can be correctly received or not.

Radio signal transmission is modeled by sending a packet onto a `WifiChannel`, to which multiple other `WifiPhy` are connected. `WifiChannel` delays the incoming packet according to `PropagationDelayModel` and then calls all attached `WifiPhy`s. Reception power is modeled by `PropagationLossModel` and the calculated value is handed up to `WifiPhy`.

Depending on the indicated reception power, `WifiPhy` determines whether it can correctly receive the packet. This decision is based on `ErrorRateModel`'s emulation of channel characteristics. These are discussed in detail in section 6.6.

Correctly received control packets are handled by `MacLow`. If the packet is a RTS, then `MacLow` will automatically send a CTS after SIFS. Data packets and management frames are forwarded upwards to `MacRxMiddle`.

At `MacRxMiddle` the frame sequence number is checked and duplicate packets are eliminated. Furthermore `MacRxMiddle` reassembles fragmented packets and pushes complete ones upwards to `MacHigh`.

`MacHigh` handles management frames like probes or association requests according to provided services. All data packets are forwarded up to higher protocol layers.

In this thesis the PHY layer of ns-3 is enhanced and compared to the corresponding implementation in ns-2. Furthermore EDCA is implemented and validated by extending `DcaTxop` and `DcfManager`.

Part II

Enhancements

Chapter 5

Propagation Model Enhancements

In wireless network simulations maybe the most complex subproblem is emulating the propagation of radio waves. Much work has been done to grasp the characteristics of antennas, radio wave propagation and transceivers. Both analytical and empirical approaches are common and lead to different models.

The problem is a very basic one: how does a receiver experience the signal transmitted by a sender. In this chapter, only aspects of wave propagation are discussed, signal and packet reception criteria are the topic of following chapters.

Radio signal propagation determines when and with what reception power a signal arrives at a receiver. For this purpose a wireless simulator requires a model emulating physical effects. This medium model can become very complex depending on the level of detail targeted. Due to the irregular, chaotic nature of signal propagation, the level of realism can be varied greatly. Ultimately the employed model granularity must be adapted to an experiment's needs.

However, selection of an appropriate propagation model is challenging due to the many factors which influence signal propagation. Moreover, the model's properties has decisive impact on a simulation's outcome and may mean the difference between success and failure.

Depending on a model's granularity different effects are taken into account. Signal propagation is highly dependent on the environment, which may change constantly. To make experimental results comparable, reference scenarios must be designed carefully. Again, the scenario or environment modeling can be done with different levels of detail. Simple models may completely ignore mobility and simulate a static or momentary static world. Other more elaborate models can take sender and receiver motion into account and emulate effects of their movement.

A high level of detail can be achieved with ray tracing of the signal, its diffraction and reflexions. This implies a highly detailed model of the environment and may also take motion into account. However, bit- and even packet-level signal ray tracing are both computationally expensive and only usable for small experiment setups.

Greater abstraction is gained by using probabilistic models, which are based on empirical measurements or analytic considerations. Environmental influences and mobility are approximated in simulation using pseudo-random values generated with specially crafted probability distributions. Depending on the desired environment characteristics, different probability distributions and parameters can be used to estimate reception signal strength. Computationally this is much less intensive than ray tracing, but makes experiments depend on random values.

Simpler than probabilistic models are deterministic propagation formulas. The reception power is derived by an equation based on an analytic environment model. This model can range from an empty free-space world to include multi-ray effects with directional motion.

In ns-3, three deterministic propagation loss models and one with random start values were available. For this thesis a true probabilistic model was ported from ns-2. All these models are discussed and compared in the following two sections. Per packet only one reception power is calculated for the whole signal duration. Section 6.8 discusses the consequences of this approach and how packet reception is determined.

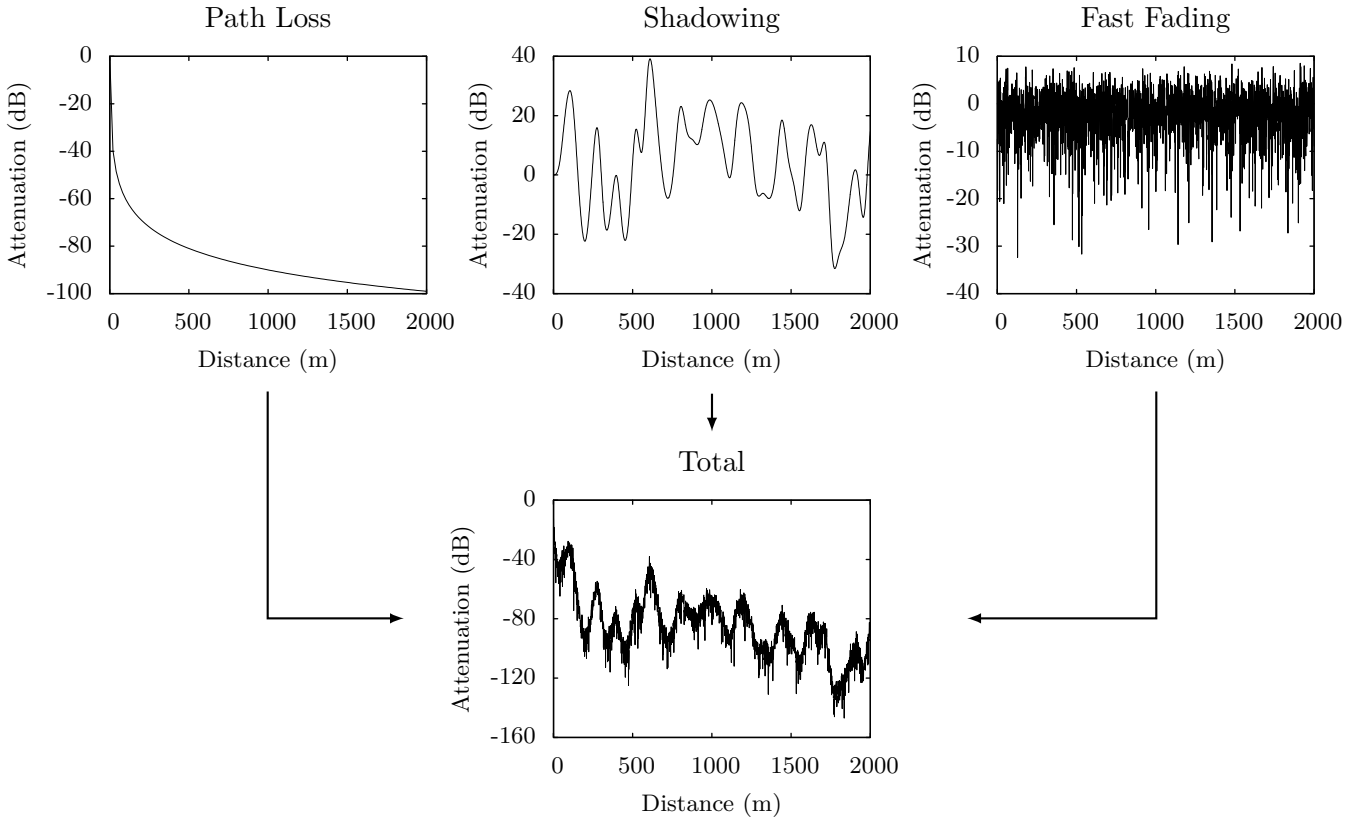


Figure 5.1: Composition of different scale propagation loss effects

5.1 Propagation in ns-3

Propagation in ns-3 is computed within the `WifiChannel` class. Packets are pushed to the `WifiChannel` from connected `WifiPhy` objects as shown in figure 5.2. In ns-3.4 there is only one implementation of both classes: `YansWifiChannel` and `YansWifiPhy`. In this thesis a second pair is added, called `Ns2ExtWifiChannel` and `Ns2ExtWifiPhy`, which provide equal characteristics as ns-2's extended implementation. Both channel implementations, `YansWifiChannel` and `Ns2ExtWifiChannel`, are identical and are only distinguished because of C++ type checking.

Packets are pushed to `WifiChannel` together with their transmission power (txPower). For each attached, receiving `WifiPhy`, signal delay and attenuation is calculated in associated classes `PropagationDelayModel` and `PropagationLossModel`. Most experiments will used a propagation delay model called `ConstantSpeedPropagationDelayModel`, which calculates the delay proportional to distance d: $t_{\text{delay}} = \frac{d}{c}$ where $c = 300 \times 10^6 \frac{\text{m}}{\text{s}}$.

The distance of sender and receiver is represented in ns-3 using `MobilityModel`. As the name suggests, this class describes mobility of a `Node` and different motion patterns are provided by derived subclasses. For signal propagation modeling, the current position and velocity can be requested as three dimensional vectors.

The deterministic and probabilistic propagation loss models in ns-3 can be classified by the environmental effects they capture (see figure 5.1). The most basic signal attenuation, caused by the distance between sender and receiver, is called *path loss*. Second largest effect in scale is called *shadowing*, which arises from superposition of reflections from obstacles and other large-scale objects. Typically signal variation due to *shadowing* occurs in ranges of ten to hundred meters. However, also small distance changes on the scale of wavelengths can have great influences on signal reception. These are represented using *fast fading* probabilistic propagation loss models, which are generally distance-invariant but time-variant.

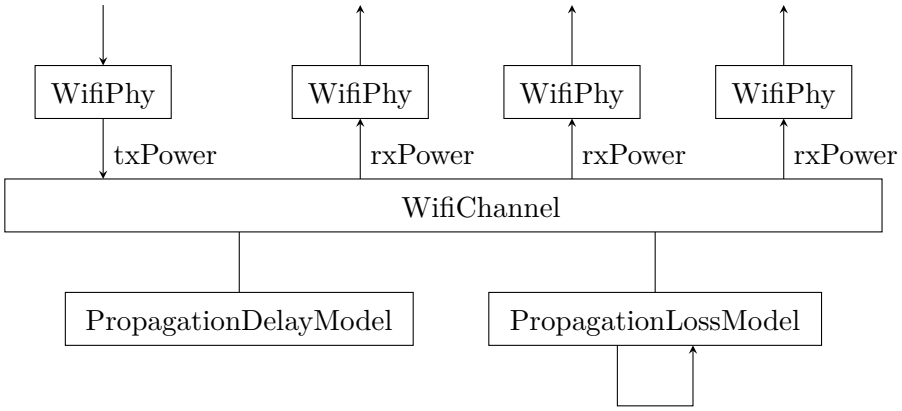


Figure 5.2: Propagation modeling with `WifiChannel`

5.2 Basic Propagation Loss Models

In ns-3.4 five propagation loss models are available. These models can be chained to form a composite model emulating multiple effects. Because currently no shadowing model exists in ns-3, the most common composition is a path loss model followed by a fast fading model.

Three different path loss models are available. The simplest, `FriisPropagationLossModel`, computes the free-space attenuation after Friis [7] using a modernized equation [29, p. 107]:

$$\frac{P_r}{P_t} = \frac{G_t G_r \lambda^2}{(4\pi d)^2 L} \tag{5.1}$$

where P_r is reception and P_t transmission power in watt, G_t and G_r dimensionless transmission and reception antenna factors, L a general dimensionless system loss coefficient, λ the wavelength and d the distance in meters.

This equation can be solved by d to calculate distance depending on all other parameters.

$$d = \frac{\lambda}{4\pi}\sqrt{\frac{P_t G_t G_r}{P_r L}} \tag{5.2}$$

Some values of d are calculated for common WLAN frequencies, transmission and reception powers in table 5.1. The coefficients L, G_t, G_r are all set to 1. λ is determined by $\frac{c}{f}$ with f being different WLAN frequencies and $c = 300 \times 10^6 \frac{\mathrm{m}}{\mathrm{s}}$ the speed of light.

The second propagation loss model available in ns-3 is called `LogDistancePropagationLossModel` and implements path loss using the following equation 5.3 [29, p. 138]. The log-distance path loss model is usually expressed in dB or dBm (see section A.1 for a note on decibel) and depends on an exponent parameter. This parameter can be determined for different environments by empirical measurements. The equation also requires a reference distance and path loss at that position; in ns-3 the free-space path loss at 1 m with 5.15 GHz (equaling −46.67 dB) is used by default. With this reference loss model and exponent 2 the log-distance model is equivalent to the free-space model.

$$L = L_0 + 10 \cdot n \cdot \log_{10}\left(\frac{d}{d_0}\right) \tag{5.3}$$

where L is the resulting path loss in decibel, n the distance exponent, d the transmission and d_0 the reference distance in meters and L_0 the loss at d_0 in decibel.

Similarly the log-distance equation can be solved by d to calculate ranges for specific reception power thresholds. Log-distance ranges are tabulated together with free-space ranges for the same parameters in table 5.1.

For this thesis a third path loss model was added to ns-3, named `ThreeLogDistancePropagationLossModel`, and is already available in the official ns-3.4 release. It is a variant of the log-distance path loss model with three distance fields. In each field a different path loss exponent is used.

P_t	P_r	Free-space range with $\lambda = \frac{c}{f}$ at			Log-distance range with				
					2.54 GHz		5.15 GHz		
		2.54 GHz	5.15 GHz	5.9 GHz	$n = 2$	$n = 2.5$	$n = 2$	$n = 2.5$	$n = 3$
5 dBm	-82 dBm	210 m	104 m	91 m	210 m	72 m	104 m	41 m	22 m
10 dBm	-82 dBm	374 m	185 m	161 m	374 m	114 m	185 m	65 m	32 m
20 dBm	-82 dBm	1183 m	584 m	509 m	1183 m	287 m	584 m	163 m	70 m
5 dBm	-94 dBm	838 m	413 m	361 m	838 m	218 m	413 m	124 m	55 m
10 dBm	-94 dBm	1490 m	735 m	641 m	1490 m	346 m	735 m	196 m	81 m
20 dBm	-94 dBm	4711 m	2323 m	2028 m	4711 m	868 m	2323 m	493 m	175 m

Table 5.1: Free-space and log-distance reception range for common parameters

This model is based on the code contributed to ns-2 by the DSN: the contributed Nakagami propagation model contains this three-field as path loss model, to which a Nakagami-shaped fast fading loss is added. For ns-3 these two combined models were separated and can be used and configured individually. Nakagami propagation loss modeling is discussed in the next section.

The three-log-distance model has three distance fields: near, middle and far. Actually a fourth interval is also defined from 0 to the first reference distance. In this invalid field the attenuation is set to 0. The distance boundaries are named as described in following sequence.

$$\underbrace{0 \cdots\cdots}_{=0} \underbrace{d_0 \cdots\cdots}_{n_0} \underbrace{d_1 \cdots\cdots}_{n_1} \underbrace{d_2 \cdots\cdots}_{n_2} \infty$$

In each field, the log-distance equation 5.3 is used with a different exponent n_i. Each field's reference loss distance (d_0 in equation 5.3) is set to the end of the previous field and reference loss calculated accordingly. Through this composition method, the complete equation 5.4 forms a continuous function of d.

$$L = \begin{cases} 0 & d < d_0 \\ L_0 + 10 \cdot n_0 \log_{10}(\frac{d}{d_0}) & d_0 \leq d < d_1 \\ L_0 + 10 \cdot n_0 \log_{10}(\frac{d_1}{d_0}) + 10 \cdot n_1 \log_{10}(\frac{d}{d_1}) & d_1 \leq d < d_2 \\ L_0 + 10 \cdot n_0 \log_{10}(\frac{d_1}{d_0}) + 10 \cdot n_1 \log_{10}(\frac{d_2}{d_1}) + 10 \cdot n_2 \log_{10}(\frac{d}{d_2}) & d_2 \leq d \end{cases} \tag{5.4}$$

with d propagation distance in meters, d_0, d_1, d_2 the three distance field boundaries in meters, n_0, n_1, n_2 the dimensionless path loss exponents of the three valid fields and L_0 the propagation loss in decibel of the reference model at d_0.

For ns-3.4 the default values of ns-2 were adopted: $d_0 = 1\,\text{m}$, $d_1 = 200\,\text{m}$, $d_2 = 500\,\text{m}$, $n_0 = 1.9$, $n_1 = 3.8$, $n_2 = 3.8$ and reference loss $L_0 = 46.67\,\text{dB}$, which is the free-space path loss at 1 m with 5.15 GHz. All parameters are exported by the class as attributes and are easily changed using ns-3 configuration tools.

In figure 5.3 the reception power of the three deterministic path loss models are plotted for 20 dBm transmission power. Each model is plotted with default parameters as defined by the model's implementation. For `LogDistancePropagationLossModel` a second plot with exponent 2.5 is included. And for `ThreeLogDistancePropagationLossModel` a second data line with exponents 1.0, 3.0 and 10.0 was added to highlight the three distance fields $1\,\text{m} \cdots 200\,\text{m} \cdots 500\,\text{m} \cdots \infty$.

5.3 Further Models in ns-3.4

Two further propagation loss models are included in ns-3.4: `RandomPropagationLossModel` and `JakesPropagationLossModel`.

Propagation loss in `RandomPropagationLossModel` is determined by a random number generator following a specific random distribution. In conjunction with one of ns-3's random distribution `ExponentialVariable`, `NormalVariable` or `WeibullVariable` this model can be used to add probabilistic attenuation. However,

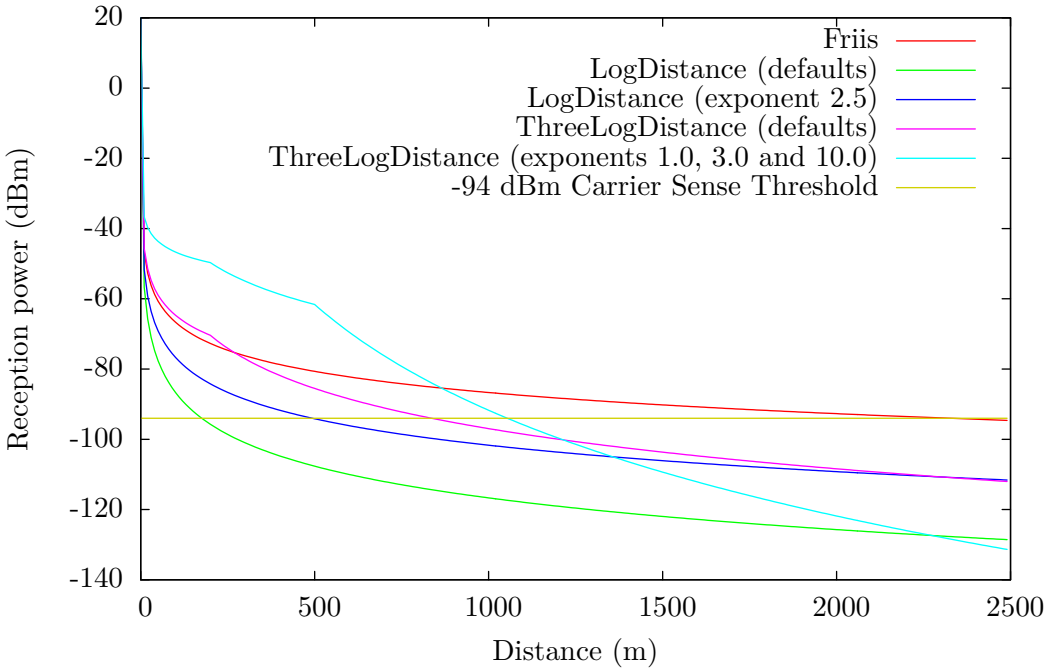


Figure 5.3: Three deterministic propagation loss models in ns-3 (transmission power 20 dBm)

Rayleigh and Nakagami fast fading cannot be represented with this class; the new `NakagamiPropagationLossModel` should be used.

`JakesPropagationLossModel` is named after William Jakes, who described it in a textbook on mobile communications [18]. It approximates a Rayleigh fading channel by summing sinoids representing multiple rays arriving at the receiver. The sinoids' initial phase is randomly distributed and advances with simulated time. When calculating the superposition of all sinoids, a predefined Doppler shift is added to the phases to simulate node movement.

These two models are not in the scope of this thesis and only mentioned for the sake of completeness.

5.4 Nakagami-m Fast Fading

To lay an equal basis for later work, the propagation loss models of ns-2 and ns-3 were compared. In this context the Nakagami propagation model, contributed to ns-2 by the DSN, was ported to ns-3 and verified.

Nakagami fast fading is a modification of the standard Rayleigh fast fading model, described in many standard textbooks on wireless communication [9, 29, 33]. Both are statistical models of time-variant, multi-path signal fading, which describes the superposition effect of multiple radio propagation rays reaching a receiver by different paths. Rayleigh fast fading describes a model in which an infinitely large number of independent rays reach the receiver. Due to the central limit theorem their amplitude and phase can be regarded as normally distributed. The signal envelope is then the norm of a complex Gaussian random variable (both components independent normal variables). This envelope follows the Rayleigh probability distribution, which can be defined as

$$p_{\text{Ray}}(x;\omega) = \frac{2x}{\omega} e^{-\frac{x^2}{\omega}} \tag{5.5}$$

where $\omega > 0$ is a spread parameter. For signal strength calculations, the spread parameter is set to the average receive power in watt before fast fading.

Nakagami fading is a more general fast fading model, which includes Rayleigh fast fading as a special case. It can be tuned with a shape parameter m to better approximate propagation properties of different environments. In a channel with Nakagami fading properties, the signal envelope follows the Nakagami-m

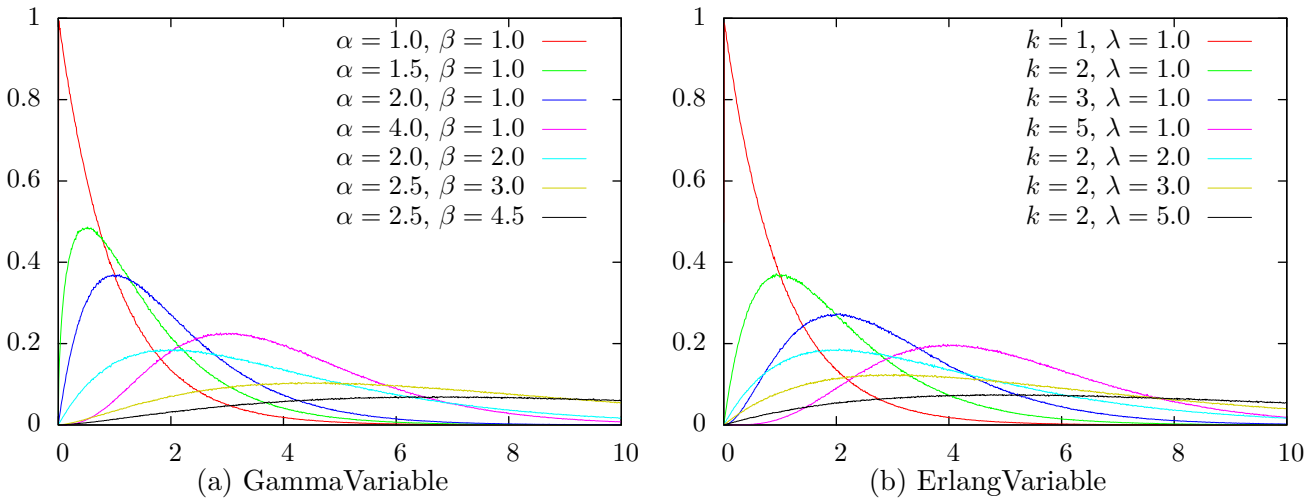


Figure 5.4: Histograms of new ns-3 random distributions

distribution, which has the following probability density function.

$$p_{\text{Nak}}(x; m, \omega) = 2x^{2m-1} \frac{m^m e^{-\frac{m}{\omega}x^2}}{\omega^m \Gamma(m)} \tag{5.6}$$

where $m \geq 0.5$ is the shape, $\omega > 0$ a spread parameter, and $\Gamma(x) = \int_0^\infty t^{x-1} e^{-t}\,\mathrm{d}t$ the gamma function. For $m = 1$ the Nakagami distribution is equal to the Rayleigh probability distribution.

The Nakagami distribution is closely related to the gamma distribution, which is typically defined with the following probability density function. Other definitions with different parameter names are also common and equivalent.

$$p_{\text{Gamma}}(x; \alpha, \beta) = \frac{x^{\alpha-1} e^{-\frac{x}{\beta}}}{\beta^\alpha \Gamma(\alpha)} \tag{5.7}$$

with $\alpha > 0$ a shape, $\beta > 0$ a scale parameter and $\Gamma(\cdot)$ the gamma function. For integer values of α the $\Gamma(n)$ function is equal to the factorial $(n-1)!$ and in this special case the gamma distribution is also called Erlang distribution. This random distribution is equal to the sum of k independent exponentially distributed random variables and has the probability density function

$$p_{\text{Erlang}}(x; k, \lambda) = \frac{x^{k-1} e^{-\frac{x}{\lambda}}}{\lambda^k (k-1)!} \tag{5.8}$$

with $k > 0$ an integer shape parameter and $\lambda > 0$ a rate parameter. Obviously equation 5.8 is equal to equation 5.7 for $k = \alpha$ and $\lambda = \beta$. Both gamma and Erlang distributions are plotted in figure 5.4 for some example parameters.

Simulation of Nakagami fast fading focuses on generating random samples depending on the average reception power before fading. Reception power after fast fading is derived from equation 5.6 by substituting variables $x \to x^2$ and thus the square root of a Nakagami random variable has following density function, which can be represented using the gamma distribution:

$$p_{\text{Nak}}(x^2; m, \omega) = x^{m-1} \left(\frac{m}{\omega}\right)^m \frac{e^{-\frac{m}{\omega}x}}{\Gamma(m)} = p_{\text{Gamma}}\left(x; m, \frac{\omega}{m}\right) \tag{5.9}$$

Note that during the variable substitution, $\mathrm{d}x^2 = 2x$ is divided from the probability density function, to keep the function's area normalized to 1. So it is possible to simulate Nakagami-m fast fading channel properties using gamma variates generated with the parameters above.

For this purpose both gamma and Erlang distribution generators were ported from ns-2 to ns-3 and are available as `GammaVariable` and `ErlangVariable`. The core code of the gamma variate generator in ns-2

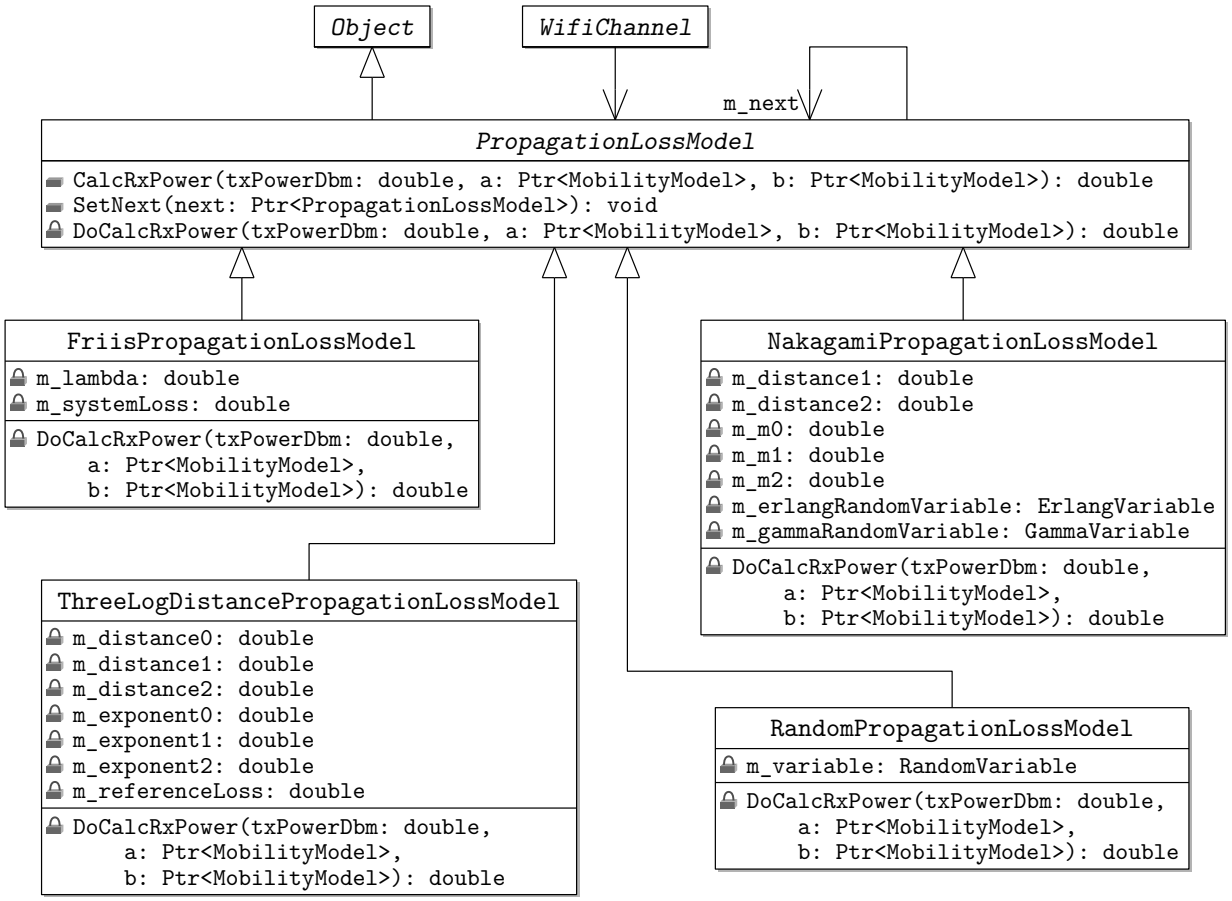


Figure 5.5: UML diagram of propagation loss mode classes

and ns-3 is based on a fast algorithm devised by Marsaglia and Tsang [23], which is based on only normal and uniform random variates. Because the Erlang distribution is a special case of the gamma distribution, `GammaVariable` can be used instead of `ErlangVariable`. However, a variate of an Erlang distribution can be generated much faster using k exponential variates than using the more general gamma algorithm.

The histogram plots in figure 5.4 were generating using ns-3's implementation of both random distribution generators. Their shape match the theoretical curves exactly.

5.5 Implementation and Verification

As described in the last section, two new random distributions were added to ns-3 to simulate Nakagami fast fading. A new propagation loss model class, called `NakagamiPropagationLossModel`, was added and implemented using the two new `RandomVariables`. The Unified Modeling Language (UML) diagram in figure 5.5 shows the new propagation loss model in context with some of the others previously mentioned.

For compatibility with the ns-2 implementation, `NakagamiPropagationLossModel` also supports different m parameters depending on distance. To match ns-2, three distance fields are used with three m parameters as follows

$$\underbrace{0 \cdots\cdots}_{m_0} \underbrace{d_1 \cdots\cdots}_{m_1} \underbrace{d_2 \cdots\cdots}_{m_2} \infty$$

The default values of range boundaries and m parameters were retained from ns-2. They are $d_1 = 80\,\mathrm{m}$, $d_2 = 200\,\mathrm{m}$, $m_0 = 1.5$ and $m_1 = m_2 = 0.7$.

Within these fields, fast fading is calculated using `GammaVariable` or `ErlangVariable` according to equation 5.9 with ω the original reception power in watt.

The UML diagram in figure 5.5 shows how `PropagationLossModel` classes work together. The wifi channel object contains an association to only one propagation loss object and calls `CalcRxPower()` with the original

transmission power and two node mobility models. Within `CalcRxPower()` the virtual function `DoCalcRxPower()` is called to perform the first model's calculations. Afterwards, if `m_next` is defined, the chained propagation loss model is invoked with the current model's results. This way each propagation loss model can add any kind of propagation loss (not only additive loss) as done by the Nakagami model.

By chaining `ThreeLogDistancePropagationLossModel` and `NakagamiPropagationLossModel` a propagation loss model can be created which is equivalent to Nakagami propagation in ns-2.

To verify the implementations of `ThreeLogDistancePropagationLossModel`, `GammaVariable`, `ErlangVariable` and `NakagamiPropagationLossModel` in ns-3, a simulation scenario was created with ns-2.33 and all Nakagami propagation loss values saved directly from the function `Nakagami::Pr()` in `ns2/mobile/nakagami.cc`. The simulation scenario was a simple two nodes experiment and is described in section 6.7.1. The taken Nakagami probes are plotted as a histogram in figure 5.7, which will be explained together with a corresponding one from ns-3.

In ns-3 direct access to the propagation loss models is easier than in ns-2, because no complete simulation needs to be built. A sample program (`main-propagation-loss.cc`) was contributed to ns-3, which plots all propagation loss models with default and modified parameters. It is meant to be a reference for wireless simulation users. In all plots a transmission power of 20 dBm $\hat{=}$ 0.1 mW is used.

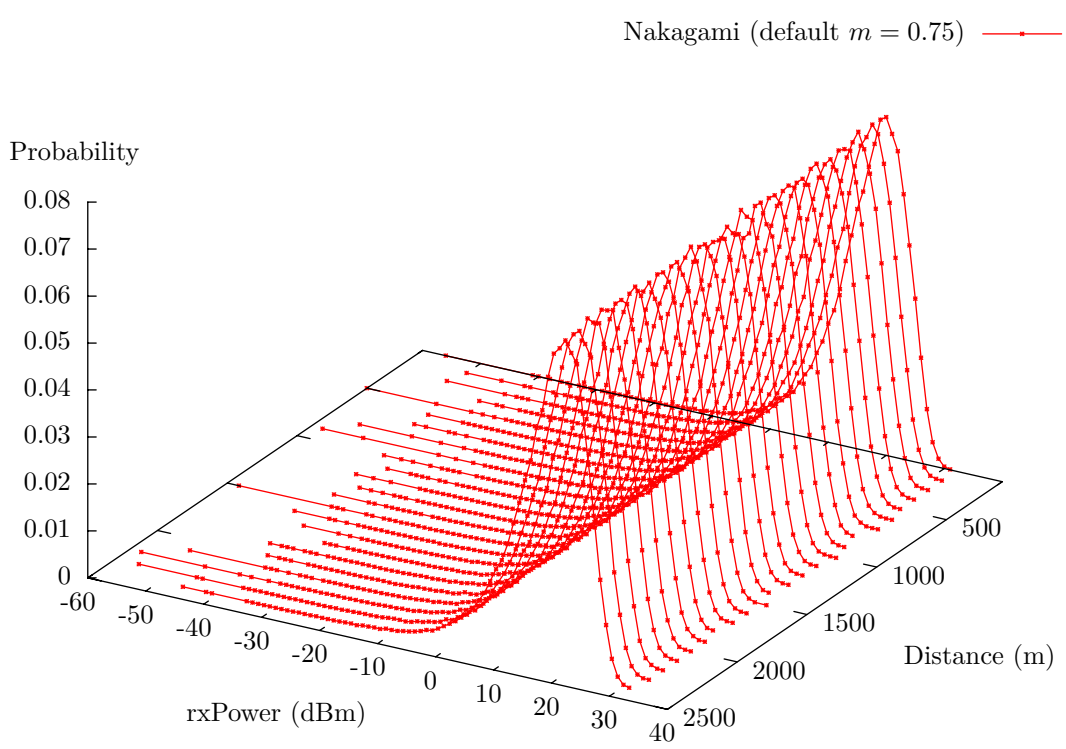
Figure 5.6(a) shows a first histogram of the reception power returned by an instance of `NakagamiPropagationLossModel` with default parameters as tested by the sampling program. In the second figure 5.6(b) `ThreeLogDistancePropagationLossModel` is chained with `NakagamiPropagationLossModel`.

All three figures 5.6(a), 5.6(b) and 5.7 show probability histograms as a 3D plot with distance and power as x and y axis. For each node distance 100 – 2 500 m with 100 m steps the reception power distribution is plotted as a curve slice. Thus the area below each distance slice is 1.

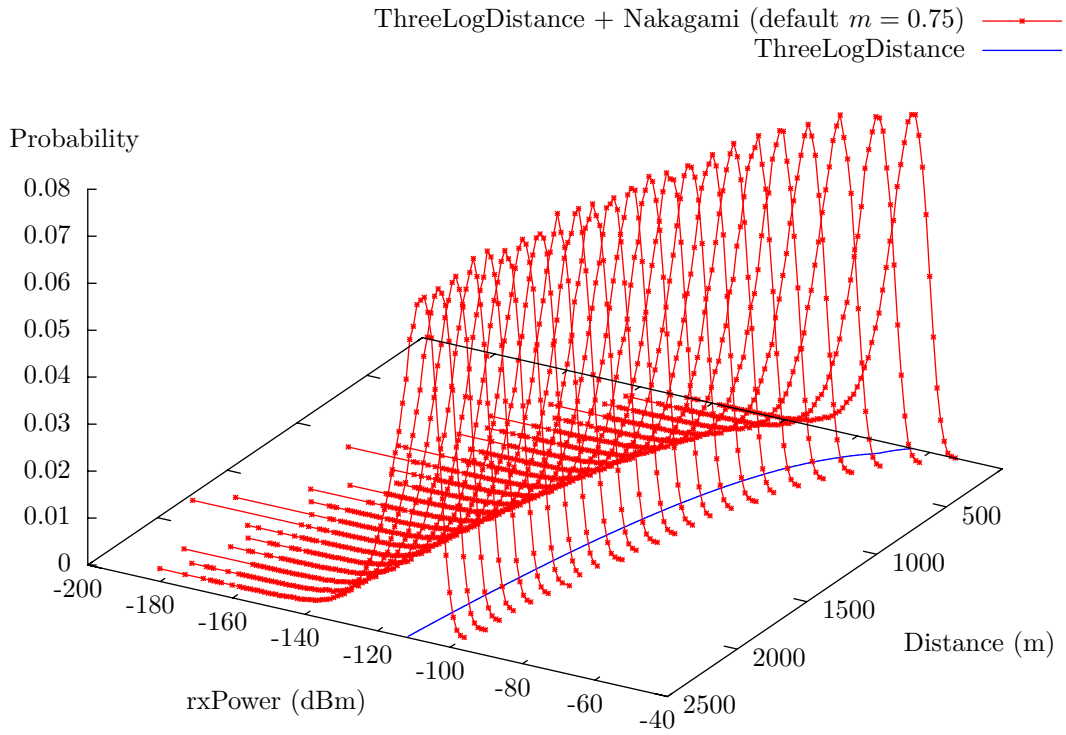
Note that in the plain `NakagamiPropagationLossModel` diagram in figure 5.6(a), all curve slices are equal, because Nakagami fast fading does not vary with distance. When path loss is added in figure 5.6(b) the skewed bell curves move to smaller reception power values. The probability bell curves follow the deterministic three-field log-distance path loss curve shown at the plot's base. This three-field log-distance curve is identical to the corresponding one in figure 5.3.

Comparison of figure 5.7 and figure 5.6(b) and further verifications during implementation yield conclusive evidence that the Nakagami propagation model in ns-3 gives results equal to ns-2.

All 3D plots show only the Nakagami curves for the default $m = 0.75$ parameter. The power probability curves of some further, common m parameters are plotted in figure 5.8. For $m = 1.0$ the curve represents Rayleigh fast fading power probability. Higher m parameters allow less power spread and thus will decrease outlier packet reception probability.



(a) Only `NakagamiPropagationLossModel` with 20 dBm transmission power



(b) `ThreeLogDistancePropagationLossModel` and `NakagamiPropagationLossModel` chained with 20 dBm transmission power.

Figure 5.6: Nakagami propagation loss model in ns-3

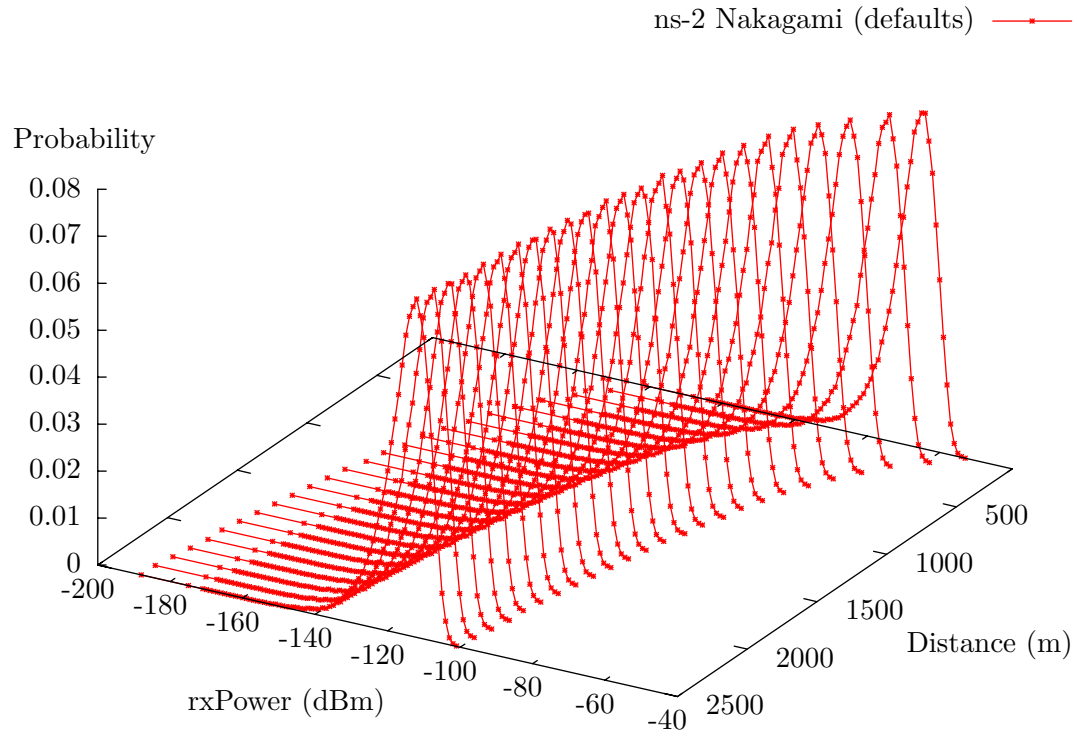


Figure 5.7: Reception power histogramm from ns-2 Nakagami propagation model

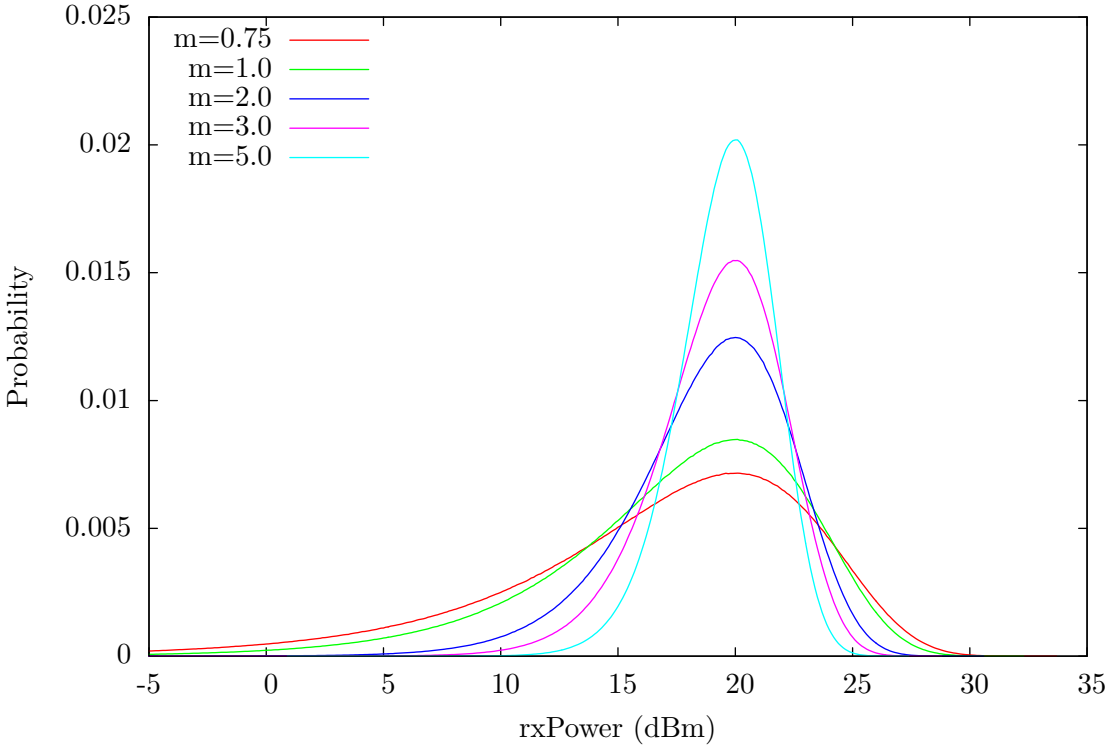


Figure 5.8: Nakagami-m reception power distribution for different m parameters

Chapter 6

PHY Layer Enhancements

In the last chapter propagation loss models in ns-2 and ns-3 were compared. They are the basis for modeling a radio channel, which is used as transmission medium in wireless networks. This chapter discusses how this radio channel is modeled in ns-2 and ns-3. Finally, enhancements made to ns-3's model are described and verified against simulations in ns-2.

Medium layer (radio channel) and physical layer (802.11 PHY) are very challenging to model in a discrete event packet simulator. Radio channel and transceiver module are part of the physical world and thus do not follow a set of strict logic rules like a protocol or network application. Behavior of the radio channel is generally grasped using a simplified mathematical model and statistical methods are employing to simulate different propagation effects. Likewise, the radio chipset's signal reception and decoding mechanisms are captured by models with different degrees of detail. For purposes of packet simulation, the employed models are sufficiently simplified to allow viable implementations and fast simulations.

Radio propagation is modeled by the medium layer, which describes a radio communication channel. This channel can have different properties depending on surrounding environment. Channel properties are represented in ns-2 and ns-3 using propagation delay and loss models as described in section 5. Unlike reality, the channel model in ns-2/3 currently does not exhibit interference or other multi-packet effects, instead it boils down to a packet distribution class, which calculates reception power for each wireless receiver independent from other packets.

Emulation of interference is left to PHY layer models, which represent the radio transceiver in a wireless LAN card. The 802.11 wireless LAN defines multiple PHY layers (see section 2.3), which all have different transmission methods and even different media. Different from a real 802.11 chip set, in ns-2/3 the PHY layer model ultimately determines whether a packet was received correctly. In modeling this reception decision, the range of transceiver characteristics that can be taken into account is very large. These characteristics are not part of the 802.11 standard, but differ between wireless chipsets and determine their quality.

The PHY layer implementation of ns-2.33 (`WirelessPhyExt`) and ns-3.4 both currently contain cumulative noise and interference modeling. ns-2 contains a SINR-based reception criterion and supports the frame capture effect. In this chapter these ns-2 implementations are compared to ns-3's 802.11 code, which simulates reception using a BER/PER criterion and lacks frame capture. For this thesis, the SINR reception criterion and frame capture effect were ported to ns-3.

6.1 Modeling the Transceiver

As mentioned in this chapter's introduction, packet interference modeling is not handled by the channel implementation, but left to the PHY layer code.

The foundation of all interference calculations is the additive white Gaussian noise (AWGN) channel. It adds a Gaussian distributed noise level to all signals. By augmenting this very basic channel model with the propagation loss models from section 5, different environments can be simulated. When multiple signals are transmitted simultaneously, their interference must be calculated for correct reception modeling.

In the discrete event simulators ns-2/3, the continuous statistical noise level of a AWGN channel is modeled using a constant noise floor level. In ns-3 this noise floor is calculated from thermal noise at 290 K depending on channel bandwidth. For comparison with ns-2, code was added to set the noise floor to a configurable constant value. In all following experiments a constant noise floor of -99 dBm is used.

For each packet sent on the channel, the propagation loss model calculates one reception signal strength from the original transmission power. Without interference the reception power remains constant for the complete duration of the packet. This is a great simplification compared to real radio channels, which will vary constantly under influence of noise and fluctuation of transmission or reception quality. In discrete-event simulations these effects must be emulated using statistical propagation loss models.

When two packets are transmitted simultaneously, they will interfere at a common receiver. See figure 6.1 for a quantitative diagram of two packets as received at one listening node. Diagram (a) shows the constant noise floor, in diagrams (b) and (c) two packet signals are illustrated as experienced individually at the receiver. The total power on the channel received by the listener is shown in diagram (d): the power of both signals and noise are added up as superposition of all radio waves.

If the destination wants to receive signal A, other signals must be considered as interfering noise. This cumulative noise, relative to signal A, consists of the noise floor and signal B as shown in figure 6.1(e). Vice versa for signal B, the cumulative noise is the sum of noise floor and signal A.

From noise, interference and signal power the signal to interference and noise ratio (SINR), sometimes also called SNIR, can be calculated. It is an important quantity describing signal strength and used in reception criteria. The SINR for signal i is defined as

$$\mathrm{SINR}_i = \frac{P_i}{N_f + \sum_{j \neq i} P_j} \tag{6.1}$$

for N_f the noise floor in watt and P_i the power of each simultaneously received signal in watt. Note that SINR is a dimensionless ratio of two power quantities. Usually the SINR is given in dB (see section A.1). Even though the quotient in equation 6.1 can never become negative, its value in dB is negative for ratios below 1.

In figure 6.1(g) the SINR values of both signals are plotted in dB. During interference of both signals, the SINR values drop below 0 dB and transmitted bits will probably be received incorrectly.

6.2 Implementation of Cumulative Noise

Cumulative noise and SINR calculations, as explained in the previous section, are implemented in both ns-2 by the DSN and in ns-3. Both implementations were analyzed carefully and despite using different methods were determined to deliver equivalent results.

To implement cumulative noise in discrete-event simulators, the power determined at each packet's arrival event must be simulated to extend for the signal's complete duration. From these packet indication events the instantaneous total power on the channel must be calculated.

In ns-2.33 the component named `PowerMonitor` is used to keep track of all packet signals currently received. This is done using a single powerLevel variable, which is initialized with the noise floor. At each packet's arrival time the reception power is added to powerLevel. The corresponding subtraction is done by an event scheduled to occur at the packet signal's end time. Thus powerLevel contains the instantaneous power on the channel.

In ns-3 cumulative noise is handled by the `InterferenceHelper` class. For each received packet signal an entry in a list is created. The entry contains arrival time $t_{\mathrm{start},i}$, end time $t_{\mathrm{end},i}$, reception power P_i, packet size and more parameters. The list is sorted by start time. See figure 6.2 for an illustration of the list, with each rectangle representing a packet entry.

From this list the instantaneous interference can be calculated by summing all entries overlapping the current time. However, for the bit error rate (BER)/packet error rate (PER) reception criterion used in ns-3, the list structure can also be queried for all interference *changes* during a packet's transmission interval. Between

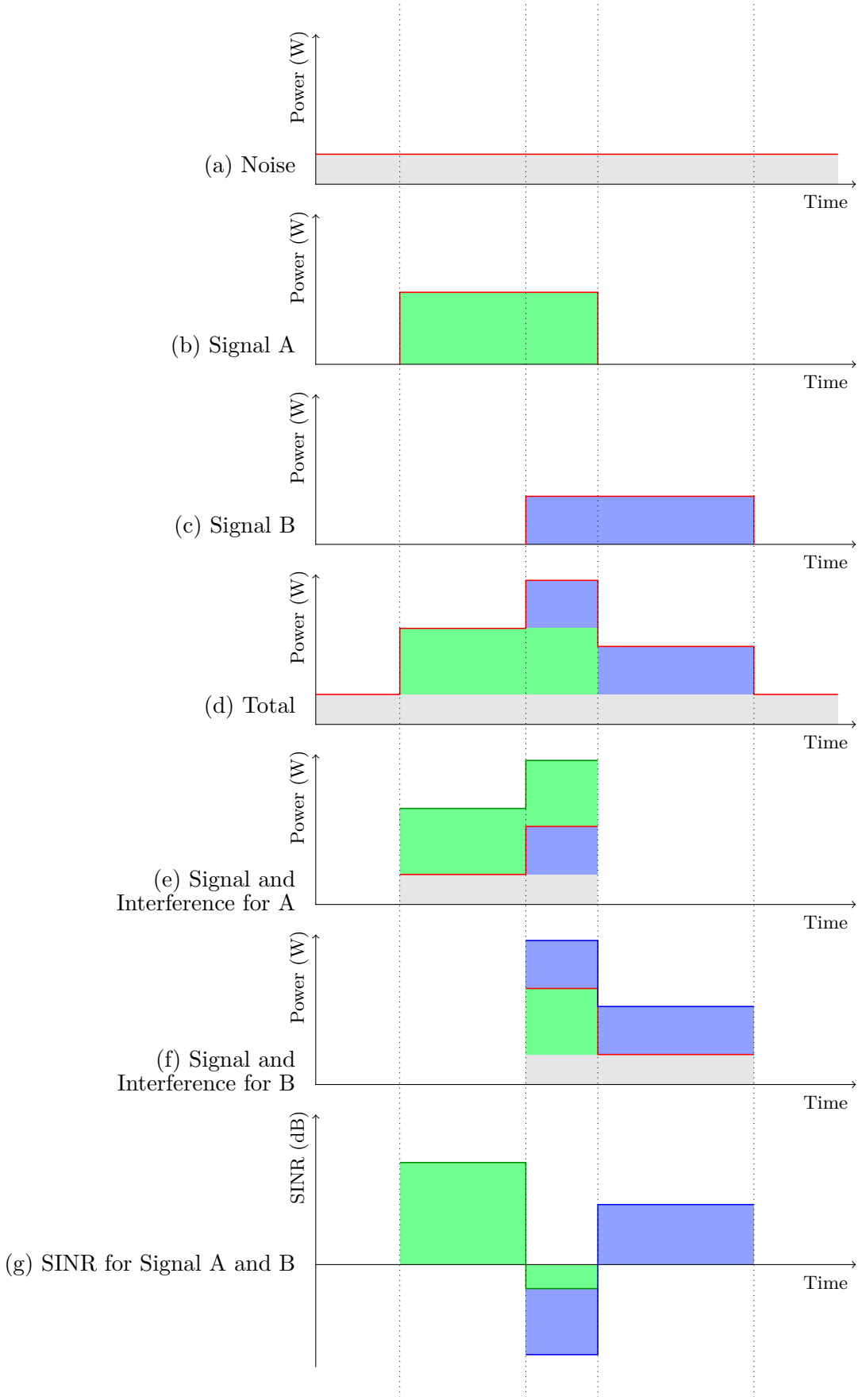


Figure 6.1: Noise, signal, interference and SINR

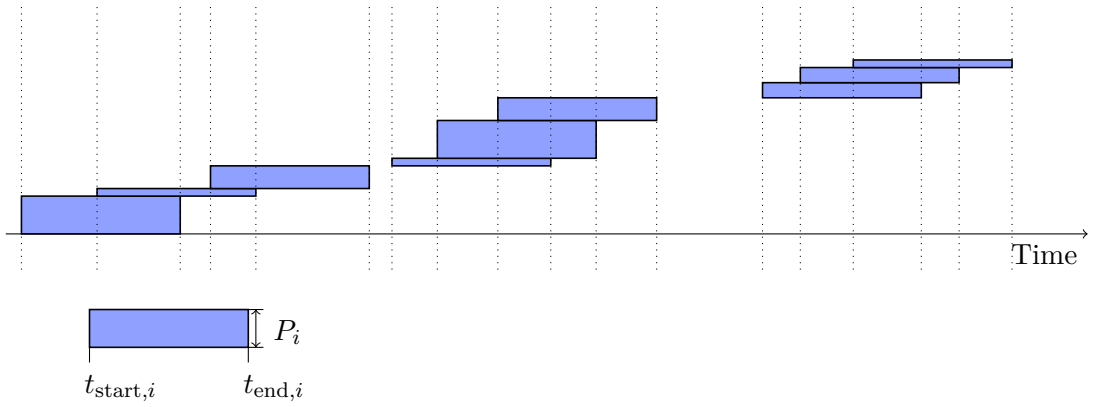


Figure 6.2: ns-3 `InterferenceHelper` event list

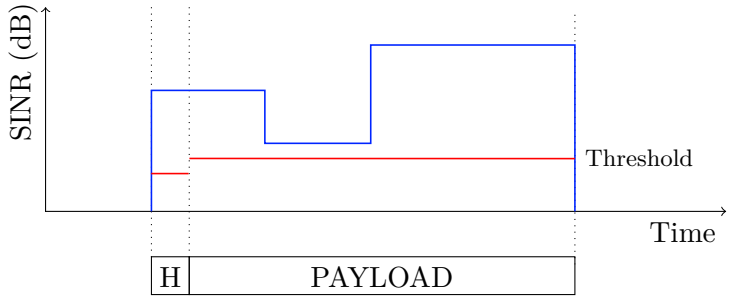


Figure 6.3: SINR threshold reception criterion

each change, the interfering power level remains constant. How BER/PER values are calculated for these intervals is explained in section 6.6. As simulated time progresses, entries no longer necessary are removed from the list.

6.3 SINR Reception Criterion

The `WirelessPhyExt` implemented in ns-2 by the DSN uses a SINR threshold criterion to determine packet reception. This criterion emulates verification of the CRC32 checksum in the trailer of each 802.11 frame. Only by computing this checksum can a wireless receiver determine whether the packet was correctly received.

The SINR reception criterion states that a packet is correctly received if its SINR value remains above a configured threshold for the complete packet duration (see figure 6.3). This threshold depends on the wireless modulation mode employed, faster modulations require higher SINRs for correct reception. Since header and payload of a 802.11 frame are encoded using different modulation schemes, different SINR thresholds apply to header and payload. The thresholds used in ns-2.33 for four basic 802.11a modulation schemes (see section 2.3.2) are shown in table 6.1.

During reception of a frame, the SINR value may change due to interference. In ns-2/3 discrete-event

Keying	Defaults in ns-2.33	Updated in ns-3
BPSK	4 dB	5 dB
QPSK	7 dB	8 dB
16-QAM	12 dB	15 dB
64-QAM	20 dB	25 dB

Table 6.1: ns-2 SINR thresholds and new ns-3 values

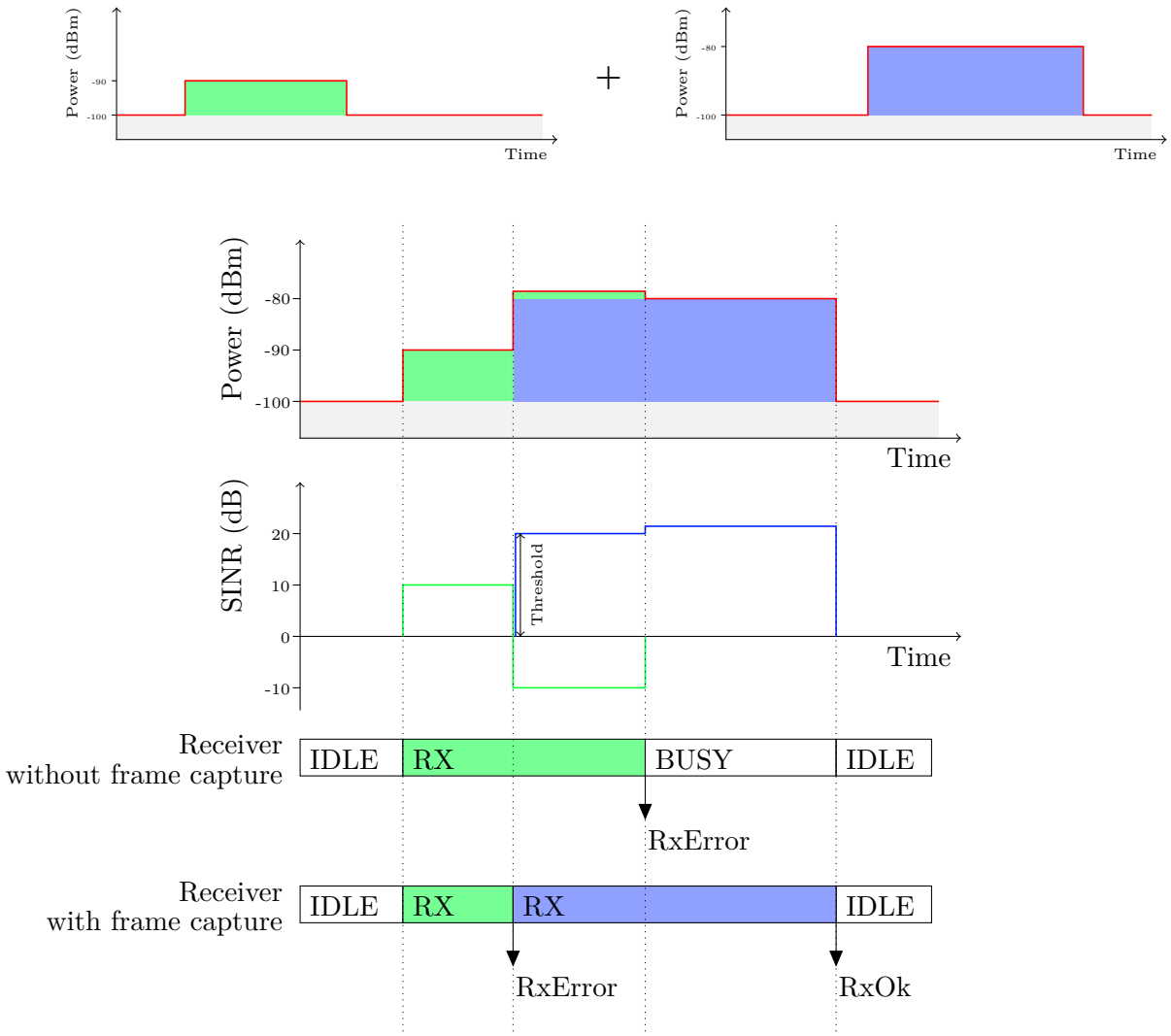


Figure 6.4: Capture effect with two packets

simulations these changes are always marked by the beginning or ending of a simultaneously transmitted signal. At each such time point the SINR must be recalculated and compared to the current threshold.

6.4 Frame Capture Effect

The so called capture effect is a feature of modern wireless chipsets to “switch” to a stronger signal during the reception phase of a weaker frame [32].

This situation occurs when the transceiver is receiving a relatively weak signal, which is subsequently jammed by a stronger frame’s signal as illustrated in figure 6.4. Due to the additional interference, the SINR of the first packet drops and correct reception will fail.

If the receiver does not support frame capture, it will stay tuned to the weak signal and determines at the frame’s end that the received data is corrupt. Since only one signal can be received simultaneously, the stronger frame is lost.

A more intelligent receiver supporting frame capture determines that reception is impossible and indicates an error immediately. Instead of continuing to listen to the weak signal, it will tune to the stronger frame and correctly receive it.

The decision whether to switch signals is determined by a frame capture policy. In the ns-2 implementation, two conditions must hold for frame capture to be triggered. First the SINR of the weaker frame must drop below the specific reception thresholds (table 6.1). Secondly the stronger signal must reach a configurable

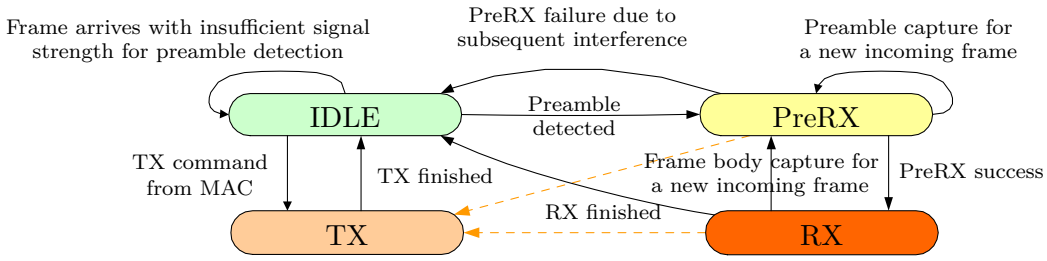


Figure 6.5: ns-2 `WirelessPhyExt` state diagram (adapted from [2])

SINR capture threshold. During preamble reception of the weaker frame, this SINR threshold is by default 5 dB and during payload reception 10 dB. The first case is also called "preamble capture" and the second case "data capture", thereby describing during which phase the weaker packet was discarded. Note that interference in the SINR value of the stronger signal contains the power of the weaker packet.

For ns-2, the name "preamble" capture is not correct, because the default duration value in ns-2.33 also includes the PLCP header (see figure 2.2). Strictly speaking this is not part of the OFDM preamble, and thus the case should be called "header" capture. For sake of recognition, the ns-2 nomenclature was retained.

6.5 Implementation Issues

Implementation of the SINR reception criterion is based on cumulative noise calculation as described in section 6.2. From cumulative noise and the synchronized signal power level, the SINR can be calculated. For successful reception, the SINR must remain above a rate-dependent threshold for the whole packet's duration.

In discrete event simulation as ns-2/3, SINR of a received packet only changes at interfering packets' beginning or end time points. Because the received packet's SINR can only become lower at interfering packet arrival times, packet end time points need not be considered.

6.5.1 ns-2 Implementation

In ns-2, the `WirelessPhyExt` class function `sendUp()` is called for each packet arrival event. At these time points, the simulated PHY state may change according to the state machine in figure 6.5.

When the PHY is idle, reception of a new incoming packet may commence if the calculated SINR is above the configured header reception threshold. In this case, the PreRX state is entered and an event is scheduled to occur after the frame header's duration. This event handler checks that the SINR of the frame payload is above the payload reception threshold, which is possibly higher due to a faster modulation rate. If successful, the RX state is entered and a finishing event is scheduled. This finishing event checks whether the packet was flagged as faulty and either discards or passes the packet up to higher layers.

As required by 802.11 half-duplex transceivers, during packet transmission in the TX state no packet may be received.

To correctly implement the SINR reception criterion, changes in SINR levels must be checked. Because changes can only occur when an additional signal is detected, SINR checks can be done in the `sendUp()` function. These additional events occur during the waiting intervals between the transition events of the usual IDLE → PreRX, PreRX → RX, RX → IDLE sequence. If the PHY's state is PreRX or RX, the current packet's SINR level must be rechecked to exceed the corresponding threshold in presence of the additional interfering signal. If the threshold is not reached, the incoming packet is marked with an error flag.

Frame capture is implemented in the state machine as changes to the usual event sequence. If an additional frame is received during the PreRX state, preamble frame capture may be triggered if the incoming frame's SINR reaches the required value. However, switching to the new packet can only occur if the currently received frame's SINR drops below the required reception threshold. Similarly, in RX state a new signal may trigger data capture if its SINR reaches the required level and the currently received packet drops below

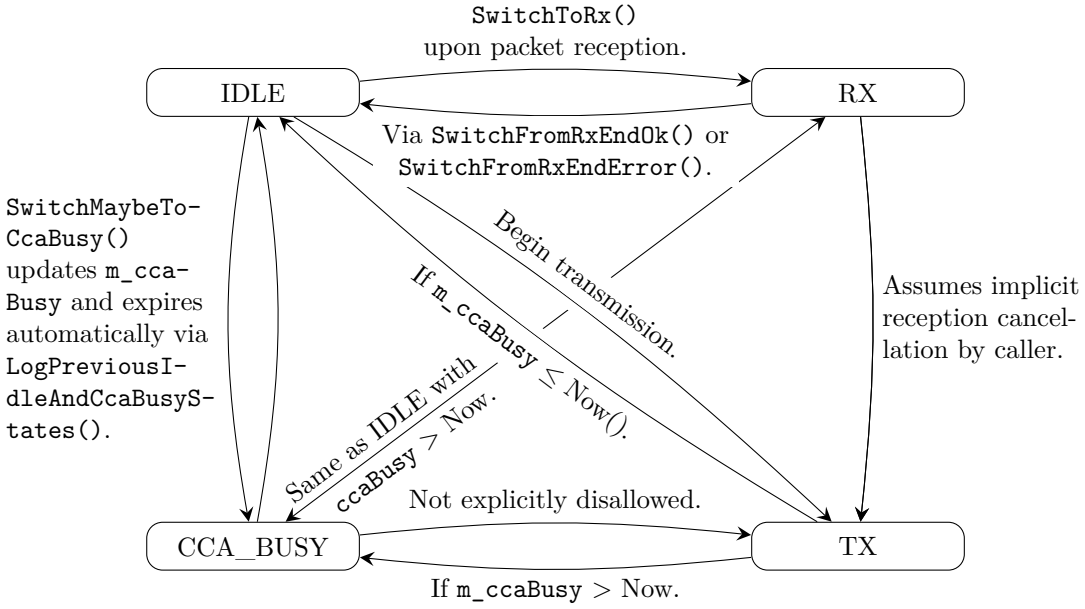


Figure 6.6: State transitions of `WifiStateHelper`

the acceptable reception threshold. If packet capture is triggered, the state switches back to PreRX with a new current packet.

The ns-2 implementation uses a strict state machine implementation following the corresponding diagram (figure 6.5) and each event transition is described in detailed Specification and Description Language (SDL) diagrams [2].

6.5.2 Porting to ns-3

For this thesis the SINR reception criterion, as implemented in ns-2, was ported to ns-3. Instead of a complete code port, the modules already available in ns-3 were reused to implement a PHY model matching the ns-2 model. Verification of the implementation is documented in section 6.7.

In the following paragraphs, the different submodules of the wifi implementation in ns-3.4 are discussed and how they are used to create a new ns-2 compatible PHY model. The classes of these submodules are illustrated in the UML diagram in figure 6.8. This diagram also contains classes implemented for this thesis that are not included in ns-3.4.

In ns-3.4 only one PHY implementation is available: `YansWifiPhy`. It uses a BER/PER reception criterion, which is described in section 6.6. The main class `YansWifiPhy` is closely dependent on two helper classes for individual emulation aspects: `InterferenceHelper` and `YansWifiStateHelper`.

As described in section 6.2, cumulative noise is managed in `InterferenceHelper` by inserting all received packets as `InterferenceHelper::Event`s into a list. From the list all SINR changes over a packet's reception duration can be calculated. This set of changes is represented as a vector of `InterferenceHelper::NiChange`s, which are used to derive BER/PER values in the calculation functions in `YansErrorRateModel`. The final reception decision is then made by `YansWifiPhy::EndReceivePacket()` based upon the determined PER value.

The state of `YansWifiPhy` is managed by the `YansWifiStateHelper` class. It contains a state machine-like implementation with four states: IDLE, CCA_BUSY, RX and TX. In ns-3.4 the reception state was named SYNC, which was changed to RX. Obviously this would be the place to add a PreRX state to imitate the state machine from ns-2. However, the class does not contain a simple `enum` state variable, instead the state is encoded in two `Time` values `m_endTx`, `m_endCcaBusy` and one boolean `m_rxing`. The current PHY state is determined by checking the time values against current simulation time: if `m_endTx` is in the future, then the state is TX. If it is not in TX and `m_rxing` is true, then the state is RX. If it is not RX or TX, it is either CCA_BUSY or IDLE depending on `m_endCcaBusy`.

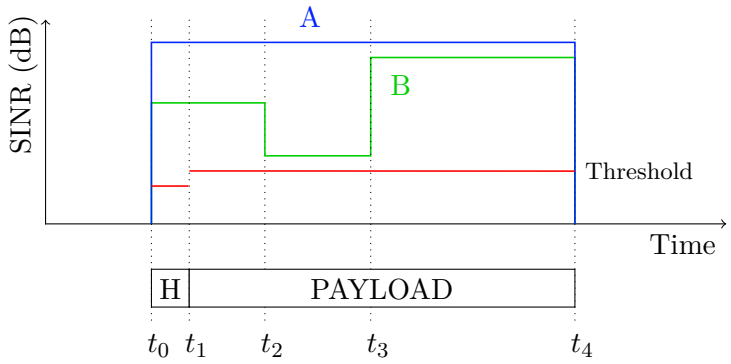


Figure 6.7: SINR check time points

Transitions between these states are done by the various `SwitchTo...()` functions in the `YansWifiStateHelper` class. State switching is done by updating one of the `m_end... Time` variables and `m_rxing`. However, because simulator time progresses outside the scope of the state machine, state changes can happen without an explicit method call simply due to elapsing simulated time. This makes comparison with a traditional `enum` state machine difficult. Nevertheless, creation of a state diagram was attempted in figure 6.6. It shows a diagram of all possible state transitions in `YansWifiStateHelper`. Note that all but one are possible: no direct transition from TX to RX is possible. Transitions to IDLE or CCA_BUSY depend only on the value of `m_ccaBusy`, otherwise they are equal. Due to the implicit state machine, a state transition requires complex maintenance work on the variables to backlog elapsed state switches. Furthermore external trace functions are called to correctly export state switch events.

When comparing the two state diagrams 6.5 and 6.6, the first observation is that there are different state identifiers: ns-2 has the extra PreRX state for preamble and payload distinction. On the other hand ns-3 has an extra CCA_BUSY state. In ns-2 the CCA_BUSY medium indication is not handled as an extra state, because the indication is orthogonal to reception or transmission handling. Instead, the indication is signaled directly by `PowerMonitor` when the channel energy reaches a configured threshold called *carrier sense threshold*.

Due to the complex integration of the implicit state machine with tracing and higher layer signaling, adding of a new PreRX state was avoided. It is also not necessary as will be shown later.

The new PHY model added to ns-3 is called `Ns2ExtWifiPhy` and aims at providing equal behavior as the ns-2 `WirelessPhyExt` class. For this the `YansWifiPhy` was cloned together with the associated `YansWifiChannel`. The new class `Ns2ExtWifiChannel` is identical to the yans version, except that it operates with a list of `Ns2ExtWifiPhy`s.

The SINR reception criterion was implemented in `Ns2ExtWifiPhy` by disabling BER/PER calculations. For this purpose, the function `InterferenceHelper::CalculateSnr()` was added to return the SINR at the current simulated time. To match the yans design, the SINR reception thresholds were implemented by adding a new `ErrorRateModel` class called `Ns2ExtErrorRateModel`, even though no real error rate computation is done. The class contains one function `GetSuccessRate()`, which returns 1 or 0 depending on SINR and wifi mode threshold as defined in the table 6.1. The newer values in this table were taken from an updated patch to ns-2.33 by the DSN.

Frame capture was also implemented in `Ns2ExtWifiPhy`. To the original `YansWifiStateHelper` implicit state machine only one function was added: `SwitchFromRxAbort()` required to abort a packet's reception when required for frame capture.

No extra PreRX state is required to implement SINR reception or frame capture correctly. In ns-2 the only action taken when changing from PreRX to RX is a check, whether the current SINR allows correct packet reception according to the payload data rate threshold. However, this SINR threshold verification can also be done at the end of the packet's duration.

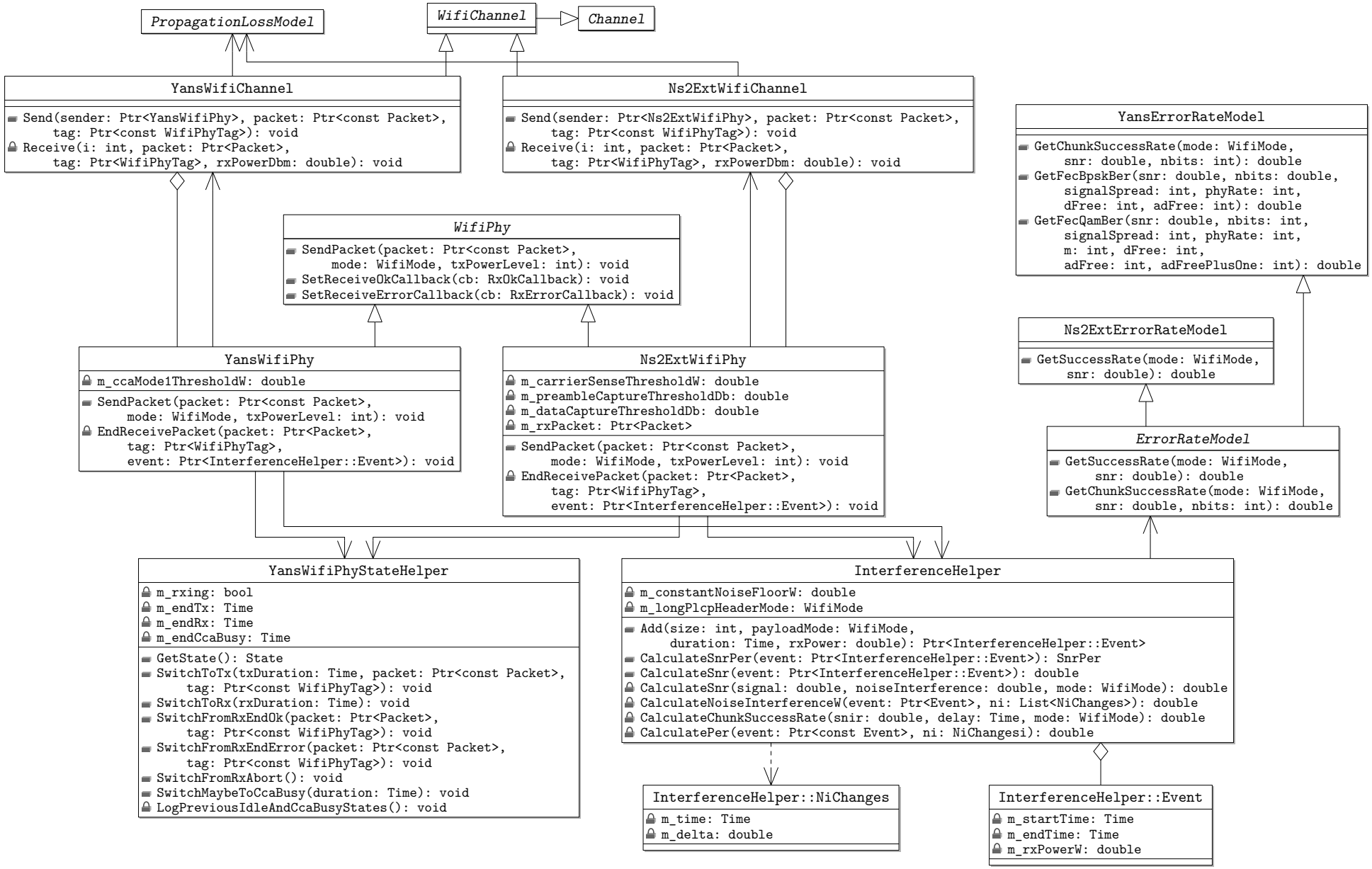


Figure 6.8: UML diagram of `WifiPhy` classes

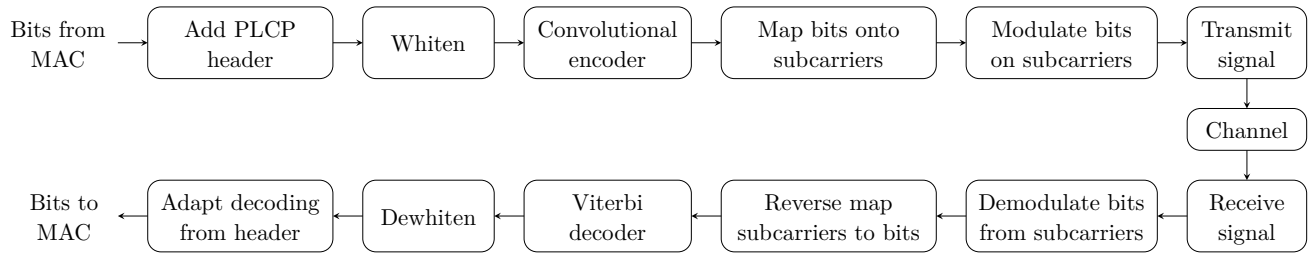


Figure 6.9: Modulation sequence modeled by BER/PER

This is illustrated in figure 6.7: consider packet A received with constant threshold. In ns-2 the extra PreRX verifies the threshold at t_1, whereas in `Ns2ExtWifiPhy` the threshold is checked at t_4. If the SINR threshold fluctuates during reception, as for packet B, then at each time point a new frame starts or ends. For each time point, however, only the new SINR value needs to be checked, because for frame starts, as at t_2, this is always the lower one. For frame ends, as at t_3, the previous lower value was already checked. Thus SINR-based reception can be implemented without PreRX state.

Another distinction is made in ns-2 between RX and PreRX for frame capture: depending on whether the current packet's preamble or payload is being received, two different capture thresholds apply. For this too no extra state is needed, a simple comparison of the packet start time and current time suffices.

The frame capture effect is checked for each new packet arriving from the channel at `Ns2ExtWifiPhy::SendPacket()`. If the currently synchronized packet, stored in `m_rxPacket`, cannot be correctly received due to the reduced SINR level, then the new packet may be switched to. This decision is done as in ns-2 by comparing the new packet SINR with either `m_preambleCaptureThreshold` or `m_dataCaptureThreshold` depending on the discarded packet's reception phase. If capture is triggered, reception of the synchronized packet is immediately aborted and appropriate error trace events are indicated. The state machine is momentarily switched to IDLE using `SwitchFromRxAbort()`. At the same time reception of the new packet is started as usual and the state switched back to RX.

Both SINR threshold and the frame capture effect implemented in `Ns2ExtWifiPhy` are verified in section 6.7. The additional code is being proposed for merge into the next major release of ns-3.

6.6 BER/PER Reception Criterion

To determine whether a packet could be received correctly, `YansWifiPhy`, the default wifi model for 802.11a in ns-3, uses a packet error rate (PER) criterion. This PER attempts to grasp the signal decoding process in a wireless receiver chip using statistical analysis. The PER is built up of more basic modulation specific bit error rate (BER) formulas. Most of this statistical processing is implemented in `YansErrorRateModel`, which was ported to ns-3 from yans (see section 4.2). Currently only 802.11a OFDM wireless modes are supported, work on other modes like DSSS or 802.11b CCK is in progress.

The BER/PER model is described in the corresponding yans paper [20], but also by others authors [21, 38]. The statistical analysis used for 802.11 transmission modes can be found in standard textbooks on wireless communications [9, 28]. This section gives a detailed explanation on the foundations of the BER/PER equations, however, it does not attempt to fully derive each statistical formula, instead textbooks are cited for more detailed review.

The sequence of encoding operations done by the 802.11a PHY layer is illustrated as a block diagram in figure 6.9. To model reception, the BER/PER criterion focuses only on the decoding steps and attempts to model each step separately. Demodulation of OFDM subcarriers, which are each keyed using BPSK, QPSK or M-QAM, yields a bit error rate depending on average received SINR. Once the bits are decoded, the bit interleaving permutation done during decoding must be reversed. The resulting code bits sequence uses a convolutional coding, which is undone, usually using a Viterbi decoder. Convolutional encoding is used by 802.11a to decrease bit error rate by adding redundancy. Finally the data bits must be dewhitened. All the corresponding encoding steps are described in greater detail in section 2.3.2. The focus of this section is on discussing analytic bit error rates and computing a whole packet error rate.

6.6.1 Digital Modulation

Error Functions

In conjunction with error rates, three special error functions are often used for convenience: the $Q(\cdot)$, the $\mathrm{erf}(\cdot)$ "error rate" function and its complement $\mathrm{erfc}(\cdot)$. The $Q(x)$ function is defined as the area right of x below a standard normal (Gaussian) distribution. The $\mathrm{erf}(\cdot)$ and $\mathrm{erfc}(\cdot)$ are closely related to $Q(\cdot)$ and only differ in scale and translation. Both $\mathrm{erf}(\cdot)$ and $\mathrm{erfc}(\cdot)$ are defined by a `math.h` compatible with C99 or POSIX.1-2001 and is available in standard libc libraries.

The following equation table gives a definition of all three functions and shows two relationships.

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-\frac{t^2}{2}} dt \qquad Q(x) = \frac{1}{2} \operatorname{erfc}\left(\frac{x}{\sqrt{2}}\right)$$

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \qquad \operatorname{erfc}(x) = 1 - \operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$$

Energy Per Bit or Symbol

To describe reception probability, some measure of signal quality is required. This section introduces the quantities E_b, E_s, γ_b and $\bar{\gamma}_s$ needed to describe error rates.

In digital modulation schemes, a bit can be encoded by phase shifts or amplitude of the signals. BPSK, QPSK and M-QAM are described in section 2.3.2. The bit error rate for each modulation depends directly on the signal strength, which, in this context, is expressed in energy per bit E_b or energy per symbol E_s. A symbol in 802.11a contains 1, 2, 4 or 6 bits.

Energy per bit E_b and energy per symbol E_s, both measured in Joule, are easily derived from the reception power by

$$E_b = P_r \cdot t_b \qquad E_s = P_r \cdot t_s \tag{6.2}$$

where P_r is the reception power of the complete signal in watt, t_b duration of one bit and t_s duration of one symbol. For 802.11a modulation schemes from table 2.3, the bit and symbol durations can be calculated with

$$t_b = \frac{R}{\mathrm{DataRate}} \qquad t_s = \frac{t_b}{N_{\mathrm{BPSC}}} \tag{6.3}$$

As SINR is more expressive for signal quality of P_r, the so called SINR per bit γ_b and average SINR per symbol $\bar{\gamma}_s$ are more useful for bit error calculations. SINR per bit and symbol are defined as

$$\gamma_b = \frac{E_b}{N_0} \qquad \bar{\gamma}_s = \frac{E_s}{N_0} \tag{6.4}$$

where N_0 is the noise and interference spectral density in watt per hertz.

This can also be expressed using the SINR (equation 6.1) as calculated from P_r by

$$\gamma_b = \mathrm{SINR} \cdot B \cdot t_b \qquad \bar{\gamma}_s = \mathrm{SINR} \cdot B \cdot t_s \tag{6.5}$$

where B is the channel bandwidth in hertz (20 MHz, 10 MHz or 5 MHz in 802.11a).

Bit Error Rate of BPSK, QPSK and M-QAM

Using the quantities for bits and symbols defined in the last section, the bit error rate can be given exactly for BPSK and QPSK. This is done using two conditional Gaussian probability densities over the signal phase shifts and is described in standard textbooks [9, p. 174] [28, p. 255]. The probability of bit error P_b, when receiving a signal modulated with BPSK or QPSK, is

$$P_{b,\mathrm{BPSK}}(\gamma_b) = Q\left(\sqrt{2\gamma_b}\right) = \frac{1}{2} \operatorname{erfc}\left(\sqrt{\gamma_b}\right) \tag{6.6}$$

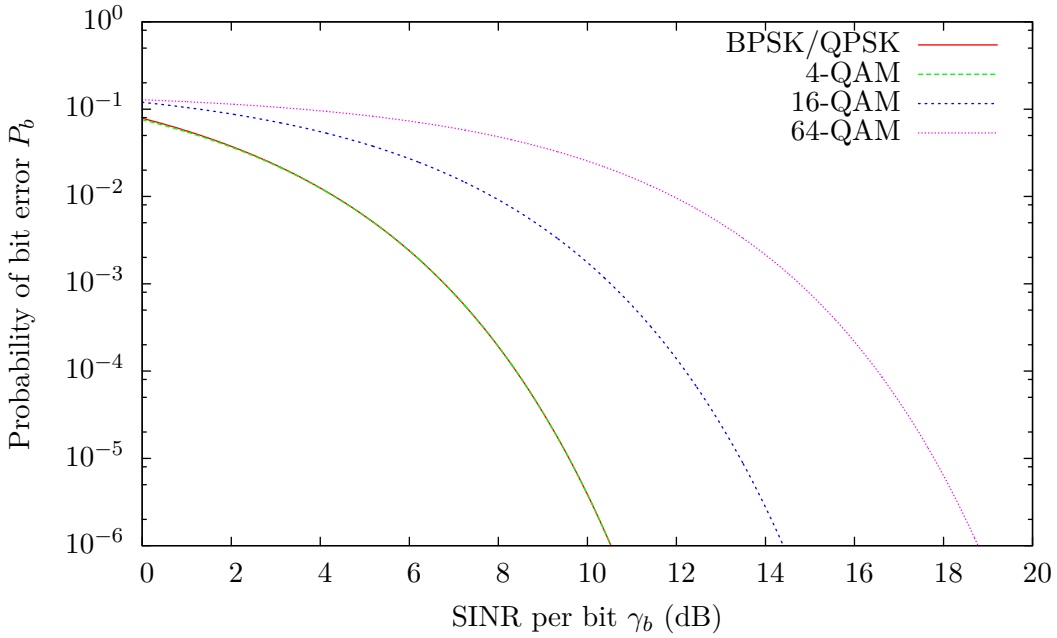


Figure 6.10: BER plot of BPSK, QPSK and M-QAM

Similarly can be done for M-QAM. In M-QAM, $k := \log_2 M$ bits are represented in one symbol by using $M = 2^k$ different phase and amplitude constellations (see figure 2.5). The error probability is derived by regarding each of the M constellation points as independently distributed normal random variables, with special consideration for the outer points [9, p. 177] [28, p. 259]. The probability of an error in a received symbol P_s is exactly

$$P_{s,M\text{-QAM}}(\bar{\gamma}_s) = 1 - \left[1 - 2\left(1 - \frac{1}{\sqrt{M}}\right) \cdot Q\left(\sqrt{\frac{3\,\bar{\gamma}_s}{M-1}}\right)\right]^2 \tag{6.7}$$

Using the equation $\bar{\gamma}_s = k \cdot \gamma_b$ and dividing by k, the corresponding the bit error probability P_b of a M-QAM encoded bit can be given as

$$P_{b,M\text{-QAM}} \approx \frac{1}{\log_2 M}\left(1 - \left[1 - 2\left(1 - \frac{1}{\sqrt{M}}\right) \cdot Q\left(\sqrt{\frac{3 \cdot \gamma_b \cdot \log_2 M}{M-1}}\right)\right]^2\right) \tag{6.8}$$

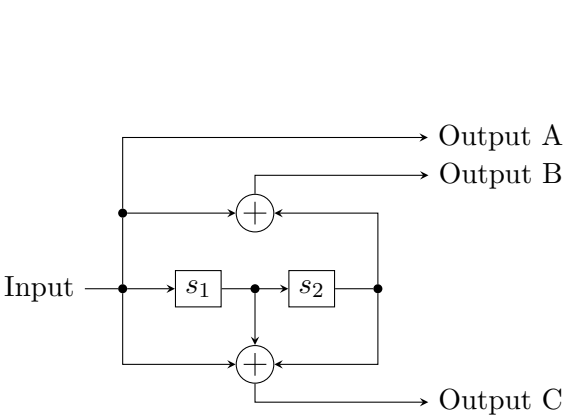
This equation is not exact due to possible bit errors which occur at non-neighboring constellation points. However, this omitted probability is negligible small.

The bit error rate curves of BPSK, QPSK and some M-QAM binary digit modulations are shown in figure 6.10. Note that BPSK, QPSK and 4-QAM have equal BER.

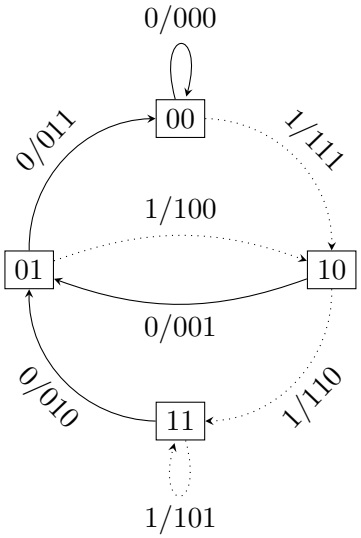
6.6.2 Convolutional Decoding

In the 802.11a decoding sequence, the coded bits are detected from symbol signals keyed using BPSK, QPSK or M-QAM. After detecting the bit sequence, the interleaving permutation map applied during encoding must be reversed. This has no effect on the bit error rate, because each bit only changes place in the sequence. However, since signal detection errors are not uniformly distributed but occur in close proximity, this bit swapping may have an effect on the total PER. Currently this effect is not modeled in ns-3.

The bit sequence is encoded using the convolutional coding defined in section 2.3.2 and must be decoding accordingly. Most wireless chips will use a Viterbi decoder for this purpose. For the purpose of error rate calculations, convolutional decoding is now described short.



(a) Linear feedback shift register



(b) State machine

Figure 6.11: Diagrams of a simple convolution code

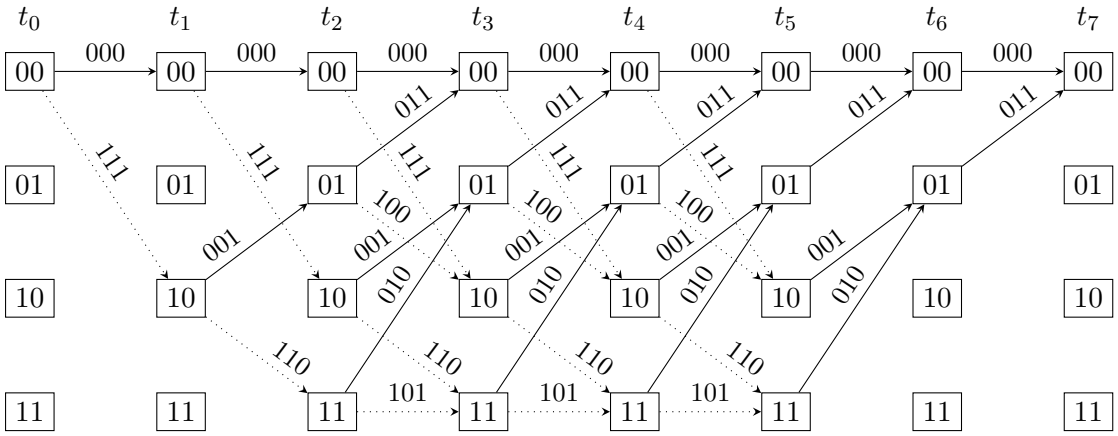


Figure 6.12: Trellis diagram of the simple convolution code

Each convolutional encoder has three basic parameters: k the number of input bits shifted at once, K the number of shift stages and n the number of output bits per shift. For each output bit a function of current register states must be given, often using binary or octal notation.

The 802.11 convolution encoder processes $k = 1$ input bits each shift, has $K = 7$ stages (of which only six are shown in figure 2.4) and $n = 2$ output bit lines. The state machine of the encoder contains $2^{K-1} = 2^6 = 64$ states and is shown in the appendix figure B.1.

For purpose of this quick review of convolutional error rates, a simpler encoder with $k = 1$, $K = 3$ and $n = 3$ is shown as linear shift register and Mealy state machine in figure 6.11 and as a trellis diagram in figure 6.12. The same example is used in many textbooks [9, 28]. The output generator functions are $[100]_2$, $[101]_2$ and $[111]_2$ in binary representation. In the state machine and trellis diagram the state nodes are each labeled with the current contents of the two shift registers. A solid graph edge represents a state transition caused by a zero input bit and a dotted line a one input bit.

To understand decoding of convolutional codes, the trellis diagram is the most useful representation. In figure 6.12 all possible valid encodings of five input bits are shown with two tail bits. Initially the registers are set to zero, and thus the encoder is in 00 state. The five input bits are read from t_0 until t_5 with $5 \cdot 3 = 15$ output bits. Convolutional code input sequences, as in 802.11a, are padded with zero *tail bits*, which are used by the decoder to correctly return to the "all-zero" state. This is done by adding two zero tail bits, which are encoded in t_5 to t_7. The 802.11a convolutional encoder requires six tail bits as shown in figure 2.2.

So the simple convolution code produces in total 21 output bits from 5 input bits and 2 tail bits. Thus there are only $2^5 = 32$ valid bit sequences out of the 2^{21} possible output bit sequences. For convolutional decoding the received bit sequence's "closest" valid sequence is determined and thereby bit errors corrected. This decoding strategy is called *maximum likelihood* detection and is optimal, because it views the complete bit sequence and compares it to all possible valid original sequences. During comparison two distance metrics are commonly used: the Hamming (bit-error) distance and Euclidean distance of two vectors. Depending on the metric a convolutional decoder is called a *hard-decision* decoder for Hamming distances or *soft-decision* decoder for Euclidean distances.

In 802.11a hardware, the Viterbi algorithm is most commonly used to accelerate decoding. This algorithm also performs optimal decoding, but does not require a full comparison of the incoming bit sequence with all possible valid sequences. Instead, it uses a dynamic programming technique, which requires only 2^{K-1} stored intermediate values, one for each level shown in the trellis or state in the finite automaton. For each state the current minimum "cost" according to the metric and the corresponding input sequence is saved. To process a coded n bit tuple, the new cost is calculated for each state from the two possible incoming edges: the zero and the one edge. For both edges the new cost is calculated from previous cost and the "cost" distance (usually Hamming or Euclidean) of that edge's n-bit sequence and the input n-bit sequence. Of both incoming paths only the one with smaller cost is saved; the other cannot possibly be the best path. Are both path costs equal a random decision is made. At the code bit sequence's end the best matching input sequence is saved at the all-zero state.

Due to the extra bits in the convoluted bit sequence, decoding the input improves bit error rates. In the following decoding error rate equations of a Viterbi or maximum likelihood estimator are described and reviewed (see [9, p. 257] or [28, p. 485]).

As a first step the probability of at least i wrong bits in a d bit sequences are calculated. The number of d bit sequences with i wrong bits is $\binom{d}{i}$. If the probability of a one-bit error is p, the probability of exactly i errors is

$$\binom{d}{i} p^i (1-p)^{d-i} \tag{6.9}$$

By summing over this equation one can calculate the probability that more than half of the d bit sequence is incorrect. For a d bit sequence of odd length more than $\frac{d+1}{2}$ bits must be incorrect. And for an even length at least $\frac{d}{2} + 1$ must be incorrect or $\frac{d}{2}$ with only half the probability, because of a random decoding decision. Thus the probability that more than half the sequence of d incorrectly detected is

$$P_d(p) = \begin{cases} \sum_{i=\frac{d+1}{2}}^{d} \binom{d}{i} p^i (1-p)^{d-i} & d \text{ odd} \\ \frac{1}{2}\binom{d}{\frac{d}{2}} p^{\frac{d}{2}} (1-p)^{\frac{d}{2}} + \sum_{i=\frac{d}{2}+1}^{d} \binom{d}{i} p^i (1-p)^{d-i} & d \text{ even} \end{cases} \tag{6.10}$$

where p is the probability of each bit correctly detected.

For a decoding error to occur during convolutional decoding, a wrong path through the trellis or state machine must be selected. This wrong path corresponds to an incorrectly detected bit sequence. However due to bit error correcting through redundancy, more than one bit error is required for the decoder to select the wrong path. For a *hard-decision* decoder using the Hamming metric, at least half the bits must be incorrectly detected to choose a different path. Thus selecting an incorrect decoding path for a d bit sequence corresponds to $P_d(p)$, where in the even bit case with equal probability of two paths, a random decision is made.

Selecting a wrong branch once, however, may still be corrected later during decoding because paths may be joined again. This error correcting property of the convolutional code can be upper bounded by summing over all paths longer than a specific length called d_{free}, which must be determined individually for each convolutional code. For each path length d all a_d paths of the same length must be considered. So the

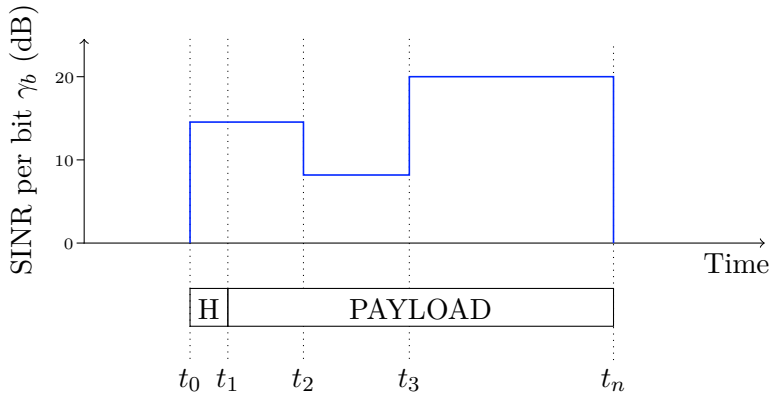


Figure 6.13: BER/PER segments with equal γ_b

decoding error probability $P_{\text{dec}}(p)$ for a convolutional code is

$$P_{\text{dec}}(p) \leq \sum_{d=d_{\text{free}}}^{\infty} a_d P_d(p) \tag{6.11}$$

where p is the probability of a single bit error. For p the equations 6.6 or 6.8 can inserted depending on underlying modulation scheme, and lead to

$$P_{b,\text{dec,mod}}(\gamma_b) \leq \sum_{d=d_{\text{free}}}^{\infty} a_d P_d(P_{b,\text{mod}}(\gamma_b)) \tag{6.12}$$

for bit error rate after convolutional decoding of a bit sequence transmitted using given modulation and received with γ_b SINR per bit.

6.6.3 Packet Error Rate

To determine packet error rate of a whole packet, the decoding error rates of each segment with equal SINR per bit γ_b must be calculated. See figure 6.13 for an example diagram of γ_b fluctuations during a packet's transmission duration. Because in 802.11a, header and payload may be encoded with different modulation schemes, a further distinction may be necessary.

For a segment containing l bytes, assuming uniform and independent bit errors distribution, the probability of error is

$$P_{\text{seg,mod}}(\gamma_b, l) \leq \left(1 - P_{b,\text{dec,mod}}(\gamma_b)\right)^{8 \cdot l} \tag{6.13}$$

and by combining all segments the complete PER is

$$P_{\text{per}} \leq 1 - \prod_{i=0}^{n} \left(1 - P_{\text{seg,mod}_i}(\gamma_{b,i}, l_i)\right) \tag{6.14}$$

where n is the number of segments, l_i is the number of bytes, mod_i the modulation schema and $\gamma_{b,i}$ the SINR per bit in segment i.

In ns-3, the PER is calculated according to equation 6.14 and compared for the ultimate reception decision to a random integer from $[0 \ldots 1]$. A simplified version of equation 6.12 is used: for BPSK and QPSK only the first length d_{free} and for M-QAM only the first two lengths d_{free} and $d_{\text{free}} + 1$ are added up.

Dewhitening of the deconvoluted bit sequence is currently not explicitly modeled in ns-3. This step has no impact on received packet error rate, because even though a single bit error has large effect on the dewhitening process, any bit error after dewhitening is detected by the CRC32 checksum (within reasonable probability). Thus whitening only amplifies actual discrete bit errors, which would be detected anyway. The transmission packet error rate is therefore not affected.

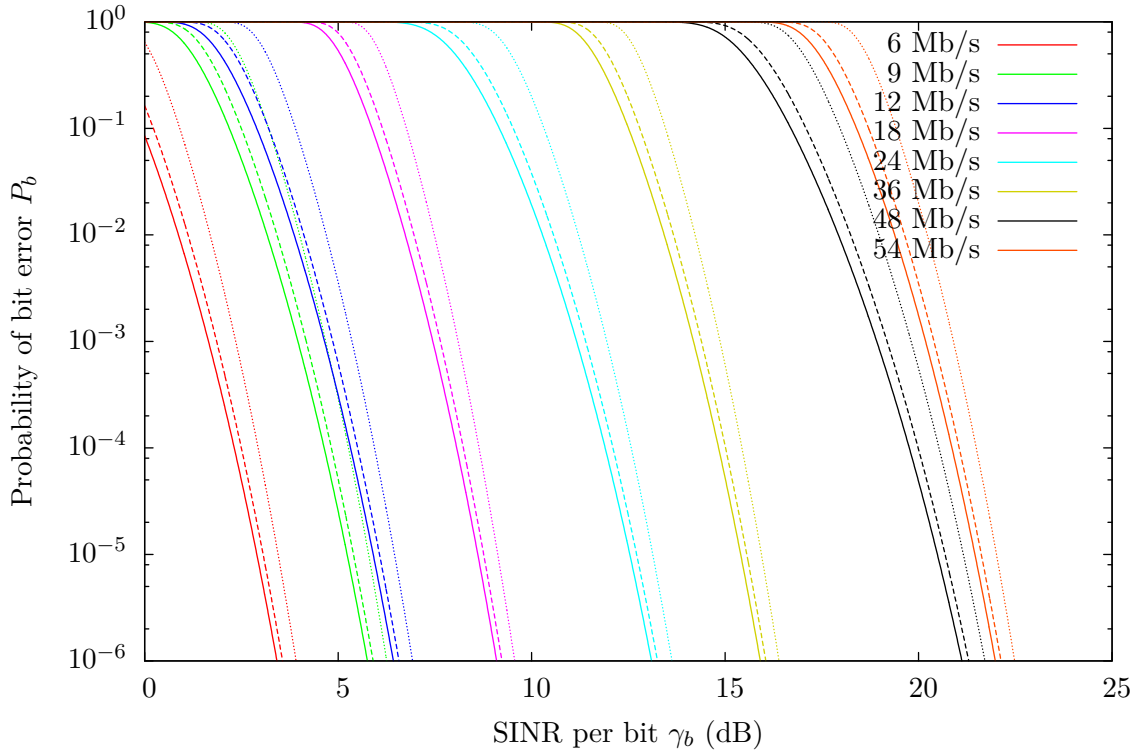


Figure 6.14: PER plots for 802.11a rates with 200, 400 and 2 304 bytes frame size

Values for d_{free} and a_d can be found, for example, in the tables calculated by Frenger et al. [6]. The values relevant for 802.11a are shown in table 6.2.

The effective packet error rate as used by ns-3 is plotted in figure 6.14 for all 802.11a transmission modes and three different frame sizes. The dotted line is for 200 byte, the dashed for 400 bytes and the solid line for 2 304 bytes packet size.

6.7 Verification

To verify the implementation of the SINR reception criterion and frame capture effect in `Ns2ExtWifiPhy`, two experiment scenarios were created. They are designed to specifically examine the two added features and were implemented in both ns-2.33 and enhanced ns-3.

6.7.1 Two Nodes Distance Scenario

To examine the SINR reception criterion, a very simple scenario containing two nodes is used. The node layout is illustrated in figure 6.15. One node sends packets at a constant rate to the other, which receives and counts them. The distance between the two nodes is varied. By determining the packet reception probability and comparing it to an equal experiment in ns-2, the reception criterion is validated. Neither cumulative noise nor the capture effect have influence in this scenario, because only one packet is sent at a specific time.

Coding rate	d_{free}	$a_{d_{\text{free}}}$	$a_{d_{\text{free}}+1}$	$a_{d_{\text{free}}+2}$	$a_{d_{\text{free}}+3}$
$R = 1/2$	10	11	0	38	0
$R = 3/4$	5	8	31	160	892
$R = 2/3$	6	1	16	48	158

Table 6.2: Property values of (punctured) convolutional codes of 802.11a (from [6])

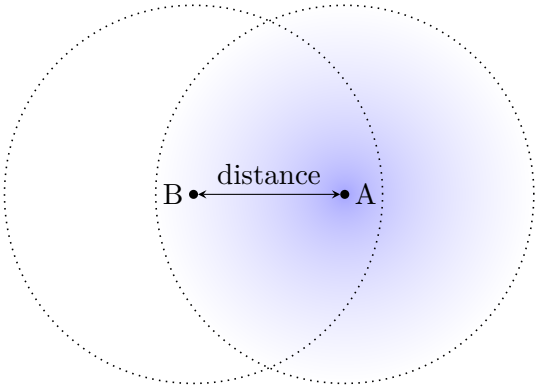


Figure 6.15: Two nodes scenario for reception criteria

The sender node is located at a distance ranging from 0 – 2 500 m with 10 m steps. Test broadcast packets containing 800 bytes are sent by one node every 5 ms with 20 dBm transmit power. This corresponds to a payload data rate of 160 KB/s. No ACK packets are transmitted back by the receiver. For the initial experiment run, both ns-2 and ns-3 were configured to use the 802.11a base transmission rate of 6 Mb/s. Each simulation run executes for 50 s simulated time and in this time 10 000 packets are sent in total.

Different propagation loss models were used in the experiments, and to get these to match in ns-2 and ns-3 turned out to be a great challenge. Even very small differences in propagation calculations lead to unequal reception probability curves, because SINR reception depends directly on calculated propagation losses. The noise floor was set to -99 dBm, as in all experiments in this thesis.

Figure 6.16 contains two result plots of the two nodes experiment, with each data point averaged over 100 independent replications. The 99% confidence interval is shown for each distance step.

Both plots show equal results for propagation loss models implemented in both of the two simulator. This shows that for the two nodes, reception range test scenario both PHY implementations behave equally.

The plotted results can also be verified against equations in section 5.2. The easiest to check is free-space propagation. Due to the constant noise floor of -99 dBm and the required 5 dB SINR for reception, an incoming packet needs to be received with -94 dBm power to be accepted. Compare the reception range for 20 dBm P_t and -94 dBm P_r in table 5.1 with the plot for Friis propagation. The reception drop is exactly between distance 2 320 m and 2 330 m.

Other 802.11a modes were also tested. Plots of these results (with free-space and Nakagami) are shown in figures 6.20 and 6.21. These are discussed in the context of a reception criteria comparison.

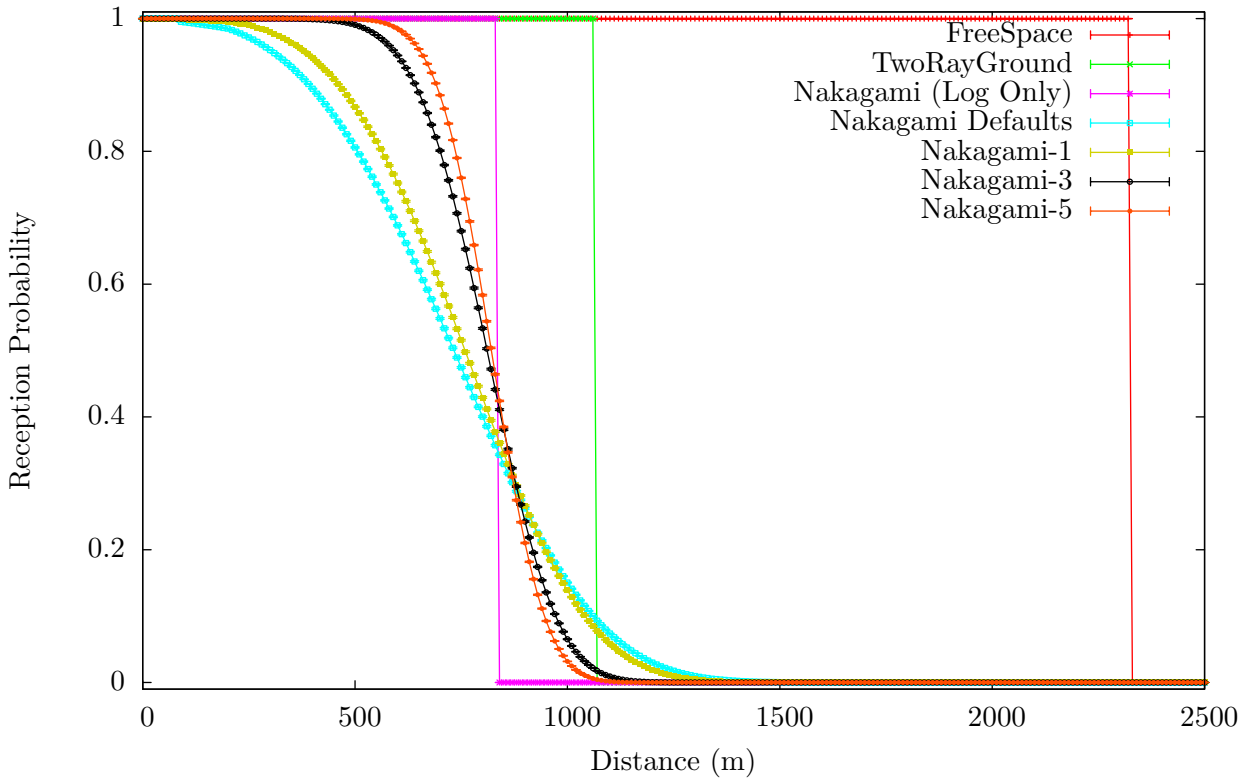
6.7.2 Three Nodes Capture Scenario

The second simulation set up is crafted to highlight the capture effect. For this purpose at least three wireless nodes are required: a listening “center” node C, one distant interferer B and one near, strong sender A. All three nodes are placed on a straight line as shown in figure 6.17(a).

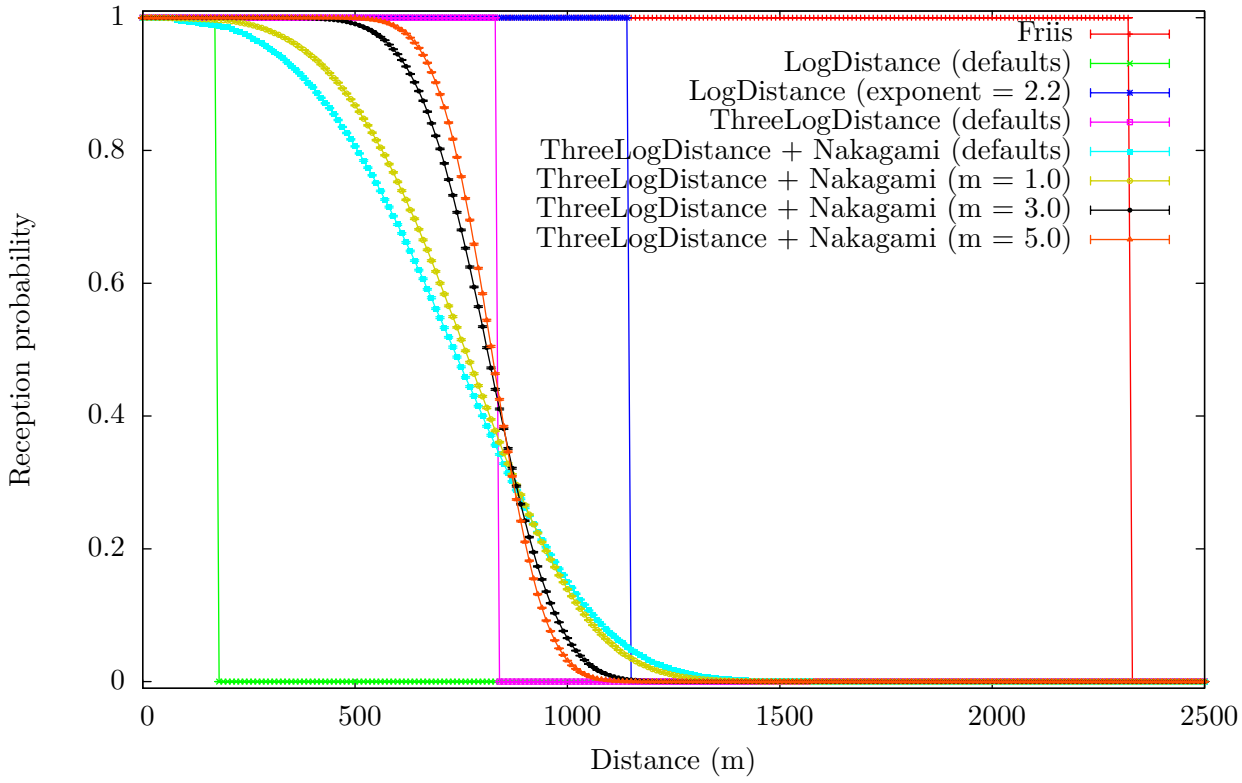
To provoke frame capture, the distant interferer B starts sending a frame, which is received at the listening destination node C, but not at the other sender A. Within the duration of this frame, the near sender A starts transmitting as well. This stronger signal will cause the currently received packet at C to be corrupt. If C supports frame capture, it will drop the weak incoming frame from B and correctly switch to the new stronger signal from A. This frame sequence is illustrated in figure 6.17(b). Depending on the time delta Δt, after which station A starts sending, “preamble” or “data” capture may be triggered at the receiver C.

In the experiment both sender A and interferer B send packets every 5 ms containing 200 payload bytes with 20 dBm transmit power. The receiving node C counts how many packets are received from A, frames from B are not counted. The carrier sense threshold of all nodes is set to -82 dBm and the noise floor is again -99 dBm.

For the first experiment free-space propagation loss is used. The nodes A and B must be far enough apart so that a packet from B is not received at A. As listed in table 5.1, the reception range in this constellation

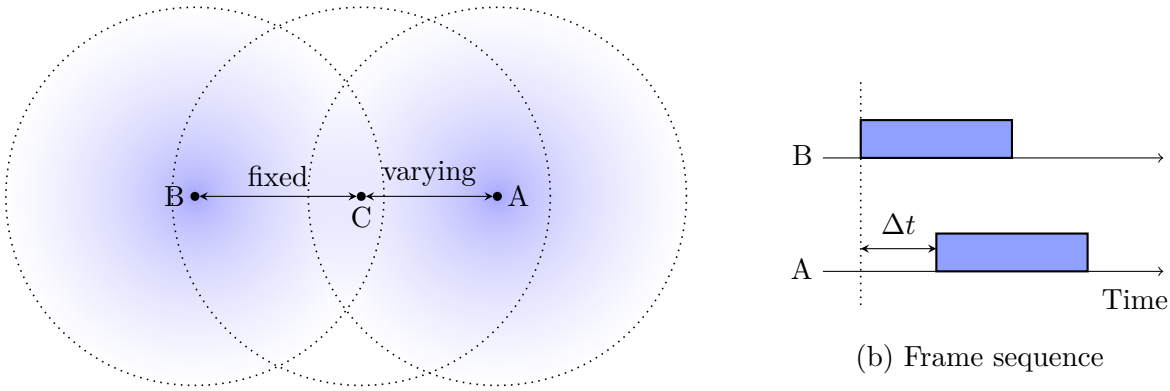


(a) In ns-2 with `WirelessPhyExt`



(b) In ns-3 with `Ns2ExtWifiPhy`

Figure 6.16: Two nodes experiment at 6 Mb/s with different propagation loss models



(a) Node layout

(b) Frame sequence

Figure 6.17: Three nodes scenario

is 2 323 m. So for the first experiment the nodes B and C are set 2 000 m apart. The distance between C and A is varied and thus the reception power of packets from A changes.

Second varying quantity in the simulation is Δt, the delay of the frame sent by A. Depending on the delay and distance, different aspects of the capture effect are triggered.

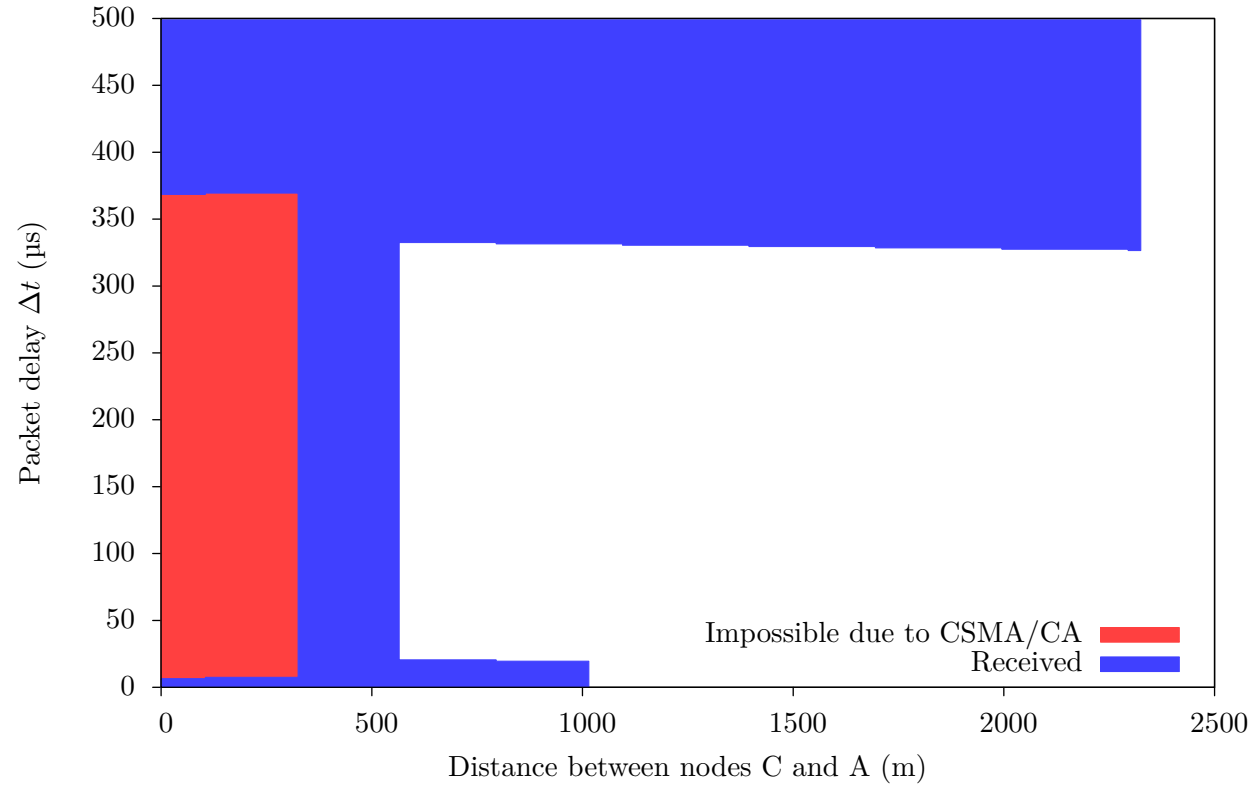
This scenario was implement in both ns-2.33 and ns-3.4 with the new `Ns2ExtWifiPhy`. For each delay and distance pair, the simulation is run only 0.1 s and so only 20 packets are processed. This is fully sufficient due to the deterministic set up of this scenario. No random effects can occur, because free-space propagation is fixed and long inter-packet time always allows the random DCF backoff to elapse.

The experimental results are plotted as area diagrams in figure 6.18. Note the x and y axis: x axis is the distance between C and A, while the y axis signifies the time interval Δt after which A starts to send after B. For each 10 m distance step and 1 µs time step an experiment instance was run. The experiment determines if A was allowed to send after Δt and whether packets from A could be received at B. If A was not allowed to send because of CSMA/CA rules, the data point in the plot is marked red. Any other color represents valid reception of packets from A. Furthermore, in ns-3, the capture effects triggered during packet reception were determined. Experiment configurations in which reception is always possible or when only possible with preamble capture or data capture are marked with three different colors is figure 6.18(b). In ns-2 the special coloring was not done, because the information necessary was not readily accessible for analysis, thus the blue area in figure 6.18(a) represents reception with or without capture.

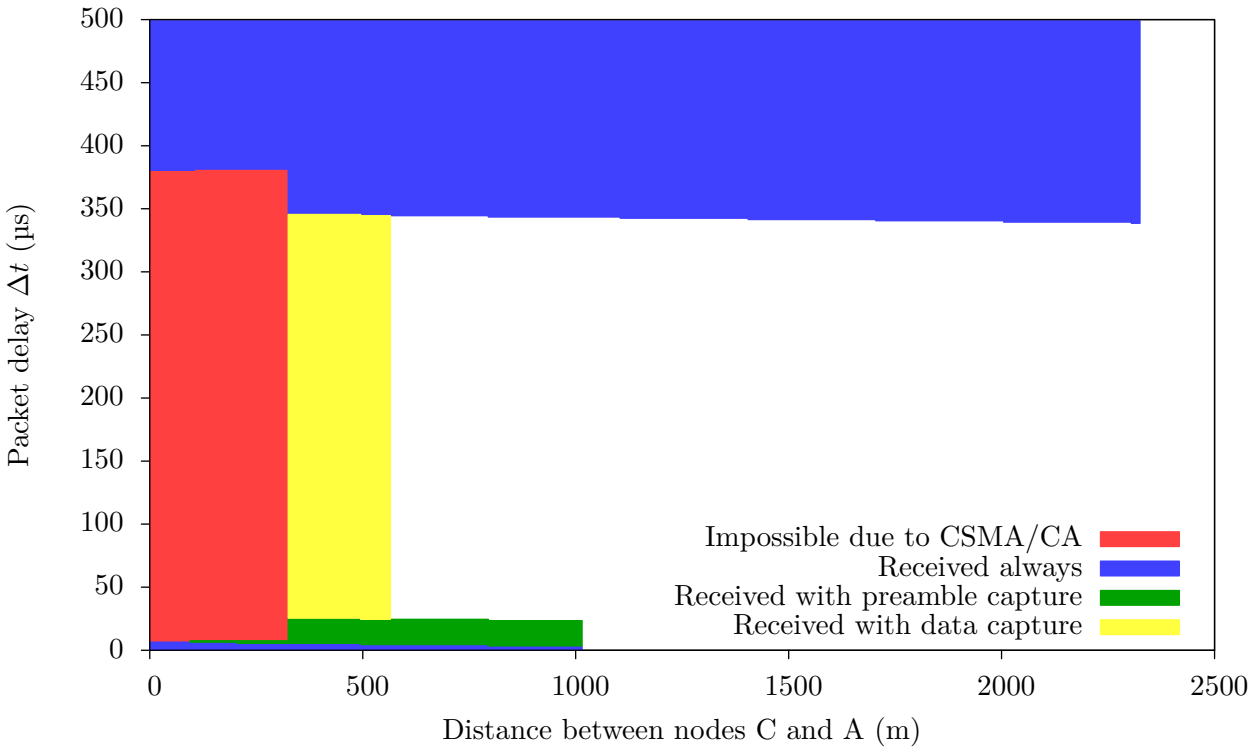
Many interesting effects can be seen in these constellation diagrams. First note the red area, in which A is not allowed to send after Δt due to CSMA/CA rules: this happens if A hears the interfering frame from B and starts receiving it. Once B and A are further apart than 2 323 m, reception power at A drops below the required 5 dB threshold and A cannot receive packets from B. Thus node A is allowed to send an overriding packet. This happens when C and A are further apart than 323 m, because B is a fixed 2 000 m away from C.

However two more effects are seen below the 323 m distance threshold: A can successfully send a packet for very small Δt and values larger than about 370 µs. Due to simulated signal propagation at the speed of light, packets are first heard at 2 000 m distance after 6.6 µs. This means that packets from B are first heard at C after this time interval. If A is close enough to C and sends within 6 µs, its packet will reach C before those transmitted by B, and thus will always be correctly accepted. This effect can be seen as the small unconditional reception area close to the x axis. It extends only to about 1 000 m due to the plot's 1 µs resolution.

The second effect with Δt larger than about 370 µs is due to packet duration. Duration of a packet with 200 payload bytes sent using the 6 Mb/s mode of 802.11a is 340 µs. This includes LLC, SNAP, MAC and PLCP headers, see section 7.4.1 on frame duration calculations. After further DIFS (= 34 µs) time, CSMA/CA rules allow a new frame to be sent. In the three nodes experiment, this means that if A's send wish is delayed for longer than 374 µs, then it will always be allowed to sent and received correctly at C. Note that no random backoff is required, because sending of the previous packet is always sufficiently long ago and no



(a) In ns-2 with `WirelessPhyExt`



(b) In ns-3 with `Ns2ExtWifiPhy`

Figure 6.18: Three nodes experiment with free-space propagation

collisions are detected.

Here a small difference between the ns-2 and ns-3 plots can be seen: packet duration in ns-2 is slightly shorter than in ns-3. In the ns-2 plot the red area extends up to 362 µs, because frame duration is calculated to be 328 µs. This difference is caused by ns-2 not adding an eight byte long LLC/SNAP header to the payload. The different frame length, however, are not relevant for this verification of the capture effect and only result in slightly shifted areas.

For distances larger than 323 m, the main capture effects can be seen. If capture is disabled, reception in the yellow and green areas is not allowed. However, even without capture, node A can correctly send to C, if packet transmission is delayed for longer than the frame duration from B. As discussed in the previous paragraphs, the packet duration is 340 µs or 328 µs, to which the propagation delay of at least 6.6 µs must be added. After this time interval, station C is idle again and can correctly receive packets from A. This is shown by the blue area in figure 6.18(b), which extends until 2 323 m, after which reception power drops below the SINR threshold.

For the range between 323 m and about 1 000 m, preamble capture is triggered at C, because the signal from A is superimposed on that from B. However, preamble capture is only activated if the interfering stronger signal is received within the PLCP header time, which is 20 µs in 802.11a. So the green area extends only up to 20 µs minus propagation delay.

Data capture is possible during the full duration of the weaker signal. However, only at a higher SINR threshold, which is shown by the smaller range of the yellow area. In this scenario, data capture is only activated within about 550 m.

If station A is further away than about 1 000 m or 500 m, then frame capture at C is not triggered due to the required threshold. Thus no reception is possible for the area right of the capture areas.

Comparison of these two plots shows that implementation of the capture effect in ns-3 is fully working and shows equal behavior as in ns-2.

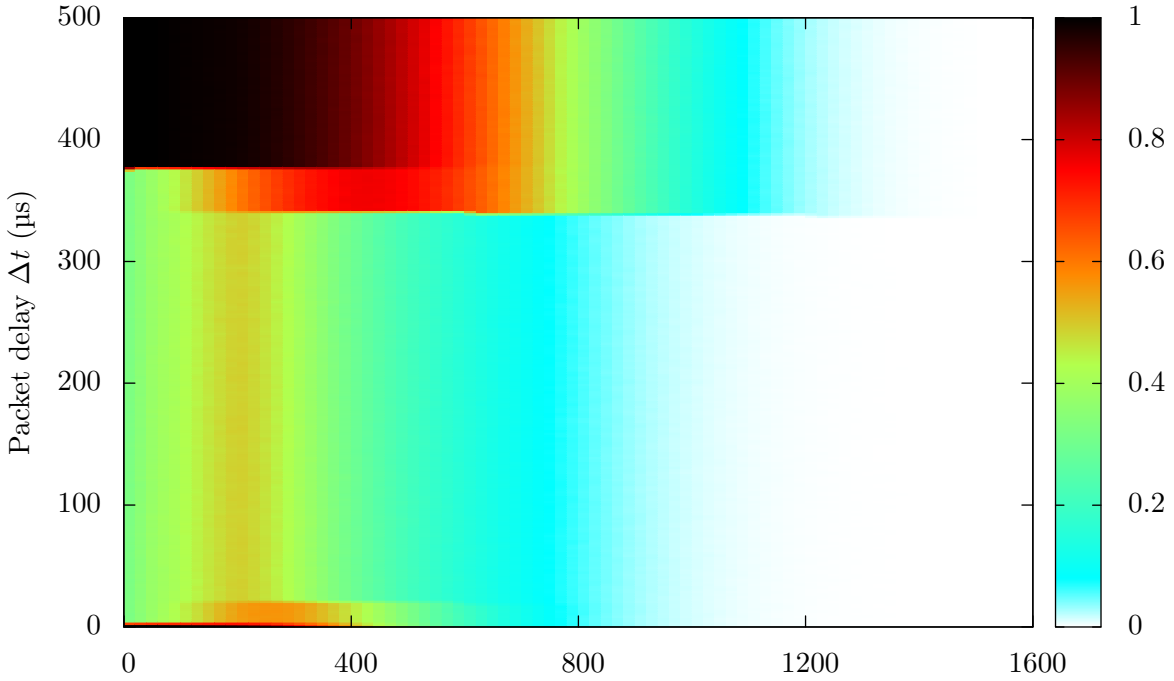
With Nakagami Propagation

The three nodes scenario was used for a third set of simulations. This time Nakagami and three-log-distance propagation loss models were configured. Due to the probabilistic nature of these models, reception of A at C becomes a statistical figure. Furthermore the discrete areas, in which the different effects are triggered, become blurred and no clear line can be drawn between them. This experiment was only run in ns-3 with the new capture implementation.

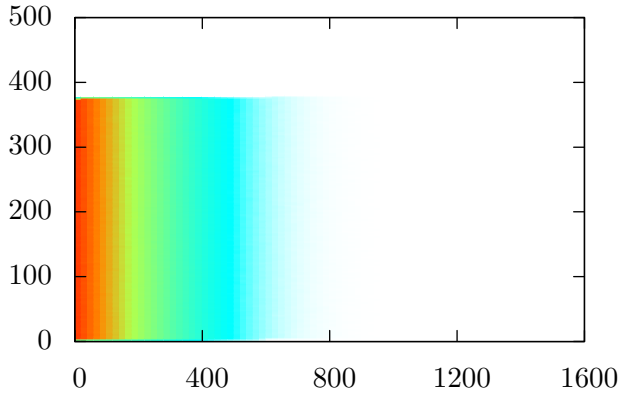
Scenario parameters were adapted to the lower reception range with log-distance and Nakagami propagation. Node B is placed at a fixed 600 m from C, while distance from A to C varies from 0 – 1 500 m. For each pair of distance and Δt the simulation was run 100 s in which 20 000 packets are generated by each sender node. The step resolution was decreased to 20 m distance steps and 2 µs time steps.

Simulation results are shown as a color-coded 2D plot in figure 6.19. For each distance and Δt pair, the probability of reception of packets from node A by station C is shown with the corresponding color. The four smaller plots show probability of the different effects separately. The top large plot shows only reception probability, and so is a sum of only the last three smaller plots; CSMA/CA prohibits reception and is not included in the sum.

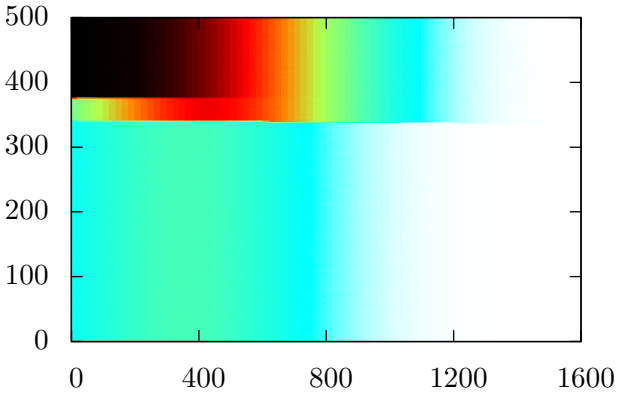
All effects described in the last section are also visible in these plots, though less clearly due to probabilistic reception. Note that statistical reception power only has great effect for the distance axis. Propagation delay has no probabilistic component and thus effects based on Δt intervals are more prominent.



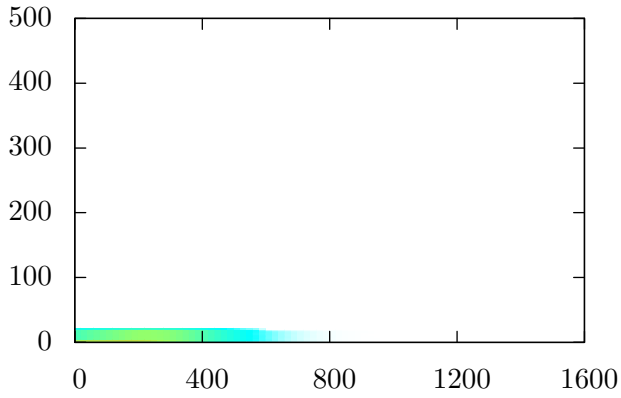
(a) Received



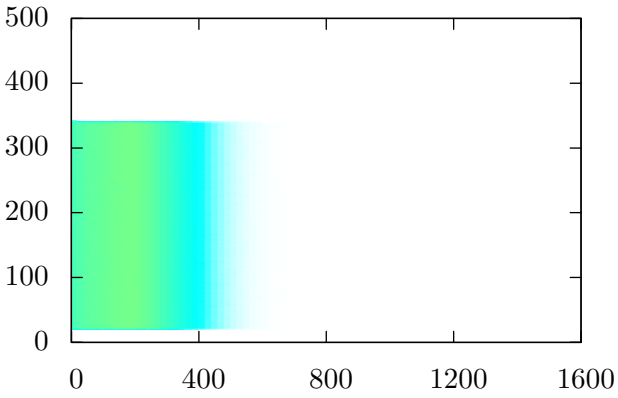
(b) Impossible due to CSMA/CA



(c) Received always



(d) Received with preamble capture



(e) Received with data capture

Figure 6.19: Three nodes scenario – Reception probability with Nakagami propagation

6.8 Discussion of Reception Criteria

With the newly added `Ns2ExtWifiPhy` and existing `YansWifiPhy` there are two 802.11 PHY models available in ns-3. They use two different reception criteria: SINR threshold, described in section 6.3, and BER/PER, described in section 6.6.

Both models are based on two different approaches to simulating the wireless transceiver layer. It is not possible to say that one is truly superior for all cases. Instead both have advantages and disadvantages that must be taken into account when selecting a model for experimentation.

The SINR threshold model gives a very simple, deterministic model of the transceiver. Wireless reception range with SINR is deterministic, if the propagation loss model is deterministic. This enables easy setting of a fixed, exact reception range, which proves useful for many experiments. The reception criterion is only based on channel effects as cumulative noise and reception loss, all specific transceiver properties are represented by the SINR threshold value itself.

On the other hand, BER/PER tries to grasp the reception decoding process analytically. It is composed of a highly detailed statistical analysis of theoretic digit bit keyings, signal modulation, convolutional decoding and more aspects of the transceiver. The analytical approach requires simplifications to the process, that may not be fully acceptable. For example, the analytical bit error rate assumes an independent, uniform distribution of bit errors, which does not reflect reality. However, how this simplification affects reliability of the reception criterion is not easily stated. Beyond that, it is clear that by focusing on the decoding process alone, the full complexity of a wireless receiver chipset is not covered. Representing other components in statistical equations is difficult if possible at all.

One of these is the implemented capture effect, as described in section 6.4. It is definitely an important modern chipset property, particularly for simulations with high numbers of broadcasting nodes in a wide-spread environment, which is modeled by probabilistic propagation models. In such setups, without capture, weak signals from distance stations can prevent reception of strong from near senders. This can have significant impact on experimental results and decisions based on these simulations.

Currently, the capture effect is implemented only in the `Ns2ExtWifiPhy` model, because its triggering policy is directly related to the SINR threshold criterion. The policy is not immediately compatible to the BER/PER reception criterion, because prior to switching to the new signal, reception of the old packet must be abandoned. However, in the BER/PER criterion this decision, whether a packet can be correctly received, is made at the end of the packet and takes the complete reception period into consideration. An instantaneous decision to abort a packet is not defined by the PER criterion. More work is needed on this subject.

In figures 6.20 and 6.21 the reception probability of all 802.11a modes is plotted for both BER/PER and SINR threshold models. The node scenario is the same two nodes constellation as discussed in section 6.7.1. In figure 6.20, deterministic free-space propagation was selected, and the SINR threshold model yields discrete reception ranges. On the contrary, BER/PER shows statistical behavior at the reception range even for a deterministic propagation loss model. Figure 6.21 shows the same simulation with three-log-distance and Nakagami fast fading, both with default parameters. The sharp drop in probability at 80 m is due to a new m value taking effect in the new distance field, whereas the second, continuous drop at 200 m is caused by a new log-distance exponent taking effect.

So which model should be picked? This depends on the requirements of the targeted experiment.

BER/PER is good if a solid mathematical foundation of the reception process is desired. However the BER/PER approach consists of a fixed set of equations. No method is currently included to adapt the criterion to special properties of a real wireless chipset.

SINR is suited for deterministic experiments. Furthermore, the SINR threshold is a single figure, that can easily be adapted to empirically measured values of a specific chipset.

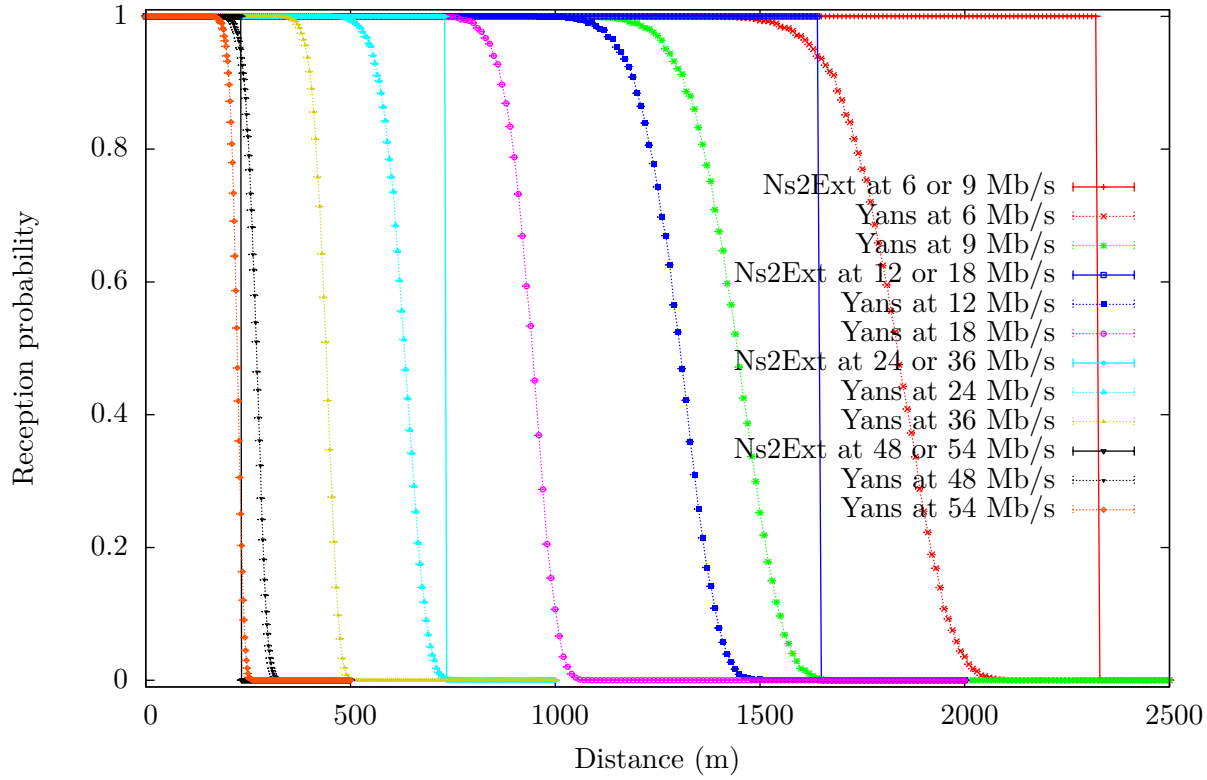


Figure 6.20: Two nodes reception probability with Friis propagation loss models and `Ns2ExtWifiPhy` or `YansWifiPhy` in ns-3

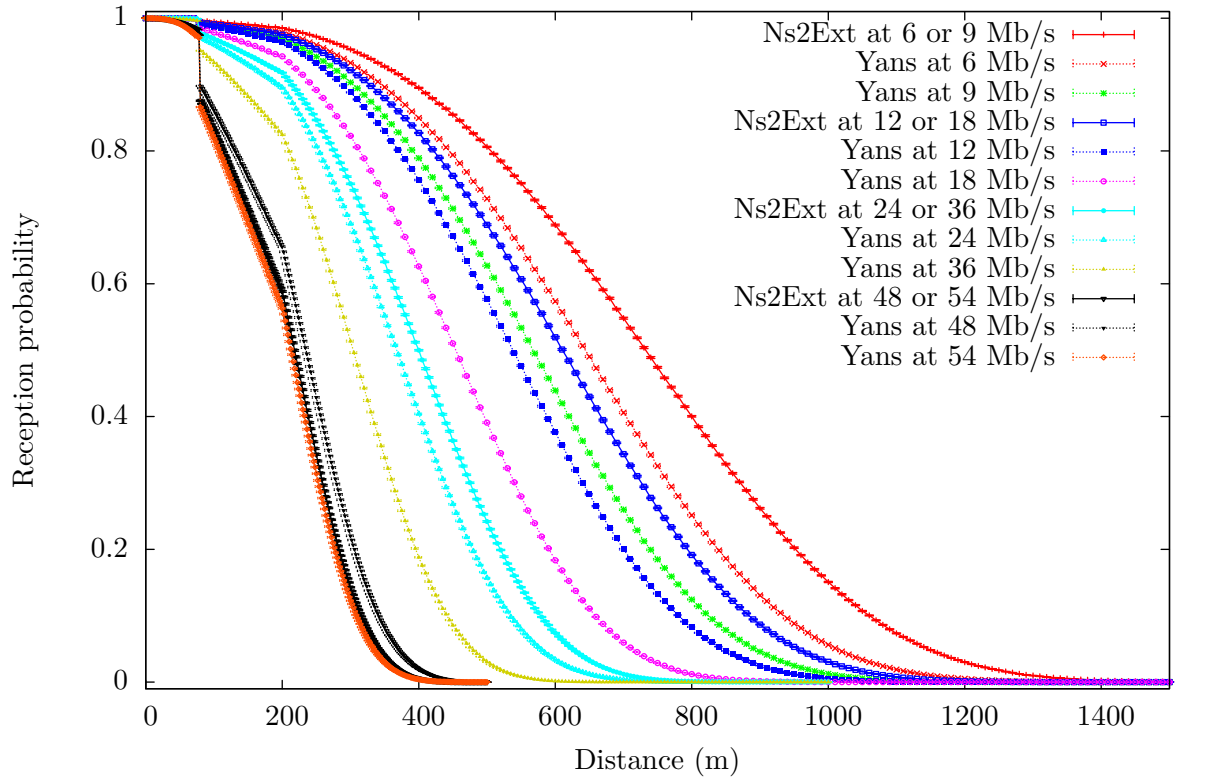


Figure 6.21: Two nodes reception probability with `ThreeLogDistance-` and `NakagamiPropagationLoss-Models`, and `Ns2ExtWifiPhy` or `YansWifiPhy` in ns-3

Chapter 7

EDCA QoS Extensions

The second objective of this thesis is to add EDCA extensions to ns-3. Main target of the EDCA implementation is to support relative QoS for communication in wireless ad-hoc networks. These encompasses MANET and VANET simulations, which require no complex HCCA coordination or AP management.

The EDCA implementation was built on the existing DCF implementation in ns-3 and supports both the 802.11e and draft 802.11p default parameter sets. Custom parameters can also be applied to the four EDCAFs.

Before reviewing the EDCA implementation, the existing DCF simulation design and interrelationship between the frame transmission coordination classes in ns-3 is discussed.

7.1 Modeling DCF

The DCF implementation of ns-3 was ported from yans (see section 4.2) by Mathieu Lacage, original implementor of the code in yans and core developer of ns-3. It is very well designed and already has provisioning for multiple, competing channel access functions. The UML diagram in figure 7.1 shows an overview of ns-3 classes related to DCF handling. In this section the processing of DCF backoff and medium access granting is described. Because the existing DCF code is already structured for multiple coordination functions, the corresponding components of EDCA are mentioned in this section. Original DCF is implemented with one EDCAF.

Processing of DCF access is based on "signals" from the PHY and lower MAC layer. From the PHY, physical carrier sense events as packet reception, transmission and CCA_BUSY indications must be signaled for correct DCF management. Likewise, the lower MAC layer must pass NAV durations from each received packet to the DCF manager.

These carrier sense events are sent to the main DCF coordinator class `DcfManager` via two signaling classes, which follow the *observer* design pattern. The abstract classes are named `WifiPhyListener` and `MacLowListener`, of which objects can be registered with `WifiPhy` and `MacLow` instances to receive events. From each abstract classes a specific observer class `PhyListener` and `LowListener` is derived to forward events to `DcfManager`. No processing is done in these signal forwarders.

To coordinate DCF, the `DcfManager` receives these signals and saves time and duration for NAV, frame reception, transmission and channel busy events. From these `Time` values, the function `GetAccessGrantStart()` calculates the earliest future time at which a frame may be sent or backoff is performed. This is the latest end time of all carrier sensing mechanisms (which is the earliest medium idle time), plus one SIFS or EIFS as appropriate.

Permission to send a packet is granted by the `DcfManager` to a `DcfState`, of which multiple can be registered. Each `DcfState` represents a DCF or EDCAF, and contains the necessary parameters AIFSN, backoff counter, CWmin, CWmax, current CW and TXOPLimit. A registered `DcfState` can request access to the medium by calling `RequestAccess()` at `DcfManager`. When channel access is granted to a `DcfState`, the function `DoNotifyAccessGranted()` is called and frame transmission is initiated. On collisions events, the functions `DoNotifyCollision()` or `DoNotifyInternalCollision()` are invoked to restart backoff with an increased

contention window. These two classes follow a modified, "interactive" observer pattern, because DcfStates need to request access.

In ns-3 there is only one "user" of the channel access manager: DcaTxop. It represents a complete channel access category (AC) including packet queue, coordination function and retransmission handling. It is plugged into DcfManager with an associated subclass of DcfState named DcaTxop::Dcf, which forwards channel access events to the main class. Compare this separation of concerns with figure 2.14: the DcfState corresponds to the box containing AIFS[AC] and CW[AC] values, while the whole column of queue and coordination function is represented by DcaTxop.

Packets from higher layers that are send via the wireless device are first processed at the WifiMac (MacHigh layer). This level will be discussed in more detail in section 7.2. Ad-hoc networks operating in DCF can be simulated with the AdhocWifiMac implementation, which simply queues all packets in one DcaTxop.

7.1.1 Simulating Channel Access Rules

To illustrated the functioning of DCF in ns-3, processing of one queued packet is traced. The packet arrives from a higher layer at DcaTxop via the Queue() function. In the original ns-3.4 code, the packet is not processed but only stored in the associated WifiMacQueue. To eventually gain access to the channel, DcfManager::RequestAccess() is called by DcaTxop for each newly arrived packet.

The principle idea of DcfManager is to calculate backoff countdown in the discrete event simulator with as few scheduled events as needed. Instead of scheduling a simple decrementation event handler at each slot time, the future backoff finishing time is predicted and appropriate measures are taken if the prediction is invalidated.

In RequestAccess() the coordinator first checks if immediate access to the channel is allowed. The 802.11 standard allows immediate access if the last backoff fully elapsed. For this check, the function GetBackoffEndFor() calculates the end of the previous backoff of the EDCAF from the last backoff start, AIFS and the current backoff counter value. The last backoff start time directly depends on carrier sense times m_last.... If the previous backoff end time lies in the past and the medium is idle, immediate access is granted. If the medium is busy, a collision is signaled and the backoff procedure is started. Otherwise, an unfinished backoff process continues.

The main function of DcfManager is DoGrantAccess(), which checks the backoff intervals for all registered DcfStates. Again the end time point for each DCF object is calculated by GetBackoffEndFor(). If access is requested and the backoff timeout lies in the past, medium access is granted to the coordination function by calling DoNotifyAccessGranted(). If multiple coordination functions would simultaneously be granted access, internal collision resolution is triggered. Only the DcfState with highest priority (registered first) is granted access, all others are notified via DoNotifyInternalCollision() to restart backoff.

If no registered DcfState is granted access, an event is prepared to reprocess channel access later. This event is scheduled at the earliest predicted next channel grant time, if no interfering carrier busy signals are raised. At this predicted time, the function AccessTimeout() is called to reprocess DoGrantAccess() and grant appropriate channel access.

However, a channel busy event can interrupt backoff countdown and invalidate the prediction. This is done in a busy sense event handler by advancing one of the m_last... Time values. Because GetBackoffEndFor() includes GetBackoffStartFor(), which in turn depends on GetAccessGrantStart(), advancing the carrier sense Times automatically advances the backoff end times.

To correctly simulate backoff countdown, the backoff variable must be updated on each carrier busy event. Only the number of complete SlotTimes elapsed may be deducted from the backoff counter in DcfState, partially elapsed intervals must be ignored. By calling UpdateBackoffSlotsNow() the new backoff value is set and the backoff start time updated.

The AccessTimeout() event is not rescheduled on a carrier busy event. It checks for access and reschedules at the next predicted access time automatically.

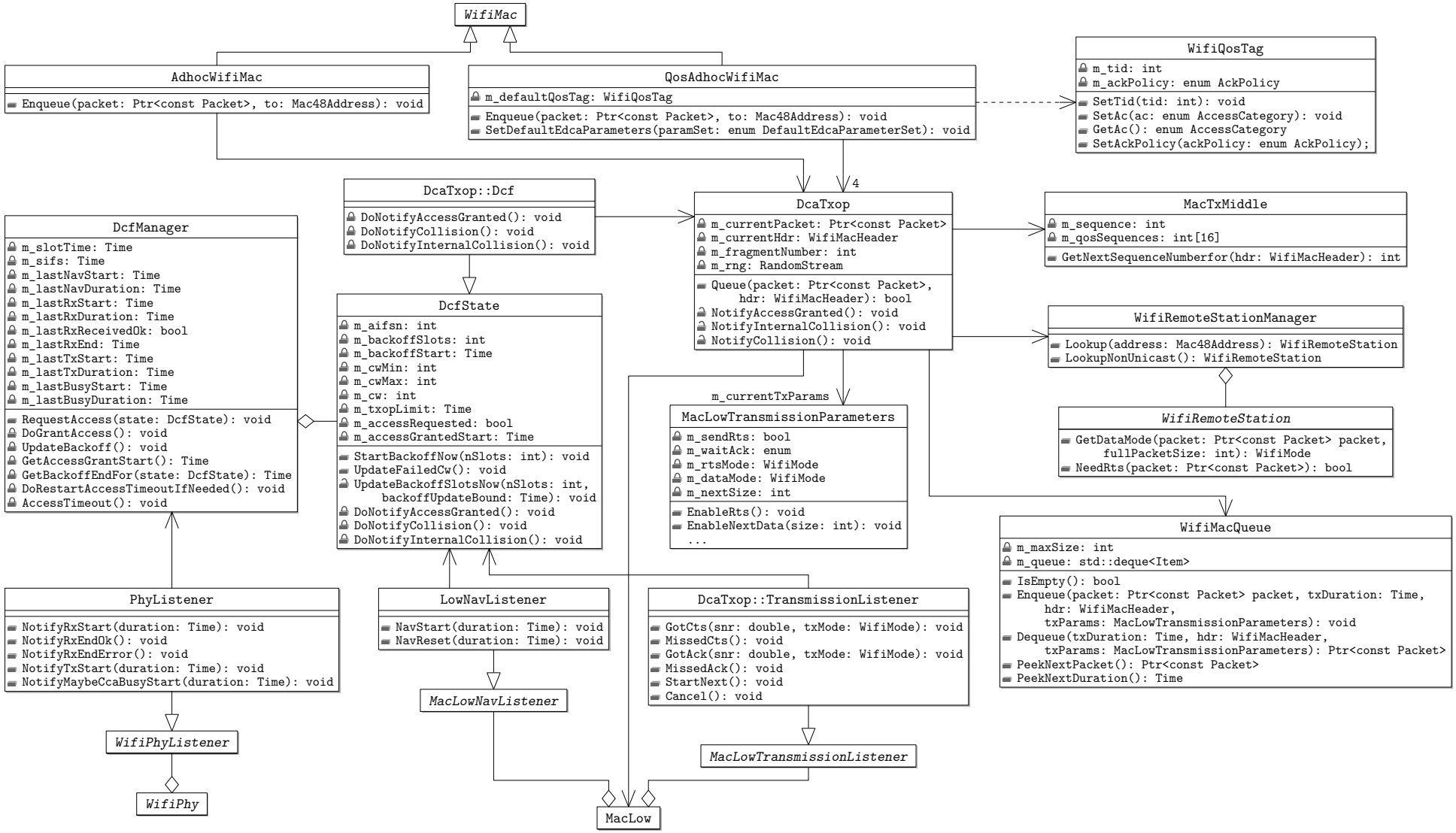


Figure 7.1: UML diagram of EDCA related classes

7.1.2 Initiating Frame Transmission

Once `DoNotifyAccessGranted()` is signaled to `DcaTxop` by the coordination manager, a single frame may be transmitted in DCF. Lower packet transmission aspects are handled by `MacLow`, which ultimately forwards the packet to `WifiPhy`.

When receiving channel access permission, `DcaTxop` dequeues the next waiting frame. For this frame, transmission parameters are requested from `StationManager` depending on frame type, size and destination. The transmission parameters can include an optional RTS/CTS exchange and fragmentation for which `StationManager` can apply a frame length threshold. In the 802.11 base standard, ACK packets are mandatory for all unicast frames, while for broadcast or groupcast frames no ACKs are sent. All these parameters are specified to `MacLow` in the `MacLowTransmissionParameters`.

`MacLow` handles transmission of a single frame, with an optional preceding RTS/CTS exchange, and waits for a following ACK frame if indicated. No retransmission is handled by `MacLow`. Successful or failed transmissions are signaled to `DcaTxop` by the abstract `MacLowTransmissionListener`, which is forwarded using `DcaTxop::TransmissionListener`. Within `DcaTxop`, retransmission events for CTS or ACK timeouts are handled if appropriate.

Fragmentation is handled in `DcaTxop` by sending only partial packets to `MacLow`. 802.11 allows fragmentation "bursts", in which a sequence of frames is transmitted with only SIFS waiting intervals. For each fragment the NAV duration is set to include the next prospective packet as well. This is signaled to `MacLow` via the `EnableNextPacket()` function and triggers a callback via `MacLowTransmissionListener::StartNext()` after SIFS of the preceding transmission. This mechanisms was changed to allow TXOP bursts, as described in the following section.

7.2 Extending Model with EDCA

In this section the modifications added to ns-3 to support EDCA are described.

`DcfManager` and `DcaTxop` already contain much of the functionality required. Multiple queues and EDCAF can be registered with `DcfManager` and channel access fully follows the 802.11e standard. Instead of DIFS the code in `DcfState` already works with AIFS and AIFSN, CWmin and CWmax can be set as required.

To provide relative QoS to upper layers, a `QosAdhocWifiMac` high MAC layer was implemented, which contains four EDCAF by creating four associated `DcaTxop`s. Two different default EDCA parameter sets can be selected by setting the attribute `EDCAParameterSet` upon creation: 802.11e and 802.11p/D4.02. The values are listed in section B.2 of the appendix.

Upper layers can select the AC or traffic identifier (TID) by applying a `WifiQosTag` to a packet. If a packet is `Enqueue()`-d without an attached `WifiQosTag`, the function uses the `m_defaultQosTag`, which is initially set to AC_BE.

The QoS field added to the MAC header for QoS-data frames allows specification of an ACK policy. Thus unicast frames can also be sent without subsequent explicit acknowledgment. The relevant `MacLow` code of ns-3 was adapted and the ACK policy may also be specified in the `WifiQosTag`.

The switching between the different `DcaTxop` access category (AC) is performed in `QosAdhocWifiMac::Enqueue()`. All other relative QoS features are implemented by `DcfManager`.

7.2.1 Implementing TXOPLimits

One crucial feature of EDCA was not present in the code of ns-3.4: TXOP limits. These prescribe a maximum frame transmission duration for each EDCAF. Furthermore, when granted access to the channel the EDCAF may send a "burst" sequences of smaller frames spaced only with SIFS.

For this thesis `DcaTxop` was extended to adhere to a TXOPLimit. Creation of a new class, e.g. called `EdcaTxop`, was not done, because the old behavior of `DcaTxop` is retained for TXOPLimit = 0.

To correctly implement TXOP limits and burst sequences, the duration of each frame enqueued in `DcaTxop` must be calculated and remain fixed. This requires that all `MacLowTransmissionParameters` are decided

upon at queue time and may not be changed later on. The old behavior of ns-3 was to allow `StationManager` to determine RTS/CTS, fragmentation and wifi mode parameters at transmission time. To retain the old behavior for DCF, only optional wifi mode overrides are declared via `MacLowTransmissionParameters`.

Once all transmission parameters are fixed in `DcaTxop::Queue()`, the transmission duration of the frame can be determined. If the frame duration exceeds TXOPLimit, queuing is rejected.

The frame duration is saved together with transmission parameters and packet data in the `WifiMacQueue`.

TXOP bursts are implemented by modifying the fragmentation mechanism. When a TXOP is granted with TXOPLimit > 0, the prospective frame transmission duration must be checked. Simultaneously the duration of the following frame is checked: if transmission of both frames with a SIFS interval is possible within the TXOP, the NAV duration of the first is set of encompass both frames. This is done by setting `EnableNextPacket()`. Because of this flag, `MacLow` calls `StartNext()` after SIFS, upon which the following frame of the burst sequence is transmitted. The same NAV extension decision is made again, and so the burst sequence can extend to fill the whole TXOP.

The implementation of EDCA including TXOP limits and bursts is verified in section 7.4.

7.3 Implementation Issues

Before discussing the scenarios to check EDCA, some problems encountered during implementation and verification are discussed. Multiple errors in the DCF code in ns-3 were found and corrections for these were promptly included in the ns-3.4 release.

During the maximum throughput experiment, described in the following section 7.4.1, the following three bugs were found. Their level of impact depends greatly on the scenario; in the verification tests they showed a major change of results.

- The formula for contention window (CW) growth was incorrectly implemented. In ns-3.3 an incorrect formula, $\mathrm{CW}_{\mathrm{new}} := \min\{2 \cdot \mathrm{CW}_{\mathrm{old}},\ \mathrm{CWmax}\}$, was used instead of the correct equation 2.2.
- For backoff processing a uniformly distributed random value is chosen from $[0 \ldots \mathrm{CW}]$. However, the value generated by the underlying `UniformVariable` was not correctly processed and not all possible integers were equally probable. This resulted in a skew of random inter-packet waiting times.
- The old `DcfManager` code issued an invalid collision signal for two broadcast packets immediately following each other with a backoff of zero. This was due to incorrect handling of immediately granted medium access in `DcfManager::RequestAccess()`.

Furthermore, ns-3.4 only contains the `OnOffApplication` for traffic generation experiments. This class is designed to generate constant bit rate (CBR) in *on* state, where *on* and *off* states alternate with randomly distributed periods. This application was not sufficiently flexible to generate the traffic streams envisioned for the experiments of this thesis. Moreover, `OnOffApplication` first generates packets at the *end* of each calculated interval.

So a new, highly flexible traffic generator called `TrafficApplication` was added to ns-3, which was based on `OnOffApplication`. It sends packets generated by an *abstract factory*, `TrafficPacketFactory`, at intervals defined solely by a `RandomVariable`. In the packet factory packets can be create with randomly distributed sizes and necessary QoS tags attached. This enables simulation of both periodic broadcast beacons sent at jittered intervals and Poisson-distributed packets streams. The periodic broadcast is used in the speed test scenario to correctly compare ns-2 and ns-3, while Poission-distributed waiting times are employed in the second of the EDCA experiments.

7.4 Verification

Two simulation experiments are used to check the EDCA implementation in ns-3. Both focus on two different aspects of modeling coordinated medium access: the first verifies correct handling of different coordination

sequences from only one node. By saturating the medium, correct inter-packet backoff management is tested. The second scenario shows how EDCA parameters effect four equal, simultaneous traffic streams. Relative QoS as provided by EDCA is thereby verified.

7.4.1 Maximum Throughput

The first experiment used to verify EDCA is simulation of the maximum throughput achievable with different parameter configurations. Initially this experiment does not sound very interesting; maximum throughput of 802.11 is a well studied topic. However, the maximum throughput depends directly on the EDCA parameters AIFSN and CWmin. Furthermore, the full, complex DCF/EDCAF and CSMA/CA scheme including random backoff and DATA/ACK or TXOP burst sequences is verified with this experiment.

Most importantly, the maximum throughput values for each configuration can be determined analytically from information in the 802.11 standard. Thus experimentally determined values can be compared to independent, exact calculations.

The node layout is kept very simple: two nodes are created at exactly the same location. One node sends a very high number of packets, saturating the channel as far as EDCA rules permit. The other node counts the number of received packets and from simulated time and packet payload bytes, the maximum *payload* throughput is determined. Because both nodes are at the same place, propagation loss and delay have no influence.

Different parameters of the saturating packet stream are varied. Wireless transmission data rate is clear to have impact. But also packet size has great impact due to the increased number of waiting intervals. These waiting intervals depend on the AIFS and CWmin parameters of the EDCAF.

Moreover, most interesting for EDCA validation are the different transmission mechanisms for a packet. Packets can be ACKed or sent as unacknowledged broadcast frames. Within a TXOP multiple frames can be sent as a burst sequence, each with or without ACK following. All four combinations were tested. RTS/CTS exchanges were not included in experiment, because they are not implemented in the scope of the modified code.

The experimentally determined maximum throughput values were compared to analytical calculations. For each frame sequence, unacknowledged broadcast, ACKed unicast, ACKed and unACKed TXOP bursts, the maximum throughput can be calculated using elementary arithmetic, solely from information in the 802.11 standard. It thus is independent from possible bugs in ns-3.

This was done for the four frame sequences and all ACs defined by the default EDCA parameters of 802.11e and 802.11p/D4.02. The analytic values were then compared to experimental results. For the experimental runs and analytic calculations, the packet sizes 80, 200, 400 and 2 304 were chosen to represent small and large payloads and the three 802.11a wireless modes 6 Mb/s, 24 Mb/s and 54 Mb/s were selected for low, medium and high data rates.

Analytic Maximum Throughput

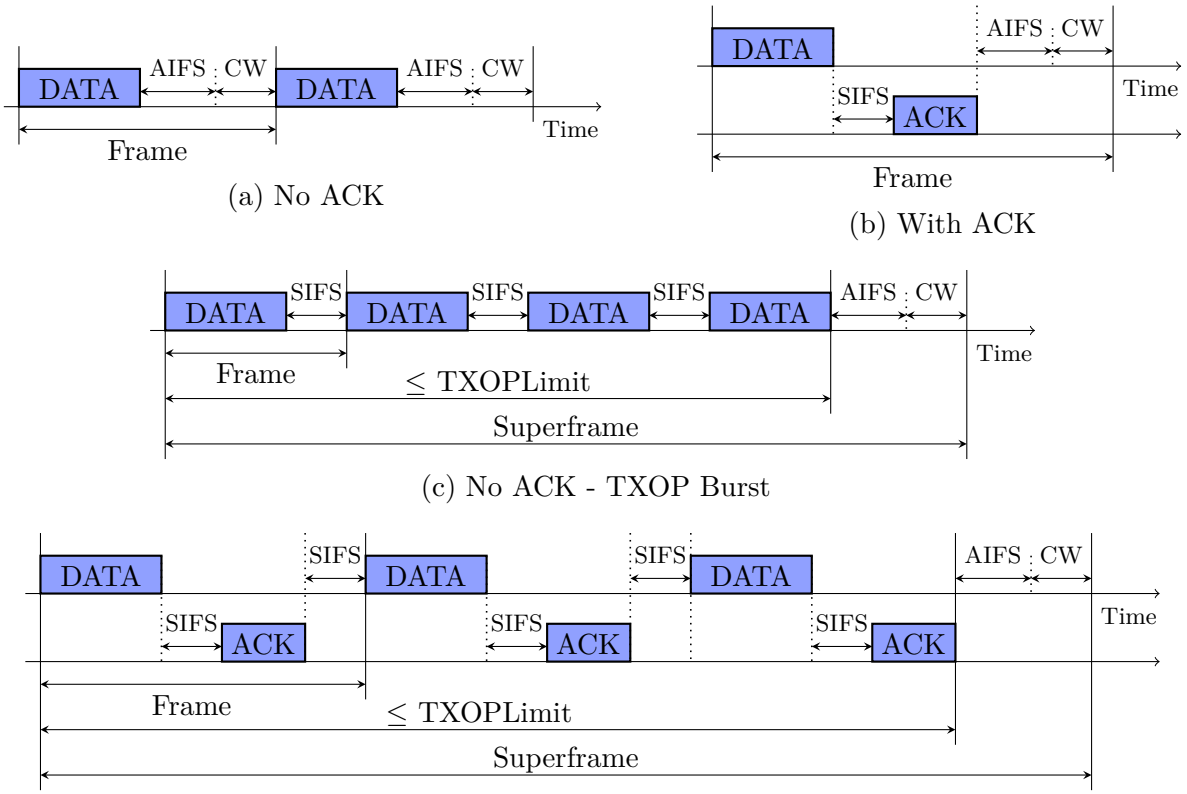
Four different frame transmission options are tested: unACKed broadcast frames, ACKed unicast frames, a burst of unACKed frames in a TXOP, and a burst of ACKed frames. All four frame sequences are illustrated in figure 7.2.

Each option yields a different maximum throughput, that also depends on the average waiting time after a frame. The waiting time consists of AIFS plus the random backoff selected from $[0 \dots \mathrm{CW}] \cdot \mathrm{SlotTime}$.

Due to lack of collisions in the simple two nodes experiment, the contention window never changes from CWmin. Furthermore, because the maximum throughput of an infinitely large number of packets is determined, the expected mean value of all random backoffs can be used. As random backoff is uniformly distributed between 0 and CWmin, the expected value is simply $\frac{\mathrm{CWmin}}{2}$.

The frame sequence of unACKed broadcast frames is shown in figure 7.2(a). To calculate maximum throughput, the length of one data frame plus the average required inter-frame time must be analyzed.

The first necessary step is to calculate the duration of a data frame containing x payload bytes. These payload bytes are wrapped with header and trailer of multiple intermediate layers before transmission.



(a) No ACK

(b) With ACK

(c) No ACK - TXOP Burst

(d) With ACK - TXOP Burst

Figure 7.2: Maximum throughput frame sequences

Every packet sent on an 802.11 device is wrapped with a three bytes LLC header followed by a five bytes long subnetwork access protocol (SNAP) extension header, both defined in 802.2. In ns-3, this header is added by `WifiNetDevice`; ns-2 lacks simulation of the LLC/SNAP header.

The 802.11 MAC adds a header and trailer to the frame payload for addressing and distributed coordination. Figure 2.7 shows all possible header fields, however, not all are prefixed to each frame. Depending on the frame type field, a different set of fields is transmitted. For maximum throughput calculation, this requires a distinction between non-QoS data frames and QoS data frames, because QoS frames contain the extra 2 bytes QoS field.

So the total number of bytes b_{data} or $b_{\text{QoS-data}}$ contained in a MAC protocol data unit (MPDU), before sending to the PHY, is

$$b_{\text{data}}(b_{\text{payload}}) = \underbrace{24\,\text{byte}}_{\text{MAC header}} + \underbrace{8\,\text{byte}}_{\text{LLC/SNAP header}} + b_{\text{payload}} + \underbrace{4\,\text{byte}}_{\text{MAC trailer}}$$

$$b_{\text{QoS-data}}(b_{\text{payload}}) = \underbrace{26\,\text{byte}}_{\text{MAC QoS header}} + \underbrace{8\,\text{byte}}_{\text{LLC/SNAP header}} + b_{\text{payload}} + \underbrace{4\,\text{byte}}_{\text{MAC trailer}}$$

where b_{payload} is the number of payload bytes. The MAC header of a normal, non-QoS frame contains the following fields: frame control, duration, address 1 (destination or broadcast address), address 2 (source address), address 3 (BSSID) and sequence number. For QoS frames the addition QoS control field is added. The MAC trailer contains only the four bytes CRC32 checksum.

For ACKed sequences the duration of an ACK frame is required. This control frame's structure is different from data frames. It consists only of the following fields: frame control, duration, address 1 (destination) and CRC32 checksum [12, figure 7-8]. The total number of bytes is thus

$$b_{\text{ACK}} = \underbrace{2\,\text{byte}}_{\text{Frame Control}} + \underbrace{2\,\text{byte}}_{\text{Duration}} + \underbrace{6\,\text{byte}}_{\text{Destination Address}} + \underbrace{4\,\text{byte}}_{\text{CRC32}}$$

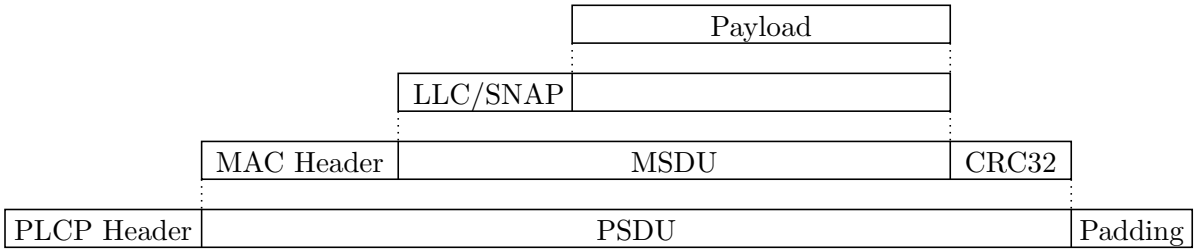


Figure 7.3: Wrapping of payload with headers and trailers

For the maximum throughput experiment three different 802.11a transmission modes are used: 6 Mb/s, 24 Mb/s and the highest 54 Mb/s. On-air PHY transmission duration of a frame t_f is specified by the 802.11 standard [12, p. 625] as

$$t_\mathrm{f}(b, N_\mathrm{NBPS}) = \underbrace{16\,\mu\mathrm{s}}_{\text{PLCP Preamble}} + \underbrace{4\,\mu\mathrm{s}}_{\text{“Signal” symbol}} + \underbrace{\left\lceil \frac{16\,\mathrm{bit} + b \cdot 8\,\frac{\mathrm{bit}}{\mathrm{byte}} + 6\,\mathrm{bit}}{N_\mathrm{NBPS}} \right\rceil \cdot 4\,\frac{\mu\mathrm{s}}{\mathrm{symbol}}}_{\text{Data symbols}}$$

where b is the number of MSDU bytes and N_NBPS the number of bits per symbol, depending on the used data rate (see table 2.3). The values are 24 $\frac{\mathrm{bit}}{\mathrm{symbol}}$ for 6 Mb/s, 96 $\frac{\mathrm{bit}}{\mathrm{symbol}}$ for 24 Mb/s and 216 $\frac{\mathrm{bit}}{\mathrm{symbol}}$ for 54 Mb/s. The PLCP wrapping of the MSDU bytes is illustrated in figure 2.2. Duration of all frames used for the maximum throughput experiment are tabulated in table 7.1.

From frame duration and waiting time until the next transmission, the length of a transmission iteration period p can be easily calculated for unACKed and ACKed sequences.

$$p_\mathrm{NoACK}(b, N_\mathrm{NBPS}, \mathrm{AC}) = t_\mathrm{f}(b, N_\mathrm{NBPS}) + \mathrm{AIFS[AC]} + \tfrac{\mathrm{CWmin}}{2} \cdot \underbrace{9\,\mu\mathrm{s}}_{\text{SlotTime}}$$

$$p_\mathrm{ACK}(b, N_\mathrm{NBPS}, \mathrm{AC}) = t_\mathrm{f}(b, N_\mathrm{NBPS}) + \underbrace{16\,\mu\mathrm{s}}_{\text{SIFSTime}} + t_\mathrm{f}(b_\mathrm{ACK}, N_\mathrm{NBPS}) + \mathrm{AIFS[AC]} + \tfrac{\mathrm{CWmin}}{2} \cdot \underbrace{9\,\mu\mathrm{s}}_{\text{SlotTime}}$$

for b the MSDU payload bytes, N_NBPS the number of bits per symbol and AC the EDCA access category.

And from the transmission iteration period, the maximum throughput rate is

$$r_{\text{throughput},Type}(b, N_\mathrm{NBPS}, \mathrm{AC}) = \frac{b \cdot 8\,\frac{\mathrm{bit}}{\mathrm{byte}}}{p_{Type}(b, N_\mathrm{NBPS}, \mathrm{AC})} \tag{7.1}$$

where $r_\mathrm{throughput}$ is in bit/s, due to multiplying bytes by eight.

For the more complex calculation of transmission iteration periods in a TXOP burst sequences, review figures 7.2(c) and 7.2(d). First the number of frames plus SIFS, which fit into one TXOPLimit, must be calculated, where the last frame is not followed by SIFS. Waiting time before the next transmission is not part of the TXOP. So the maximum number of frames in a burst with and without ACK can be determined as

$$n_\text{burst-NoACK}(b, N_\mathrm{NBPS}, \mathrm{TXOPLimit}) = \left\lfloor \frac{\mathrm{TXOPLimit} + 16\,\mu\mathrm{s}}{t_\mathrm{f}(b, N_\mathrm{NBPS}) + 16\,\mu\mathrm{s}} \right\rfloor$$

$$n_\text{burst-ACK}(b, N_\mathrm{NBPS}, \mathrm{TXOPLimit}) = \left\lfloor \frac{\mathrm{TXOPLimit} + 16\,\mu\mathrm{s}}{t_\mathrm{f}(b, N_\mathrm{NBPS}) + 16\,\mu\mathrm{s} + t_\mathrm{f}(b_\mathrm{ACK}, N_\mathrm{NBPS}) + 16\,\mu\mathrm{s}} \right\rfloor$$

	non-QoS frames				QoS frames				ACK
Payload bytes	80	200	400	2 304	80	200	400	2 304	14
6 Mb/s	180 µs	340 µs	608 µs	3 144 µs	184 µs	344 µs	608 µs	3 148 µs	44 µs
24 Mb/s	60 µs	100 µs	168 µs	804 µs	64 µs	104 µs	168 µs	804 µs	
54 Mb/s	40 µs	56 µs	88 µs	368 µs	40 µs	56 µs	88 µs	368 µs	

Table 7.1: Frame durations used in maximum throughput experiment

where 16 µs is the SIFSTime for 802.11a and TXOPLimit in (micro)seconds.

And from the number of frames in a transmission iteration superframe followed by the appropriate waiting time, the total iteration period is

$$p_{\text{burst-}Type}(b, N_{\text{NBPS}}, \text{AC}) = n_{\text{burst-}Type}(b, N_{\text{NBPS}}, \text{TXOPLimit[AC]}) \cdot \Big(t_{\text{f}}(b, N_{\text{NBPS}}) + 16\,\mu\text{s}\Big) - 16\,\mu\text{s} + \text{AIFS[AC]} + \frac{\text{CWmin}}{2} \cdot 9\,\mu\text{s}$$

again where 16 µs is the SIFSTime and 9 µs the SlotTime for 802.11a. The maximum throughput for TXOP burst sequences is thus calculated by inserting $p_{\text{burst-NoACK}}$ or $p_{\text{burst-ACK}}$ into equation 7.1.

For all experimentally tested configurations the corresponding ideal analytic values are shown in table 7.2.

Experimental Maximum Throughput

To verify EDCA and CSMA/CA rules implemented in ns-3, a maximum throughput experiment was run for each AC of the EDCA default parameters in 802.11e and 802.11p/D4.02. Three different wireless modes and four different packet sizes were tested, as listed in the introductory section of this scenario.

For each experiment configuration, 100 independent replications were run, where each run simulated 60 s time. For experiments with 2 304 byte packets the simulated time was increased to 120 s due to the longer packet duration. The mean throughput and 99% confidence interval were determined, and compared to the analytical mean value. Both ideal value and difference to the experimentally determined value are listed in table 7.2 for each tested configuration. The second value following the difference is the result's margin of error with 99% confidence level.

As listed in the table, the maximum margin of error over all experimental maximum throughput value is ±1 726 b/s for unACKed AC_BE with 802.11e and 54 Mb/s. This shows that random effects included in the experiment due to random backoff selection have only a very small influence on the result.

The difference between analytic and mean experimental maximum throughput result is at most 701 b/s for unACKed DCF traffic at 54 Mb/s. This impressively small difference shows that DCF/EDCA frame exchange sequences and the backoff procedure work as required for the single node case.

The next scenario focuses on collisions between multiple nodes and interruptions of EDCA backoff caused by carrier sensing.

7.4.2 EDCA Traffic Streams

The second experiment is designed to test the relative QoS provided by the EDCA implementation added to ns-3. In this scenario, multiple nodes are set up to send four independent data streams using the four different ACs provided by EDCA. When the number of nodes is increased, medium utilization increases as well and the four streams' access priority takes effect.

All nodes are located at the same virtual location, thus propagation loss and delay have no influence the experiment. One passive listener and n active senders are configured. Each sender transmits four traffic streams: all four streams consist of packets with 200 payload bytes sent at 160 Kb/s. The inter-packet waiting time is Poisson distributed with 100 packets per second on average. To analyzed relative QoS, the AC of the four traffic streams is set to VO, VI, BE and BK, by attaching a `WifiQosTag` to each generated packet. The packets are sent with the base wifi data rate of 6 Mb/s. Both unACKed broadcast and ACKed unicast transmission parameters were tested.

The number of sender nodes is increased stepwise from 1 to 30, of which each broadcasts or unicasts the four streams for 60 s simulated time. At the listener node all received packets are counted by AC. The received data rate of each AC is plotted in figure 7.4 for the four ACs, configured once with 802.11e and once with 802.11p/D4.02 defaults (see section B.2).

Discussion of the results must first focus on the two unACKed broadcast plots. As the number of nodes increases, medium load increases and relative QoS prioritization takes effect. The sharpest drop in throughput can be seen for AC_BK in both diagrams, while with seven nodes the total load is 4 480 kB/s and only

few background packets are not sent, with eight sender nodes and a total load of 5 120 kB/s a much larger amount of background traffic gets dropped. AC_VO, AC_VI and AC_BE priority packets clearly are sent with higher priority than AC_BK.

For higher channel loads, throughput of the lower priority traffic streams decreases as expected. Note that with 802.11e parameters, AC_VO and AC_VI have equal AIFSN = 2 and thus are treated equally in the long run. However, their throughput is reversed in figure 7.4(a): less VO packets are received than VI. This is due to the higher collision rate of VO packets on the medium: the packets are broadcasted and thus no ACK frames are generated. Because the random backoff interval of VO is only $[0 \dots 3]$, while that of VI is $[0 \dots 7]$, the collision rate of VO is higher and thus less packets are correctly received.

With 802.11p parameters (figure 7.4(b)), all AIFSN are different and therefore in the long run only VO traffic is sent. With between 13 and 20 sender nodes, the relative QoS of the three streams is most prominently shown: packets from all three categories are still sent, but different collision rates apply and prioritized throughput is achieved.

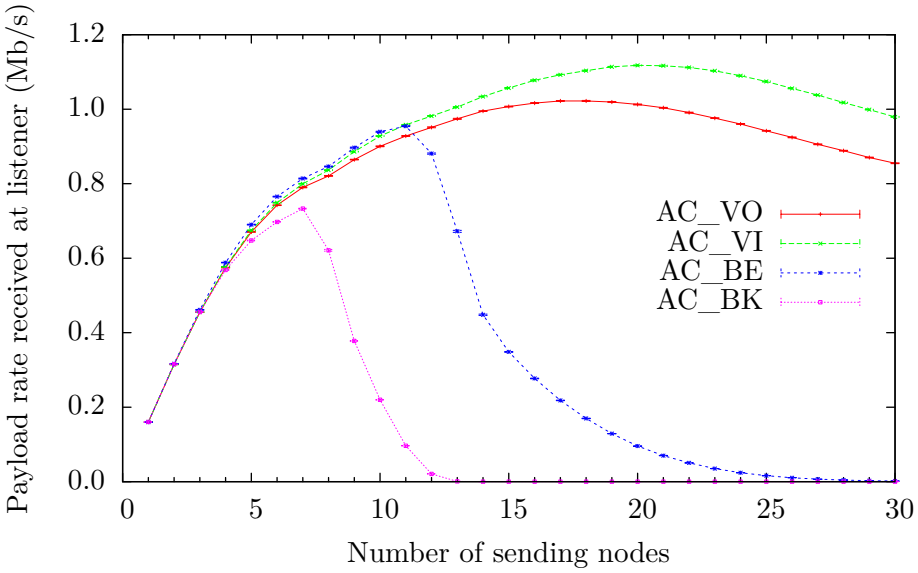
The second pair of plots contains experimental results for ACKed unicast transmissions of all n senders to the single listener. For 802.11e the expected priority of VO is now visible, due to fast retransmissions in case the first frame was lost. In both plots relative QoS, as provided by EDCA, can be seen in a more pronounced degree than in the unACKed experiments. Due to retransmissions, the channel load increases faster than with broadcast frames and thus the lower priority streams throughput falls more quickly.

Further experiments with ACKed packet streams like measuring delay, retransmissions and channel access time could be done. However, the two experiments with unACKed streams capture the core of EDCA prioritization: ACKs and retransmissions are not core EDCA components.

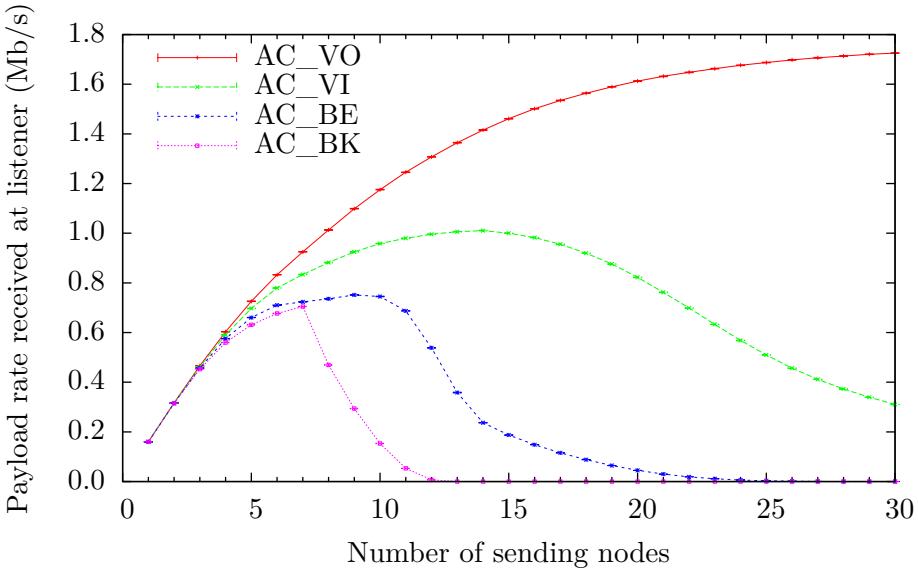
The verifying experimental results show that relative QoS is correctly provided by the EDCA implementation added to ns-3. By applying a `WifiQosTag` in existing ns-3 applications, research using relative QoS in wireless simulations is now possible.

Data rate	Analytic maximum throughput no ACK				with ACK				Difference to experimental result and margin of error no ACK				with ACK			
Payload bytes	80	200	400	2 304	80	200	400	2 304	80	200	400	2304	80	200	400	2304
DCF																
6 Mb/s	2 273 535	3 624 009	4 510 218	5 679 248	1 874 085	3 190 429	4 158 545	5 576 161	5/194	-9/ 222	143/ 212	80/ 108	36/150	78/228	45/ 207	115/ 93
24 Mb/s	3 962 848	7 940 447	11 873 840	20 355 605	2 889 391	6 118 547	9 711 684	19 090 627	42/425	-158/ 702	116/ 928	-13/ 613	-9/274	7/607	177/ 693	260/ 702
54 Mb/s	4 522 968	10 158 730	16 886 544	39 258 786	3 176 179	7 356 322	12 825 651	34 810 198	46/514	-510/1 036	179/1 708	701/1 661	23/317	105/819	356/1 053	474/1 377
802.11e / AC_VO																
6 Mb/s	3 129 584	4 349 303	5 001 954	0	2 403 305	3 716 609	4 573 062	0	9/ 30	14/ 36	38/ 42	0/ 0	16/ 22	12/ 37	35/ 39	0/ 0
24 Mb/s	7 837 577	13 047 910	17 026 937	21 646 506	4 470 835	8 698 607	12 838 516	20 221 613	40/ 64	-11/ 110	7/ 161	85/ 171	-6/ 35	9/ 76	-2/ 124	44/ 170
54 Mb/s	11 195 335	21 768 707	30 117 647	46 722 433	5 404 352	11 863 836	19 104 478	40 554 455	39/101	70/ 192	12/ 244	-109/ 293	3/ 45	28/ 92	114/ 180	128/ 303
802.11e / AC_VI																
6 Mb/s	3 148 057	4 369 346	5 028 482	0	2 419 660	3 746 446	4 595 225	0	1/ 42	-9/ 56	58/ 60	0/ 0	20/ 33	3/ 46	36/ 55	0/ 0
24 Mb/s	7 868 417	13 116 904	17 103 725	22 034 668	4 495 735	8 738 693	12 896 725	20 559 955	40/105	-53/ 152	198/ 223	128/ 241	-12/ 55	66/124	71/ 166	167/ 198
54 Mb/s	11 244 510	21 864 324	30 272 386	47 132 055	5 428 152	11 918 063	19 190 405	40 756 219	67/142	113/ 266	40/ 385	96/ 432	55/ 60	-60/149	-146/ 206	132/ 428
802.11e / AC_BE																
6 Mb/s	2 173 175	3 520 352	4 453 723	5 656 590	1 805 360	3 109 815	4 110 469	5 554 317	102/173	-144/ 242	5/ 214	88/ 107	-116/136	-28/182	7/ 212	113/ 94
24 Mb/s	3 667 622	7 459 207	11 490 126	20 155 276	2 729 211	5 828 780	9 453 471	18 914 315	-99/375	-187/ 687	155/ 998	-10/ 611	-79/233	-110/485	146/ 686	258/ 689
54 Mb/s	4 252 492	9 609 610	16 120 907	38 520 376	3 040 380	7 064 018	12 379 110	34 228 412	-202/477	-324/1 006	66/1 726	470/1 598	-85/267	-282/642	-33/1 018	459/1 353
802.11e / AC_BK																
6 Mb/s	1 936 460	3 261 978	4 241 219	5 594 779	1 638 924	2 906 449	3 928 791	5 494 709	21/152	4/ 201	232/ 230	85/ 107	7/115	151/178	14/ 203	114/ 92
24 Mb/s	3 040 380	6 387 226	10 174 881	19 391 899	2 365 989	5 152 979	8 544 726	18 240 475	236/302	104/ 582	72/ 861	-87/ 575	12/202	-313/378	151/ 620	257/ 650
54 Mb/s	3 431 635	7 901 235	13 646 055	35 825 073	2 596 349	6 095 238	10 865 874	32 083 551	292/392	319/ 780	-126/1 188	154/1 430	41/229	-322/482	193/ 902	230/1 245
802.11p / AC_VO																
6 Mb/s	2 764 579	4 086 845	4 881 770	5 768 111	2 195 540	3 543 743	4 472 397	5 661 803	33/ 62	33/ 74	21/ 65	74/ 29	23/ 38	-18/ 64	8/ 63	83/ 24
24 Mb/s	5 739 910	10 561 056	14 849 188	21 646 506	3 731 778	7 565 012	11 615 245	20 221 613	113/177	113/ 304	-67/ 333	31/ 164	30/ 84	-10/187	24/ 270	65/ 178
54 Mb/s	7 314 286	15 458 937	23 616 236	44 361 011	4 338 983	9 785 933	16 368 286	38 763 407	212/273	228/ 516	-309/ 636	236/ 525	39/101	51/282	-19/ 435	249/ 390
802.11p / AC_VI																
6 Mb/s	2 661 123	3 995 006	4 815 651	5 751 911	2 129 784	3 474 484	4 416 839	5 646 194	-28/ 55	19/ 65	11/ 64	80/ 28	15/ 42	46/ 59	-32/ 55	87/ 26
24 Mb/s	5 311 203	9 968 847	14 253 898	21 420 105	3 545 706	7 256 236	11 247 803	20 023 900	-37/174	11/ 258	45/ 281	16/ 160	-16/ 96	39/185	-156/ 218	79/ 175
54 Mb/s	6 632 124	14 222 222	22 145 329	43 420 495	4 089 457	9 275 362	15 647 922	38 043 344	-42/236	26/ 522	-108/ 539	197/ 513	16/112	69/269	-336/ 365	238/ 372
802.11p / AC_BE																
6 Mb/s	2 241 681	3 591 470	4 510 218	5 672 257	1 852 388	3 165 183	4 158 545	5 569 421	-19/ 90	-49/ 112	-5/ 133	78/ 53	-19/ 63	0/ 92	-17/ 117	94/ 47
24 Mb/s	3 867 069	7 785 888	11 873 840	20 355 605	2 838 137	6 026 365	9 711 684	19 090 627	-86/203	-62/ 363	68/ 516	24/ 307	-18/123	-57/258	87/ 357	140/ 347
54 Mb/s	4 522 968	10 158 730	16 886 544	39 258 786	3 176 179	7 356 322	12 825 651	34 810 198	-120/253	-201/ 564	22/ 948	340/ 828	-49/141	-121/338	-49/ 538	285/ 682
802.11p / AC_BK																
6 Mb/s	1 836 442	3 146 509	4 142 395	5 564 377	1 566 707	2 814 424	3 843 844	5 465 382	33/139	-63/ 186	118/ 185	81/ 106	15/118	33/186	48/ 183	126/ 89
24 Mb/s	2 800 875	5 959 032	9 624 060	19 031 492	2 218 371	4 870 624	8 152 866	17 921 245	92/259	113/ 533	-19/ 754	-136/ 576	54/158	-89/411	-86/ 645	227/ 640
54 Mb/s	3 129 584	7 256 236	12 673 267	34 614 085	2 419 660	5 704 100	10 240 000	31 108 861	-182/302	172/ 651	102/ 980	143/1 373	48/223	104/506	-593/ 797	191/1 196

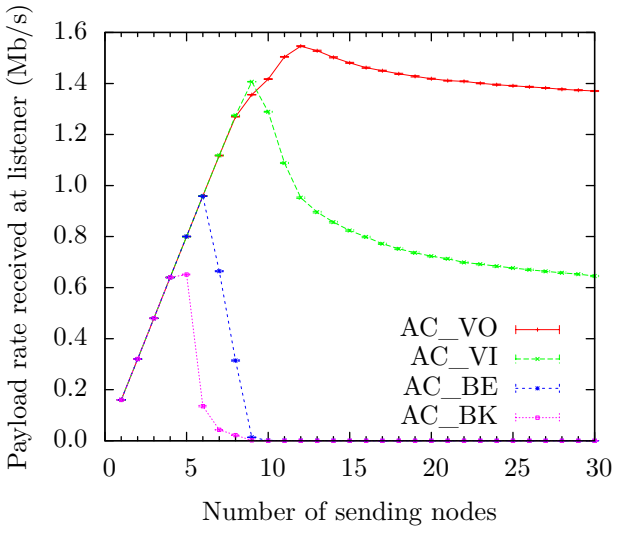
Table 7.2: Analytic maximum throughput and difference to experimental data



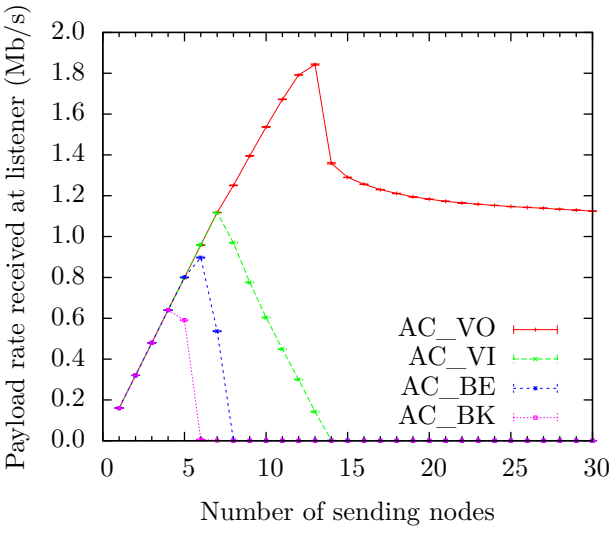
(a) UnACKed broadcast with EDCA parameters in 802.11e



(b) UnACKed broadcast with EDCA parameters in 802.11p/D4.02



(c) ACKed unicast with 802.11e



(d) ACKed unicast with 802.11p/D4.02

Figure 7.4: EDCA traffic streams broadcast throughput

Chapter 8

Speed Comparison – ns-2 vs. ns-3

Simulation run time is a major aspect for researchers wanting to switch to ns-3. If the required experiment models and applications are present in ns-3, is there a performance gain that can be achieve by switching from ns-2? This section gives an answer to this question for wireless simulations.

Besides the performance consideration, other aspects should also be taken in account. The overall better design and code quality of ns-3 is of higher value for reliable experiments. Moreover the complete restructuring of ns-3's core and rewritten individual models makes a direct comparison impossible. "Switching to ns-3" is more a process of reengineering scenarios than of porting Tcl code.

Nevertheless, in this thesis a direct performance comparison of ns-2 and ns-3 is done in for a complex wireless scenario. A sound, direct comparison was first made possible with the equal models implemented for this thesis. The speed comparison must also focus on the different influencing factors as compiler, build options and architecture. Possible future optimizations to increase performance of ns-3 are discussed after reviewing speed test results.

8.1 Highway Lanes Scenario

To compare ns-2 and ns-3, an abstract, reasonably large highway scenario was designed. All components used are available in both ns-2 and ns-3 and measured experimental results are absolutely equal.

A number of nodes is set up at fixed positions on a six lane highway as illustrated in figure 8.1. Multiples of 6 nodes are used for all experiment runs.

In the experiment the 802.11 layers are focused, other aspects as TCP, IPv4 or routing are not in the scope of this thesis and are omitted from the speed test experiment. Wireless communication is simulated in ns-2 with the `Mac802_11Ext` and `WirelessPhyExt` class, whereas in ns-3 the `Ns2ExtWifiPhy` and standard MAC modules are used. Thus the speed comparison encompasses the now compatible 802.11 MAC and PHY models, packet construction and passing between simulation nodes and many core components like the event scheduler and simulated time calculation.

The three-field log-distance propagation loss model is used without Nakagami fast fading in ns-2, and in ns-3 the corresponding `ThreeLogDistancePropagationLossModel`, added for this thesis, is configured with equal parameters.

All nodes periodically broadcast beacon-like packets at 10 Hz containing 400 bytes payload. Start time of the isochronous packet stream is uniform randomly distributed with $[0 \ldots 0.1]$ seconds.

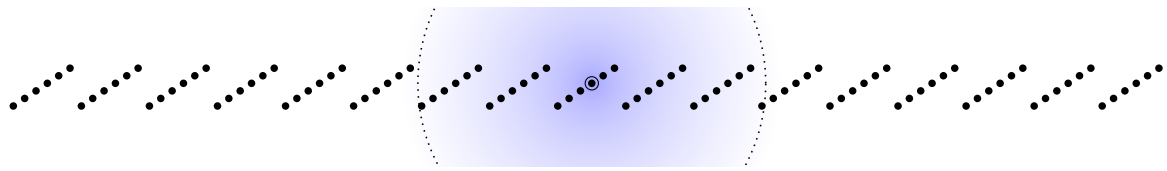


Figure 8.1: Highway lanes scenario with 102 nodes

Packets are sent at the basic 6 Mb/s 802.11a data rate with 20 dBm transmit power, the carrier sense threshold is set to -84 dBm and the noise floor is -99 dBm. Reception is determined by the SINR threshold of 5 dB, which is the default in both employed PHY models. Frame capture is also activated in both simulators with the default thresholds.

Only a very simple measurement is performed in the experiment: all sent and received packets are counted over all nodes in the scenario. Furthermore, reception errors are measured to check for collisions, but these values are not comparable in ns-2 and ns-3 due to the different effects they represent, and excluded from the further analysis.

Great effort was put into making the experiment run equally on ns-2 and ns-3. Many problems and bugs were encountered in this process. The following list contains some of the troublesome differences and, if possible, how they were rectified.

- Start time of the periodic packet streams must be randomly distributed over one beacon interval. To make both simulators produce equal results, these phase offsets were determined with a simple pseudo-random function instead of using the complex random generators of each simulator. The random function depends only on node number.
- For packet generation, the `PBCAgent` was used for periodic broadcast in ns-2, whereas in ns-3 the `TrafficApplication` class was configured with constant intervals. Getting these two applications to produce packets at exactly the same simulated time points was challenging.
- As described in previous sections, ns-2 does not add the eight byte LLC/SNAP header. This difference is remedied by increasing the packet payload to 408 bytes.
- While ns-2 does not include the LLC header, it does simulate a link layer (LL) delay of by default 25 µs. This delay was deactivated to attain identical packet sending times.
- Reception power calculation in ns-2 is primarily done in watt, with conversion to dBm were necessary for the log-distance model. Whereas in ns-3 it is generally done in dBm. This difference may introduce some rounding errors.
- Simulated time in ns-2 and ns-3 is fundamentally different, because ns-2 uses a simple *double* type, while ns-3 uses an exact 128-bit integer representing nanoseconds. Inaccuracies introduced by *double* rounding are difficult to grasp.
- Results are gained in ns-2 by analysing the trace file *after* simulation finish, whereas in ns-3 statistical analysis runs *alongside* the simulation. To alleviate this difference, the trace output of ns-2 was directly coupling to an analysis program, as will be described in the following section.

Despite all these small differences between the simulators, and after fixing a critical bug in the cumulative noise implementation of ns-2, the experimental results are exactly equal.

Moreover, a detailed tracing of exact packet send and arrival time points showed only very small variations between ns-2 and ns-3. These small variations, in the range of microseconds, were due to fundamentally different representations of simulated time, and do not influence the experimental results.

8.1.1 Compilers, Optimization Levels and Build Options

The speed comparison was done using ns-2.33, with an updated 802.11 patch from the DSN, and ns-3.4 with extensions made for this thesis. These will be merged into a future ns-3 release.

The test machine contained a quad-core 64-bit Intel Xeon CPU clocked at 3.0 GHz and had sufficient RAM to hold the complete running simulation. Great care was taken that no other CPU-consuming program was running during the speed measurements, which is attest by the result's error margins. All tests were performed on a standard, up-to-date Linux platform provided by the Ubuntu 8.10 (Intrepid) distribution.

Two different compilers were used to translate the source code: `gcc`, the GNU C/C++ compiler in version 4.3.2, and `icc`, the Intel C/C++ compiler in version 11.0.074 [16]. Use of the special Intel compiler was

said to give a performance boost, particularly for complex software. Many minor modifications of the ns-3 code were necessary to make compilation with `icc` work correctly, in particular the `waf` build system was adapted and some code portions using non-standard constructs were rewritten. ns-2 compiled using `icc` without modifications.

ns-2 was built in three different modes: default debug mode with `gcc` compiler and no optimization (`-O0`), optimized `gcc` mode with `-O3` compiler flag and using `icc` with `-O3` optimization level.

ns-3 allows two different build modes: *debug* and *optimized*. In optimized build mode, the logging and assertion macros like `NS_LOG()` and `NS_ASSERT()` are deactivated for maximum performance. Both `gcc` with `-O0` and `-O3` optimization levels and `icc` with `-O3` were tested.

Beyond these code compilation options, another architectural build option is available in ns-3. By default ns-3 is built as a *shared library* called `libns3.so` and experiments are *dynamically linked* against it. During this thesis an alternative, experimental method called *static linking* was tested. Details on two linking mechanisms will be explained when discussing speed test results.

A last configuration option was added to compare 32-bit and 64-bit architectures. The ns-3 `Time` representation uses 128-bit integers, which are manipulated using code that makes optimized use of 64-bit CPU registers. Fallback 32-bit register code exists as well, but suggests a large performance loss. However, testing the speed increase of 64-bit registers proves difficult, as time measurements from different machines cannot be compared. So instead, 32-bit emulation mode as available in the 64-bit Intel Xeon architecture was used. However, it is questionable if the obtained speed test results in this emulation mode are relevant for real 32-bit CPUs.

The speed test scenario results are fully deterministic: propagation loss is modeled only with log-distance path loss, packet broadcast intervals are constant and broadcast offsets are determined with a pseudo-random function depending only on node identifier. Furthermore, the time delta between two broadcast packet far exceeds any DCF waiting interval and thus random backoff has no influence.

All speed test results are determined over 10 independent replications. The total time required to run the binary experiment program is measured.

This leads to the question whether statistical evaluation of the experiment should be included in the measured time. In ns-2 analysis is usually done by first writing a (rather large) trace file and creating statistics after the experiment finishes, whereas ns-3 allows calculations of statistics during the experiment's execution. This architectural difference obviously favors ns-3, because writing to disk files is relatively slow. To alleviate this difference, the trace output of ns-2 was written to a named pipe directly connected to a statistics program running in parallel. Processing time of the evaluation program was included in the time measurement.

8.1.2 Execution Time Results

The highway lanes scenario was run for three different build configurations of ns-2 and seven build configurations of ns-3.

Furthermore, two extra runs with activated Nakagami fast fading were done, one in ns-2 and the other in ns-3. These two extra runs are not comparable to the others, and are used to determine the additional processing time required to generate random variates. Discussion of these two extra runs is restricted to section 8.1.2.

The execution time of 10 runs was measured for each experiment setup containing multiple of 6 nodes from 6 to 120. Average run value and 99% error margin are plotted in the result figure 8.3. The two plots contain the same data, one is shown with linear time scale for an overview and the other with logarithmic time scale for a detailed look at small time values.

Furthermore, the total number of sent and received packets were counted across all nodes in the simulation. These packet sums are shown in figure 8.2.

A brief glance at the packet count plots shows that in all simulations approximately the same number of packets are transmitted and received in each speed test configuration. This is also reflected by the exact data values: for k nodes in an experiment *exactly* $600 \cdot k$ packets are sent.

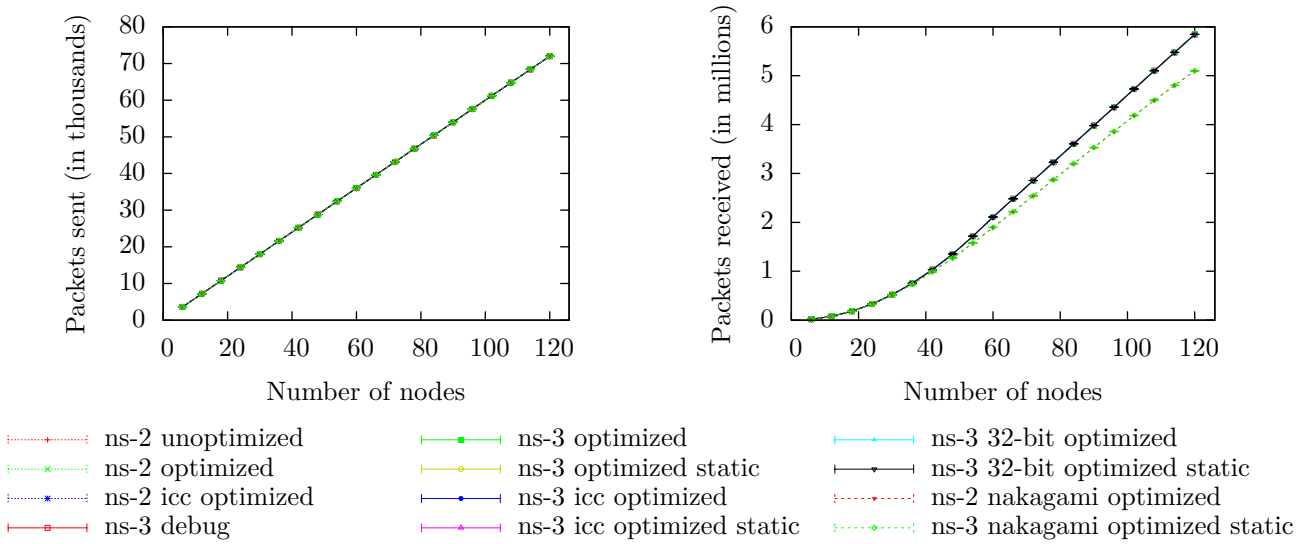


Figure 8.2: Total packets sent and received during simulation

The number of received packets is also *exactly equal* for all build configurations of ns-2 and ns-3. However, the results follow no simple proportional equation like the sent packets. Complex inter-packet contention, channel load and the frame capture effect all play a role in these results. It is remarkable that they are absolutely equal on both simulators.

Despite the identical experiments, measured execution time results in figure 8.3 range widely. Slowest simulation configuration in ns-3 in debug mode with more than twice the execution time of unoptimized ns-2. Optimizing ns-2 with `-O3` brought about a moderate speed increase of 18.9±0.5%, compilation with `icc` a further 4.1±0.3%. The percentages denote the execution time reduction as a 99% confidence interval over all numbers of nodes tested.

Greatest speed increase was measured by compiling ns-3 in optimized mode. This attests ns-3 a high number of assertion and logging macros or just easily optimized code. The speed reduction when building in optimized mode was 76.3±0.5%.

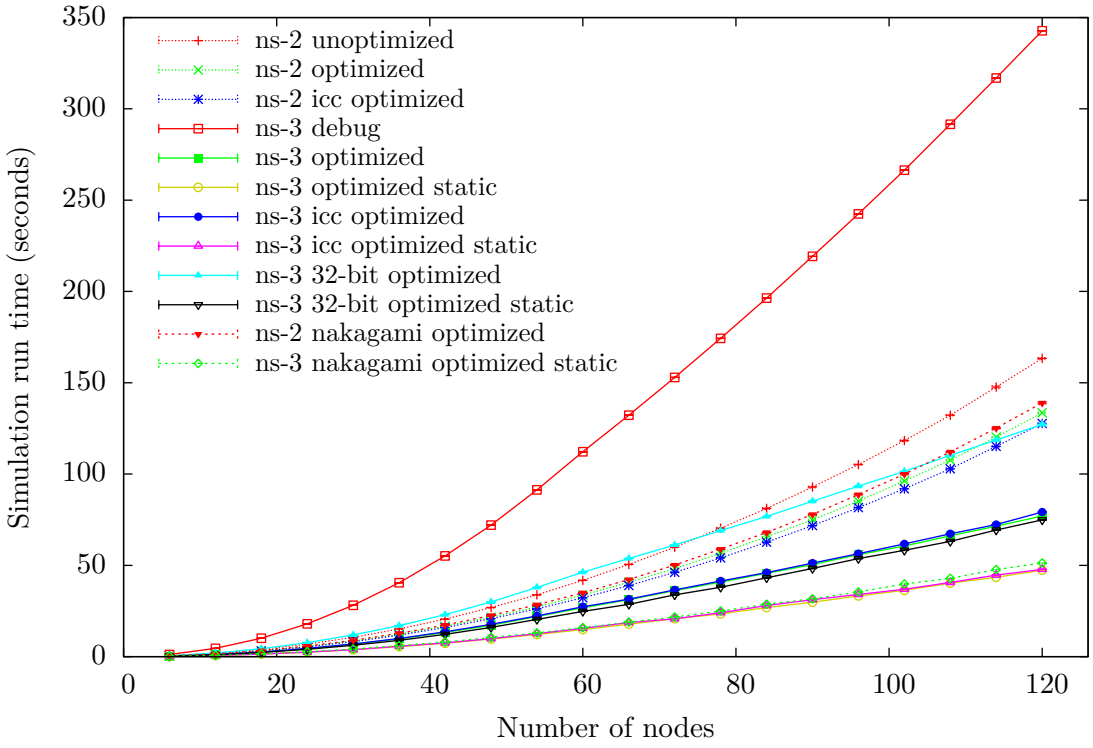
Really unexpected was the further speed decreased of ns-3 that could be achieved by *static linking* of the experiment object files. This build method brings a further 42.6±1.2% on top of the already optimized build mode. In total the compiling ns-3 in optimized static mode, brings a 86.4±0.2% run time decrease relative to debug mode. The reason for this further gain using static linking is examined in the following section 8.1.2.

Contrary to expectation, use of the Intel C/C++ compiler does not show great speed improvement. Instead, run time of ns-3 compiled with `icc` is on average 1.9±0.4% *slower* than the corresponding `gcc` configuration for dynamically linked binaries, and 2.5±0.9% slower with static linking. The binaries created by `icc` are also slightly larger. This result suggests the optimizations done by `gcc -O3` is more suitable for ns-3.

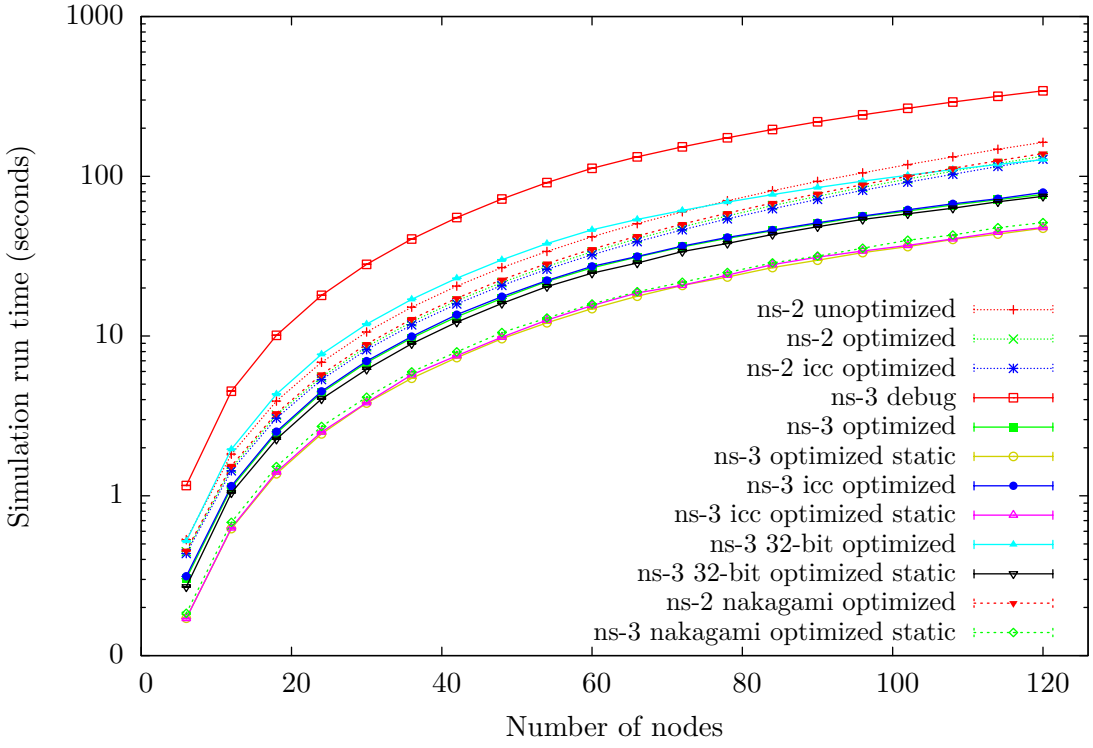
For the two 32-bit tests, the execution speed drops as expected due to less optimized processing of time representation. Some profiling of code showed that a large percentage of run time is spent in calculations of simulated time. The 32-bit speed results showed were 70.9±2.0% slows for dynamic linking and 62.8±2.0% slower for static linking relative to the corresponding 64-bit configuration. As stated before, interpretation of this result is difficult and not directly applicable for real 32-bit CPUs. However, even for the 32-bit code, static linking decreased run time by 45.3±1.3% over dynamic linking.

All these results hold true for all tested experiment sizes from 6 to 120 nodes. This can be seen in the logarithmic plot in figure 8.3 and is underlined by the small error margins in reduction percentages shown above.

Overall the speed comparison between ns-2 and ns-3 shows that “switching” to ns-3 can reduce experiment execution time to 58.6±1.8%. This is the relative speed gain from `gcc` optimized ns-2 to `gcc` optimized and statically linked ns-3.



(a) Linear time scale



(b) Logarithmic time scale

Figure 8.3: Time measurements of different compilers, optimization levels and build options

Static vs. Dynamic Linking

Statically linking the experiment binaries showed a reduction of execution time of at least 42% over all optimization and compiler combinations. For long simulations this methods should be preferred over linking with a *shared library*. However, besides creating rather large experiment executable binaries (about 30 Mb), static linking introduces difficulties in context of building language bindings for e.g. Python, because these require shared libraries containing position-independent code (PIC).

The reason for faster statically linked simulations is connected to the mechanism of *dynamic linking*. The general idea is well-known: to export commonly-used code portions into external files (called `.so` or `.dll`), which are then shared between multiple programs and loaded on-demand upon program start up or during execution. Each shared library exports a list of *symbols*, that are mapped to memory locations of functions or variables. For C++these symbols are "mangled" serializations of namespace, class, function name and arguments.

Beyond this general idea, precise information on the dynamic linking routine, as done on Linux by `ld.so`, is expert knowledge and difficult to find. The dynamic linker is part of the glibc and every executable run on Linux is loaded by it (except for itself). The current glibc maintainer, Ulrich Drepper, describes the loading procedure and implications for authoring shared libraries in a technical guide [5]. All referenced external symbols, including variables and functions imported from a `.so`, are contained in the Global Offset Table (GOT), which is a mapping to the corresponding addresses. Referenced functions can be called via an indirection call to the Procedure Linkage Table (PLT), which jumps to the appropriate address listed in the GOT. This indirection jump is omitted when statically linking objects: here a direct function call is used. For long running simulations the indirection outweighs the initial costs of library loading and (hashed) dynamic symbol lookup etc.

Another disadvantage of dynamic linking is that it assumes that all functions can be replaced at run-time. This prohibits inlining of the many functions generated by C++ templates say for ns-3's callback or smart pointer idioms.

The additional cost of PIC was also examined alongside this speed test, but omitted from the previous plots and discussion. PIC is unavoidable for generating language bindings in shared libraries, and thus the complete ns-3 code would have to be compiled twice to allow C++ experiment binary to use a non-PIC version. PIC is said to make shared library code slower, because an extra CPU register is used to make address positions relative. However, the speed test conducted with an optimized, statically linked ns-3 version compiled without PIC showed only a reduction of 1.1±0.3% over PIC in the speed test experiment. Thus this optimization proposal need not be pursued further.

As the speed increase of static linking is substantial, ns-3 will need to support this build option despite the disadvantages discussed before. Some work has begun to improve inlining by declaring these short functions to be *hidden*. With this flag they are not exported via the dynamic linking mechanism and cannot be replaced by shared libraries possibly loaded later.

Nakagami Propagation

Two extra speed test configurations were done with Nakagami propagation loss modeling. All other scenario parameters were kept the same, and thus reception power was probabilistically distributed over the original log-distance power. Figure 8.2 show that the number of sent packets stayed equal, whereas the successfully received packet count decreased. For this experiment ns-2 and ns-3 cannot show exactly the same received packet counts because of stochastic propagation, nevertheless results are very similar, as seen in the second graph.

Aim of this extra speed test was not to compare ns-2 and ns-3, but to see how much computation time is used for generating gamma variates. For ns-3 the simulation time increased by 8.1±1.0% relative to the same optimized compilation configuration without Nakagami propagation; for 120 nodes this was 4.0 s of the total 51.3 s run time.

In ns-2 the relative time increase required for Nakagami propagation was only 3.8±0.4%. However absolute time increase was higher than for ns-3: for the ns-2 experiment with 120 nodes Nakagami propagation

increased run time by 5.4 s from 133.6 s to 139.0 s.

So these two extra speed tests show that the increased processing time of Nakagami propagation values is low and thus usable for large simulations.

Further Optimization

Several idea for further optimization of ns-3 were already mentioned. Statically linking yields large performance increases and is being added for the next release of ns-3. Intel's C/C++ compiler does not show the expected performance boost. Optimization of random variates generators can be pursued, yet only a small increase in overall performance is achievable.

One hot topic currently dormant is extending ns-3 to run parallel or distributed simulations. In 2008, a project was conducted in which a federated nodes approach using message passing interface (MPI) was taken. Multiple nodes deemed "close" to each other were grouped into a federation located inseparably on one processing machine. Packets sent across federation boundaries were serialized into real network packets and sent to the appropriate processor via MPI. Conservative synchronization using MPI mutexes was envisioned.

For the current ns-3 wireless network models no great speed improvement should be expected from this very general approach. This is due to the broadcasting delivery of packets to all devices attached to the emulated medium. This delivery range can be reduced to include only stations effected by the packet, but for most scenarios these will be all simulated nodes. Particularly, because cumulative noise is currently calculated within the PHY layer model, all packets contributing to the noise level must be processed. With a more simplified physical layer model, higher parallelization could be achieved.

Currently ns-3 is not well prepared for multi-threading simulations. To allow multiple threads running parallel on one simulation experiment, access to the state of all employed network objects would need to be synchronized via mutexes. Complex interrelationships and dependencies between network objects would quickly lead to deadlocks, and these must be handled or avoided.

A viable distributed simulation approach using "ghost nodes" was tested in the GTNetS [30] simulator by George Riley, who is also a core contributor to ns-3. A future project may be conducted to implement this technique in ns-3. This should be done while the project still has manageable size for extensive changes.

Chapter 9

Conclusion

9.1 Summary

In this thesis the medium and physical layer models employed by wireless simulations in ns-2 and ns-3 were compared and enhancements of ns-2 were transferred to ns-3. These new models embed cleanly in the existing 802.11 design and function identically to the models contained in ns-2.

Both the new SINR based PHY model and the existing BER/PER model were thoroughly explained in this thesis. Moreover, the Nakagami fast fading propagation loss model contained in ns-2 was ported to ns-3 and verified to produce identical results.

On the 802.11 MAC layer, relative QoS was added by implementing EDCA extensions in ns-3. These can be activated by creating QoS-enabled wireless station in the simulation and applying QoS specifier tags to generated packets. The extensions were tested in a maximum throughput experiment against analytically determined reference values. A second experiment showed how different relative QoS settings effected an increasing number of traffic streams. The EDCA contribution of this thesis enables interesting experiments with relative QoS in mobile wireless networks.

To assess the performance gain of ns-3 over ns-2, a complex wireless scenario was built in both simulators using the models added in this study. They enabled a convincing speed comparison between ns-2 and ns-3 with identical experiments producing equal results. Multiple different compilation settings and linking methods were tested. The results and their causes were discussed and potential optimizations outlined. Execution time of the tested experiment in ns-3 is reduced by up to 58.6% over the identical simulation in ns-2.

Further advantages of ns-3 over ns-2 are the state-of-the-art design and modern software engineering methods employed, which lead to an overall improved quality of code. All models in ns-3 are extensively documented and can be configured flexibly to individual experiments. The project's policy of using regression trace tests and model validation tests increases the reliability of simulation in ns-3.

The equal 802.11 PHY models of ns-2 and ns-3 presented in this thesis allow an easy switch to the new simulator. These models are at the heart of every wireless simulation and are thoroughly discussed and verified. Moreover, the added EDCA extensions allow new and exciting experiments with relative QoS in ns-3.

9.2 Future Work

Both extensions of 802.11 and simulations with ns-3 are very active fields of work.

The new upcoming 802.11p and 802.11n amendments will enable new applications with wireless LAN. In ns-3 802.11p can be simulated by using the 802.11a model and modified propagation model parameters. Adaption to 10 MHz channels is a matter of changing bandwidth values.

Concurrent to this thesis, a project at the University of Florence focused on the 802.11n frame aggregation and block ACKs feature. Initial support is already available for ns-3. Future plans are to add multiple input

and multiple output (MIMO) and 40 MHz channel bonding, to simulate the high data speeds promised by the amendment.

Another ongoing project is conducted by the Russian Academy of Sciences and aims at adding 802.11s wireless mesh networking support to ns-3. The projected architecture is composed of a plugin attached to the lower MAC layers, a mesh-capable routing layer with automatic station detection and a virtual `NetDevice`, which forwards packets to stations across multiple hops in the mesh.

An interesting project was recently started at the University of Padova and aims at modeling the power spectral density of transmissions across the whole radio spectrum. Targeted research fields are inter-technology interference and cognitive radio band selection.

A completely different standard is focused by the 802.16 WiMAX implementation currently being developed for ns-3. The scope of this project and the WiMAX standard are much broader than 802.11 and include different transmission schedulers and bandwidth managers at the base station.

Bibliography

[1] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley, 5th edition, 2002. ISBN 0-201-70431-5. 109

[2] Qi Chen, Felix Schmidt-Eisenlohr, Daniel Jiang, Marc Torrent-Moreno, Luca Delgrossi, and Hannes Hartenstein. Overhaul of IEEE 802.11 modeling and simulation in ns-2. In *MSWiM '07: Proceedings of the 10th ACM Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 159–168, New York, NY, USA, 2007. ACM. 27, 56, 57

[3] Claudio Cicconetti, Luciano Lenzini, Enzo Mingozzi, and Giovanni Stea. A software architecture for simulating IEEE 802.11e HCCA. In *IPS-MoMe '05: Proceeding from the 3rd Workshop on Internet Performance, Simulation, Monitoring and Measurement*, pages 97–104, March 2005. 26

[4] Peter T. Davis and Craig R. MacGuffin. *Wireless Local Area Networks.* McGraw-Hill, New York, NY, USA, 1995. ISBN 0-07-015839-8. 5

[5] Ulrich Drepper. *How To Write Shared Libraries.* Version 4.0, August 2006. `http://people.redhat.com/drepper/dsohowto.pdf`. 92

[6] Pal Frenger, Pal Orten, Tony Ottosson, and Arne Svensson. Multi-rate convolutional codes. Technical Report 21, Department of Signals and Systems, Chalmers University of Technology, Göteborg, April 1998. 66

[7] Harald T. Friis. A note on a simple transmission formula. In *Proceedings of the Institute of Radio Engineers (IRE)*, volume 34, issue 5, pages 254–256, May 1946. 43

[8] Deyun Gao, Jianfei Cai, and King Ngi Ngan. Admission control in IEEE 802.11e wireless LANs. In *IEEE Network*, volume 19, issue 4, pages 6–13, July-August 2005. 24

[9] Andrea Goldsmith. *Wireless Communications.* Cambridge University Press, New York, NY, USA, 1st edition, 2005. ISBN 0-521-83716-2, 978-0-521-83716-3. 45, 60, 61, 62, 63, 64

[10] Thomas R. Henderson. ns-3 tutorial. Presentation at *WNS3 '09: Workshop on ns-3 in conjunction with SIMUTools '09*, March 2009. `http://www.nsnam.org/workshops/wns3-2009/ns-3-tutorial-part-1.pdf`. 32

[11] Thomas R. Henderson, Sumit Roy, Sally Floyd, and George F. Riley. ns-3 project goals. In *WNS2 '06: Proceeding from the 2006 Workshop on ns-2: the IP network simulator*, New York, NY, USA, October 2006. ACM. 29

[12] IEEE Std. 802.11-2007 (Revision of IEEE Std. 802.11-1999). *IEEE Standard for 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*, December 2007. 7, 9, 10, 11, 12, 14, 15, 18, 20, 24, 81, 82

[13] IEEE Std. 802.11e. *IEEE Standard for 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 – Amendment 8: Medium Access Control (MAC) Quality of Service Enhancements*, November 2005. 19, 21, 22, 106

[14] IEEE Draft Std. 802.11p/D4.02. *IEEE Draft Standard for 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 – Amendment 7: Wireless Access in Vehicular Environments*, September 2008. 12, 21, 22, 23, 106

[15] IEEE Std. 802.1D-2004 (Revision of IEEE Std. 802.1D-1998). *IEEE Standard for Local and Metropolitan Area Networks: Media Access Control (MAC) Bridges*, June 2004. 20

[16] Intel C/C++ compiler. `http://software.intel.com/en-us/intel-compilers/`. 88

[17] Teerawat Issariyakul and Ekram Hossain. *Introduction to Network Simulator NS2*. Springer, New York, NY, USA, October 2009. ISBN 978-0-387-71759-3, 0-387-71759-5. 25, 26

[18] William C. Jakes. *Microwave Mobile Communications*. Wiley, 1974. ISBN 0-471-43720-4. 45

[19] Srinivasan Keshav. REAL: A network simulator. Technical report, University of California at Berkeley, December 1988. 25

[20] Mathieu Lacage and Thomas R. Henderson. Yet another network simulator. In *WNS2 '06: Proceeding from the 2006 Workshop on ns-2: the IP network simulator*, New York, NY, USA, October 2006. ACM. 35, 60

[21] Stefan Mangold, Sunghyun Choi, and Norbert Esseling. An error model for radio transmissions of wireless LANs at 5 GHz. In *Proceedings of the 10th Aachen Symposium on Signal Theory*, pages 209–214, Aachen, Germany, September 2001. 60

[22] Stefan Mangold, Sunghyun Choi, Guido R. Hiertz, Ole Klein, and Bernhard Walke. Analysis of IEEE 802.11e for QoS support in wireless LANs. In *IEEE Wireless Communications*, volume 10, issue 6, pages 40–50, December 2003. 19

[23] George Marsaglia and Wai Wan Tsang. A simple method for generating gamma variables. *ACM Transactions on Mathematical Software*, 26(3):363–372, 2000. 47

[24] LBNL network simulator – ns version 1. `http://www-nrg.ee.lbl.gov/ns/`. 25

[25] The network simulator – ns-2. `http://www.isi.edu/nsnam/ns/`. 25

[26] The ns-3 network simulator. `http://www.nsnam.org`. 109, 123

[27] PDNS - parallel/distributed NS. `http://www.cc.gatech.edu/computing/compass/pdns/`. 30

[28] John G. Proakis. *Digital Communications*. McGraw-Hill, New York, NY, USA, 4th international edition, 2001. ISBN 0-07-232111-3, 0-07-118183-0. 60, 61, 62, 63, 64

[29] Theodore S. Rappaport. *Wireless Communications*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2002. ISBN 0-13-042232-0. 43, 45

[30] George F. Riley. The georgia tech network simulator. In *MoMeTools '03: Proceedings of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research*, pages 5–12, New York, NY, USA, 2003. ACM. 30, 34, 93

[31] George F. Riley, Talal M. Jaafar, Richard M. Fujimoto, and Mostafa H. Ammar. Space-parallel network simulations using ghosts. In *PADS '04: Proceedings of the eighteenth Workshop on Parallel and Distributed Simulation*, pages 170–177, New York, NY, USA, 2004. ACM. 30

[32] Jiho Ryu, Jeongkeun Lee, Sung-Ju Lee, and Taekyoung Kwon. Revamping the IEEE 802.11a PHY simulation models. In *MSWiM '08: Proceedings of the 11th International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 28–36, New York, NY, USA, October 2008. ACM. 55

[33] Simon R. Saunders and Alejandro Aragón-Zavala. *Antennas and Propagation for Wireless Communication Systems*. John Wiley & Sons, Hoboken, NJ, USA, 2nd edition, 2007. ISBN 978-0-470-84879-1. 45

[34] Felix Schmidt-Eisenlohr, Jon Letamendia-Murua, Marc Torrent-Moreno, and Hannes Hartenstein. Bug fixes on the IEEE 802.11 DCF module of the network simulator ns-2.28. Technical Report 2006-1, Institute of Telematics, Univeristy of Karlsruhe, 2006. 26

[35] Henrik Schulze and Christian Lüders. *Theory and Applications of OFDM and CDMA*. John Wiley & Sons, January 2006. ISBN 978-0-470-85069-5, 978-0-470-01740-1. 12

[36] International Telecommunication Union. *Radio Regulations – Volume 1, Article 5: "Frequency Allocations"*, 2004. 8

[37] Sven Wiethölter and Christian Hoene. Design and verification of an IEEE 802.11e EDCF simulation model in ns-2.26. Technical Report TKN-03-019, Technische Universität Berlin, November 2003. 26

[38] Yunpeng Zang, Lothar Stibor, Georgios Orfanos, Shumin Guo, and Hans-Jürgen Reumerman. An error model for inter-vehicle communications in highway scenarios at 5.9 GHz. In *PE-WASUN '05: Proceedings of the 2nd ACM International Workshop on Performance Evaluation of Wireless Ad-hoc, Sensor and Ubiquitous Networks*, pages 49–56, Montreal, Quebec, Canada, October 2005. ACM. 60

Appendix A

Background

A.1 A Note on Decibel

The *decibel* unit, as used in dB, dBm, dBW, dBi and others, is often confused and misunderstood. However, with some background information, the difference between these units and similar units, measuring the same quantities, is easily understood and mistakes prevented.

The most important point is that *decibel* (dB) is always a logarithmic, dimensionless value. Decibel is one tenth of the base unit *bel* (B).

$$x = \log_{10}(x)\,\mathrm{B} = 10 \cdot \log_{10}(x)\,\mathrm{dB}$$

Most often x is a ratio between two quantities, both expressed in the same dimension. Usually the numerator is a measured and the denominator a reference value.

With two values of the same imaginary unit X, say $3000\,\mathrm{X}$ and $30\,\mathrm{X}$, the ratio is

$$\frac{3000\,\mathrm{X}}{30\,\mathrm{X}} = 100 = \log_{10}(100)\,\mathrm{B} = 2\,\mathrm{B} = 20\,\mathrm{dB}$$

Thus "dB" is a notational unit, but not a separate dimension of measurement.

The most commonplace use for decibel is to measure sound pressure levels. For example a jet engine is said to have "150 dB" sound pressure level. Here the reference value is 20 µPa (rms), the assumed lowest threshold of human hearing. There is also a related unit dB(A) or dBA, for which the sound pressure values are weighted depending on frequency to account for sensitivity characteristics of the human ear.

In the domain of wireless communication and electronics the unit dBm (actually dBmW) is used to express a power level relative to 1 milliwatt. Less commonly dBW (or incorrectly just dB) is used with reference value 1 watt. To indicate the reference value a suffix to the "dB": dB-W and dB-mW or just dB-m. Nevertheless, dBm and dBW remain dimensionless ratios. A power level x can be expressed in watt, milliwatt, dBW and dBm.

For example $100\,\mathrm{mW} = 0.1\,\mathrm{W}$ can be expressed as

$$\frac{100\,\mathrm{mW}}{1\,\mathrm{mW}} = 10 \cdot \log_{10}\left(\frac{100\,\mathrm{mW}}{1\,\mathrm{mW}}\right)\,\mathrm{dBm} = 20\,\mathrm{dBm} = 10 \cdot \log_{10}\left(\frac{0.1\,\mathrm{W}}{1\,\mathrm{W}}\right) = -10\,\mathrm{dBW}$$

Yet the expression "$100\,\mathrm{mW} = 20\,\mathrm{dBm}$" is incorrect regarding units, as dBm is dimensionless. Nevertheless it is common to express power levels in dBm, even through dBm is not part of the International System of Units (SI), which uses watt for power. Translations between mW and dBm are denoted with "$\hat{=}$" in this thesis.

When calculating with dB and dBm care must be taken due to the logarithmic expression. To determine the sum of two power values given in dBm, it is incorrect to simple add both dBm values. For example $20\,\mathrm{dBm} + 2\,\mathrm{dBm} \neq 22\,\mathrm{dBm}$, instead $20\,\mathrm{dBm} + 2\,\mathrm{dBm} \mathrel{\hat{=}} 100\,\mathrm{mW} + 1.584\,\mathrm{mW} = 101.584\,\mathrm{mW} \mathrel{\hat{=}} 20.068\,\mathrm{dBm}$ (rounded to three decimals). However, calculating the ratio of two power levels is easy when both are given in dBm: here simple subtraction will suffice. The logarithm function maps multiplications to additions.

Thus it is possible to add $y\,\mathrm{dB}$ to $x\,\mathrm{dBm}$ values, but the meaning of the expression is that the power level x is increased $10^{\frac{y}{10}}$ times. This direct addition operation can be used to express the relationship between dBW and dBm:

$$\begin{aligned} x\,\mathrm{dBm} = 10\,\log_{10}\left(\frac{x}{1\,\mathrm{mW}}\right) &= 10\,\log_{10}\left(\frac{1000\cdot x}{1\,\mathrm{W}}\right) \\ &= 10\,\log_{10}\left(\frac{x}{1\,\mathrm{W}}\right) + 10\,\log_{10}(1000) = x\,\mathrm{dBW} + 30\,\mathrm{dB} \end{aligned}$$

So this concludes $x\,\mathrm{dBm} = x\,\mathrm{dBW} + 30\,\mathrm{dB}$ and $x\,\mathrm{dBm} - 30\,\mathrm{dB} = x\,\mathrm{dBW}$.

In context of propagation loss modeling an issue occurs in which distinction between dBm and dB becomes both important and difficult. The transmitted signal's power can be described in dBm or mW as desired. A propagation loss is usually described as attenuation of a transmitted signal. However, signal attenuation is measured relative to transmission power so it will usually be expressed in dB. Propagation loss model formulas can be viewed in two ways. First as a function of transmission power resulting in reception power and thus both are measured in dBm. And secondly as an attenuation formula relative to a reference power level and thus expressed in dB. Once the equation is viewed differentially and once as an absolute power equation.

In wireless communication another dB unit is frequently used in connection with antenna gain: dBi. Again this is a ratio, with the power of an isotropic antenna as reference value. This hypothetical antenna radiates energy uniformly in all directions. So dBi specifies dimensionless, directional gain compared to this ideal omnidirectional antenna. Contour diagrams of gain in dBi can often be found in antenna datasheets.

For reference, conversion formulas between W, mW, dBW and dBm are summarized in the following equation table.

$$\begin{aligned} x\,\mathrm{mW} &\mathrel{\hat{=}} 10\cdot\log_{10}(x)\,\mathrm{dBm} = (10\cdot\log_{10}(x) - 30)\,\mathrm{dBW} \\ x\,\mathrm{dBm} &\mathrel{\hat{=}} 10^{\frac{x}{10}}\,\mathrm{mW} = 10^{\frac{x}{10}-3}\,\mathrm{W} \\ x\,\mathrm{W} &\mathrel{\hat{=}} 10\cdot\log_{10}(x)\,\mathrm{dBW} = (10\cdot\log_{10}(x) + 30)\,\mathrm{dBm} \\ x\,\mathrm{dBW} &\mathrel{\hat{=}} 10^{\frac{x}{10}}\,\mathrm{W} = 10^{\frac{x}{10}+3}\,\mathrm{mW} \end{aligned}$$

Appendix B

Extra Figures and Tables

B.1 802.11a Convolutional Encoder

Figure B.1 shows the convolutional encoder defined in 802.11a as a state machine. See figure 2.4 for a diagram of the same encoder as a linear feedback shift register.

In the state machine, nodes are labeled with the contents of the six registers before reading an input bit. Each node has two outgoing edges: one for a zero input bit and one for a one input bit. Edges for zero bits are solid while edges for one bits are dotted. Each edge is labeled with the two bits outputted when making the corresponding transition.

In section 6.6.2 a simpler convolutional code is discussed in the context of bit error rate calculations.

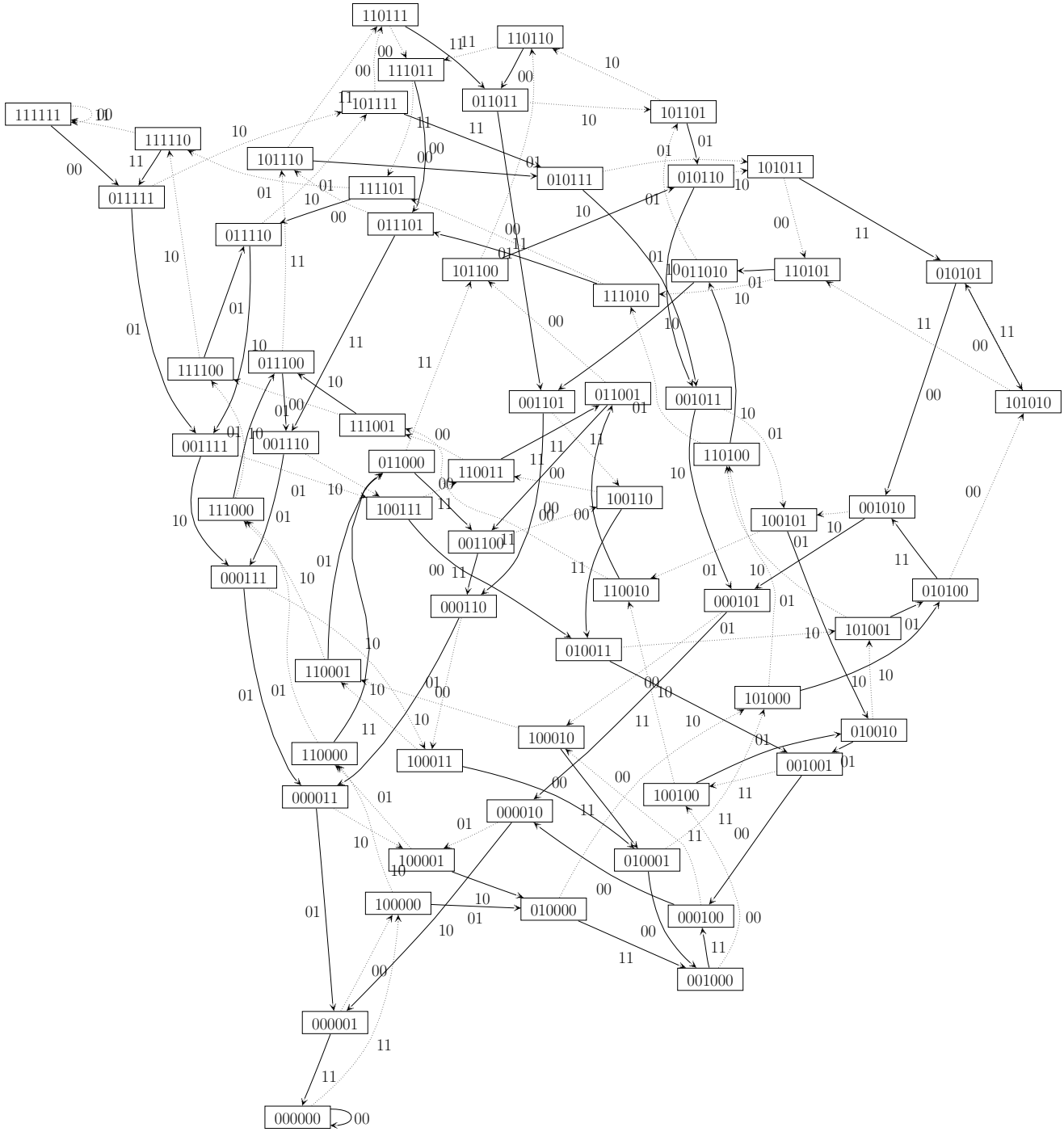


Figure B.1: State machine of the 802.11a convolutional encoder

B.2 Default EDCA Parameters

For use in an ad-hoc network (IBSS without a controlling AP) or outside the context of a BSS, the 802.11e standard [13] prescribes default EDCA parameters. The actual parameter values depend on the PHY layer used and are defined from PHY parameters by equations. These values may be changes by the SME to adapt EDCA prioritization as required. Like in the standard, the PHY parameters are prefixed in the table with an "a", which was omitted in the rest of this thesis.

An illustrated discussion of the parameter sets can be found in section 2.4.7.

Table B.2 shows the default EDCA parameter equations and values for the PHYs defined in 802.11a, b and g.

For communication outside a BSS the upcoming 802.11p [14] defines a set of parameters different from the one in 802.11e. These parameters are shown in table B.1.

ACI / AC		OFDM in 5.9 GHz band		
		20 MHz	10 MHz	5 MHz
1 / AC_BK				
AIFSN/AIFS	9	97 µs	149 µs	253 µs
CWmin	aCWmin	15	15	15
CWmax	aCWmax	1 023	1 023	1 023
TXOP Limit		0	0	0
0 / AC_BE				
AIFSN/AIFS	6	70 µs	110 µs	190 µs
CWmin	$(\text{aCWmin}+1)/2-1$	7	7	7
CWmax	aCWmin	15	15	15
TXOP Limit		0	0	0
2 / AC_VI				
AIFSN/AIFS	3	43 µs	71 µs	127 µs
CWmin	$(\text{aCWmin}+1)/4-1$	3	3	3
CWmax	$(\text{aCWmin}+1)/2-1$	7	7	7
TXOP Limit		0	0	0
3 / AC_VO				
AIFSN/AIFS	2	34 µs	58 µs	106 µs
CWmin	$(\text{aCWmin}+1)/4-1$	3	3	3
CWmax	$(\text{aCWmin}+1)/2-1$	7	7	7
TXOP Limit		0	0	0
DFS				
AIFSN/AIFS	2	34 µs	58 µs	106 µs
CWmin	aCWmin	15	15	15
CWmax	aCWmax	1 023	1 023	1 023

Table B.1: Default 802.11p/D4.02 EDCA parameters

ACI / AC		802.11a OFDM with channels			802.11b	802.11g CCK & OFDM	
		20 MHz	10 MHz	5 MHz	HR/DSSS	short preamble	long preamble
1 / AC_BK							
AIFSN/AIFS	7	79 µs	123 µs	211 µs	150 µs	73 µs	150 µs
CWmin	aCWmin	15	15	15	31	15 / 31	15 / 31
CWmax	aCWmax	1 023	1 023	1 023	1 023	1 023	1 023
TXOP Limit		0	0	0	0	0	0
0 / AC_BE							
AIFSN/AIFS	3	43 µs	71 µs	127 µs	70 µs	37 µs	70 µs
CWmin	aCWmin	15	15	15	31	15 / 31	15 / 31
CWmax	aCWmax	1 023	1 023	1 023	1 023	1 023	1 023
TXOP Limit		0	0	0	0	0	0
2 / AC_VI							
AIFSN/AIFS	2	34 µs	58 µs	106 µs	50 µs	28µs	50 µs
CWmin	$(\text{aCWmin}+1)/2-1$	7	7	7	15	7 / 15	7 / 15
CWmax	aCWmin	15	15	15	31	15 / 31	15 / 31
TXOP Limit		3 008 µs	3 008 µs	3 008 µs	6 016 µs	3 008 µs	3 008 µs
3 / AC_VO							
AIFSN/AIFS	2	34 µs	58 µs	106 µs	50 µs	28µs	50 µs
CWmin	$(\text{aCWmin}+1)/4-1$	3	3	3	7	3 / 7	3 / 7
CWmax	$(\text{aCWmin}+1)/2-1$	7	7	7	15	7 / 15	7 / 15
TXOP Limit		1 504 µs	1 504 µs	1 504 µs	3 264 µs	1 504 µs	1 504 µs
DFS							
AIFSN/AIFS	2	34 µs	58 µs	106 µs	50 µs	28µs	50 µs
CWmin	aCWmin	15	15	15	31	15 / 31	15 / 31
CWmax	aCWmax	1 023	1 023	1 023	1 023	1 023	1 023

Table B.2: Default 802.11e EDCA parameters for different PHY

Appendix C

ns-3 Crash Course

This section is a crash course introduction into ns-3's basic architecture and simulation setup. First basic building blocks as callbacks, the object system with attribute metadata and smart pointers are introduced. The second example scetches implementation of a protocol and demonstrates use of trace sources and event handlers. The first two parts focuses on the inner workings of ns-3. In the last section actual experiment source code used in the highway lanes scenario is explained step-by-step.

Much more information can be found in the ns-3 tutorial and reference manual distributed along with the source code. PDF and HTML copies are also available online on the ns-3 homepage [26].

The C++ constructs and idioms used in ns-3 are very advanced. To understand the inner works requires deep knowledge of class, templates, and the C++ idioms and design patterns implemented. However, to only *use* them, this deep understanding is not necessary. Nevertheless, even this crash course assumes a moderate level of experience with C++.

C.1 Callbacks

First unfamiliar C++ construct described here is the "ns-3 callback". It is related to the basic *function pointers* available in plain C and C++ and is sometimes also called *generalized functor* [1]. Callbacks are a very powerful design artifact and used extensively in ns-3 to decouple models and layers.

This section gives a step-by-step introduction into callbacks following the example `callback-example.cc` source code. (right-click the filename if your PDF viewer supports file attachments).

The following code listings are taken from `callback-example.cc`. To make step-by-step explanation more clear, the sequence of snippets is not in same order as in the file and some necessary but distracting lines are omitted from the listings.

```
 9 double AddInts(int a, int b)
10 {
11   return a + b;
12 }

35 int main()
36 {
37   Callback<double, int, int> cb1;
38
39   cb1 = MakeCallback(&AddInts);
40   std::cout << cb1(2, 5) << std::endl;
```

These two code examples introduce callbacks. First a simple adding function `AddInts()` is defined. For this function a matching callback is defined in `main()` by creating a `Callback<double, int, int>` variable. It is easiest to view this object as a traditional function pointer variable `double (*cb1)(int, int)`, which contains a typed memory address to some function's code location. Note the matching signature: return value a `double` and two `int` arguments.

This callback variable can be assigned to `AddInts()` by using the magic `MakeCallback()` function, which creates a callback to `AddInts()` as done in line 39. Callback variables can be invoked just like normal function pointers, as seen in line 40.

```
14 double AddDoubles(double a, double b)
15 {
16   return a + b;
17 }

35 int main()
36 {
37   Callback<double, int, int> cb1;
38
49   cb1 = MakeCallback(&AddDoubles); // compile error
50   cb1(1, 2, 3); // another compile error
51
52   Callback<double, double, double> cb2 = MakeCallback(&AddDoubles);
```

Assignments to callbacks are type-checked at compile time. Consider a different function, named `AddDoubles()`, taking `double` arguments. Assignment to the old callback, which requires `int` arguments, results in a compile error. Similarly invoking the callback with three arguments also yields a compile error. The error messages from the compiler are rather long and cryptic, often the error is easier to spot by looking at the code. Line 45 shows a correct callback assignment for `AddDoubles()`.

In the previous two assignment scenarios, callbacks did not show any advantage over ordinary function pointers. To show the real use of callbacks, one must regard functions within classes: the next snippet shows a simple class with one static and one member function.

```
19 class Point
20 {
21 public:
22   static double Norm(int a, int b)
23   {
24     return sqrt( a*a + b*b );
25   }
26
27   int x, y;
28
29   double Distance(int a, int b)
30   {
31     return sqrt( (x-a)*(x-a) + (y-b)*(y-b) );
32   }
33 };

35 int main()
36 {
37   Callback<double, int, int> cb1;
38
39   cb1 = MakeCallback(&AddInts);
40   std::cout << cb1(2, 5) << std::endl;
41
42   cb1 = MakeCallback(&Point::Norm);
43   std::cout << cb1(2, 5) << std::endl;
44
45   Point p1 = { 1, 1 };
46   cb1 = MakeCallback(&Point::Distance, &p1);
47   std::cout << cb1(2, 5) << std::endl;
```

Both functions of `Point` can be used with the callback variable previously defined. In all three assignments of `cb1`, a function with equal arguments and return value is referenced, however, these are of three different kinds.

In line 39, the callback is assigned a plain function. `MakeCallback()` in line 42 also creates a plain function callback, but this time the function is static and thus has no object context.

The real magic of callbacks is shown in lines 45–46: a callback can also be constructed for a member function with object context. When invoking the callback in 47, the function `Distance()` is called for the object `p1`. Here the callback object actually contains *two* pointers: one to the object `p1` and one to the member function `Point::Distance()`.

The last operation cannot be done with plain function pointers. Note that all three callback invocation are identical, regardless of the currently assigned function.

In ns-3, this C++ idiom is used to decouple models and protocols. Packets crossing between layers usually are passed through callbacks. By changing the callbacks, an additional layer can easily be inserted between two layers. This decoupling is exemplified the following simpler scenario:

```
55 class Alpha
56 {
57 public:
58   void ReceiveInput(double x);
59 };
60
61 class UsualLayer
62 {
63 public:
64   Alpha* m_alpha;
65
66   void DoWork()
67   {
68     double work = 5;
69     m_alpha->ReceiveInput(work);
70   }
71 };
72
73 class EnhancedLayer
74 {
75 public:
76   Callback<void, double> m_receiveWork;
77
78   void DoWork()
79   {
80     double work = 5;
81     m_receiveWork(work);
82   }
83 };
```

In the above example, `Alpha` is some model or layer receiving input. The usual method to call an instance of `Alpha` is shown in `UsualLayer`: it contains a pointer link to an `Alpha` object. This introduces a tight coupling between `UsualLayer` and `Alpha`; insertion of an intermediate processing layer between the two is difficult.

A better solution using callbacks is shown in `EnhancedLayer`. Note that `EnhancedLayer` does not mention `Alpha` anymore. Instead, it specifies a callback function, which will receive work done by the layer. The connection between an `Alpha` object and `EnhancedLayer` is established by plugging `ReceiveInput()` into the callback. However, any other function or protocol layer matching the signature could also be used. The layer class can now be detached from lower layers and stacked differently.

C.2 Objects, Ptrs, Attributes and TraceSources

After the short precursor into callbacks, this crash course continues on ns-3's object and attribute system. These will be explained by illustrating another C++ example program, defining a new class `MyProtocol`. This protocol will do "something interesting" with a packet according to two parameters and indicate success or failure through two different callbacks. It also demonstrates use of the simulator's event loop by delaying work for some time. To allow inspection of the protocol, two trace sources are defined.

The example's complete source code is embedded in this PDF and can be extracted using a compatible PDF viewer here: `myprotocol-example.cc`.

Before starting directly with the example code, reflect the broad problem of designing a simulator: any network simulator will contain a myriad of network objects building up nodes, layers, device models and more. All these objects have parameters controlling their behavior. Simulator users will want to list and customize these parameters for their experiments. Modification of these parameters must be flexible and may change with each simulation run. These parameters are described in ns-3 using object metadata encoded in a `TypeId` object. All objects exporting this metadata are grouped into a hierarchy below the superclass `Object`. This allows programmatic access to all attributes influencing the simulation. Attributes can thus be modified and listed at run time using the command line or even with a graphical user interface.

Simulation objects can also export so called *TraceSources* in their metadata, which are called from within the object's code to indicate interesting events. Some examples of TraceSources are packet reception at the MAC layer, TCP congestion window changes or when a wireless station associates with an AP.

Aside all this metadata functionality, simulated objects also require memory management in C++. An ownership relation of layers quickly becomes too complex due to the myriad of interrelated objects in the simulation. To solve this C++-inherit problem, ns-3 defines the `Ptr<>` template. Almost all simulation objects in ns-3 are derived from `Object` and use `Ptr<>` as a smart pointer. Due to the metadata system, ns-3 requires most objects to be created with `CreateObject<>` instead of `new`.

Correct object creation and pointer handling is illustrated in the following example. Getting used to this syntax may need some time, but it is mandatory in ns-3.

```
1 Ptr<Node> mynode = CreateObject<Node>();
2 Ptr<Node> node1;
3 node1 = mynode;
4 std::cout << node1->GetId();
5 Ptr<Packet> mypacket = Create<Packet>(100);
```

The snippet above shows a simple case of how to use `Ptr<Node>` instead of `Ptr*` and `CreateObject<Node>()` instead of `new Node()`. A notable exception to the use of `CreateObject<>` is shown in line 5: for performance considerations, `Packet` is not derived from `Object`. Instead, `Packet`s are instantiated using a more basic `Create<>` method.

The source code of the `MyProtocol` example begins with the following lines:

```
1 // -*- mode: c++; c-file-style: "gnu"; indent-tabs-mode: nil; -*-
2
3 #include <ns3/core-module.h>
4 #include <ns3/simulator-module.h>
5 #include <ns3/common-module.h>
6
7 using namespace ns3;
```

Most C++ source code in ns-3 begins with the line 1 shown above. This line is an emacs directive, setting the editor options to ns-3 indention standards.

Lines 3–5 include all headers from the components *core*, *simulator* and *common*, which are described in section 4.1.

Line 7 makes the whole ns3 namespace available without prefixing each class with `ns3::`. All classes exported by ns-3 are enclosed in this namespace.

The example class `MyProtocol` is derived from `Object` (actually from `ns3::Object`) and receives many features from it. The following listing contains the whole class declaration and is explained in the following paragraphs.

```
9  class MyProtocol : public Object
10 {
11 public:
12   // returns a TypeId describing the class's Attributes and TraceSources.
13   static TypeId GetTypeId();
14
```

```
15    // do something interesting with the packet
16    void ReceivePacket(Ptr<Packet> packet);
17
18    // work finished after some time
19    void FinishWork(Ptr<Packet> packet, Time workStarted);
20
21    // signature of a callback for successful and failed work
22    typedef Callback< void, Ptr<const Packet> > WorkSuccess;
23    typedef Callback< void, Ptr<const Packet>, int > WorkFailed;
24
25    // set an external function to be called on successful or failed work
26    void SetWorkSuccessCallback(WorkSuccess callback);
27    void SetWorkFailedCallback(WorkFailed callback);
28
29  private:
30    // actual callback variables, set using public functions
31    WorkSuccess m_workSuccessCallback;
32    WorkFailed m_workFailedCallback;
33
34    // parameters of the interesting work on the packet
35    int m_param1;
36    double m_param2;
37    Time m_paramTime;
38
39    // trace callback for start of work with initial packet
40    TracedCallback< Time, Ptr<const Packet> > m_workStartTrace;
41
42    // trace callback for time and duration of work done with processed packet
43    TracedCallback< Time, Time, Ptr<const Packet> > m_workEndTrace;
44  };
45
46  NS_OBJECT_ENSURE_REGISTERED(MyProtocol);
```

Most of the magic of the object and attribute system is found in the `GetTypeId()` of `MyProtocol`. This function is listed in the next snippet and returns a description of all attributes exported by the class. The metadata is contained in the returned `TypeId` object.

The example class has two main functions `ReceivePacket()` and `FinishWork()`, which are supposed to do something interesting with a packet. Note the arguments' type `Ptr<Packet>` in lines 16 and 19. The methods receive packets as smart pointers. Generally a packet is passed around in form of such a pointer; real copies must explicitly be made.

Two callbacks are `typedef`ed in `MyProtocol`, lines 22 and 23: one to indicate successful work and one for failure. These lines only `typedef` the callbacks; in lines 31 and 32 the actual memory space for them is prepared as two variables of the class. To set these private variables, two public functions `SetWorkSuccessCallback()` and `SetWorkFailedCallback()` are defined.

For illustration purposes, two parameter variables `m_param1` and `m_param2` are defined in lines 35 and 36. The third parameter `m_paramTime` is actually used to delay invocation of `FinishWork()`.

Finally, two `TracedCallback<>`s are defined by `MyProtocol` in lines 40 and 43: one indicating that work started on a packet and one for completed work. `TracedCallback<>` is based on `Callback<>` and extends it to process a *list* of attached callbacks, which are all invoked sequentially to indicate an event.

Line 46 is a macro which ensures that the metadata of `MyProtocol` is globally available.

The code listing below defines the metadata description function `GetTypeId()`.

```
48  TypeId
49  MyProtocol::GetTypeId()
50  {
51    static TypeId tid = TypeId("ns3::MyProtocol")
52      .SetParent<Object>()
53      .AddConstructor<MyProtocol>()
54      .AddAttribute("Param1",
```

```
55                    "Important parameter of work done by this protocol.",
56                    IntegerValue(502),
57                    MakeIntegerAccessor(&MyProtocol::m_param1),
58                    MakeIntegerChecker<int>())
59      .AddAttribute("Param2",
60                    "Another important parameter of work done by this protocol.",
61                    DoubleValue(999.0),
62                    MakeDoubleAccessor(&MyProtocol::m_param2),
63                    MakeDoubleChecker<double>(100, 10000))
64      .AddAttribute("ParamTime",
65                    "Actual parameter specifying work time.",
66                    TimeValue( MilliSeconds(10) ),
67                    MakeTimeAccessor(&MyProtocol::m_paramTime),
68                    MakeTimeChecker())
69      .AddTraceSource("WorkStart",
70                      "Time packet work started.",
71                      MakeTraceSourceAccessor(&MyProtocol::m_workStartTrace))
72      .AddTraceSource("WorkEnd",
73                      "Triggered on work end.",
74                      MakeTraceSourceAccessor(&MyProtocol::m_workEndTrace))
75      ;
76    return tid;
77  }
```

The function returns an object of type `TypeId`, which contains a lot of metadata on the object. The syntax for creating a `TypeId` is rather non-standard; for all usual purposes it is most practical to just copy it from an exist file and adapt it.

Lines 51–53 define the class's metadata name, parent class and default constructor in a straight-forward manner.

Lines 54–58 and 59–63 define two public attributes of the class: "Param1" and "Param2". These attributes must be viewed in a broader scope: consider them as parameters of a complex protocol implementations. Simulation users will want to change them for different runs of an experiment. A third attribute "ParamTime" is added in lines 64–68, and will control work time required by the protocol.

The first three lines of each exported attributes contains the name, a short description and the default value. The default value is not just an integral type, but uses an `AttributeValue` wrapping. This wrapping is required to enable flexible type conversions and allows special types in the attribute metadata. In the example, the default values are encapsulated in an `IntegerValue`, a `DoubleValue` and a `TimeValue` object. Each attribute also contains an accessor and a checker object, they are used by the attribute system to access the variable's memory. Discussion of accessors and checkers are outside the scope of this example.

In summary: the object exports three attributes with a lot of extra information. The ns-3 attribute system allows comfortable access to these variables, which will be described later on. In whole the attributes *replace* traditional `GetXYZ()` and `SetXYZ()` function pairs with programmatically accessible attribute definitions.

TraceSources are also exportable via the metadata system. In the example, the two sources `WorkStart` and `WorkEnd` are added to the constructed `TypeId` in lines 69–71 and 72–74. Once registered, they can be connected to different *TraceSinks* defined by the simulation user.

After the previous highly obscure piece of code, follow two refreshingly simple assignment functions:

```
79  void
80  MyProtocol::SetWorkSuccessCallback(WorkSuccess callback)
81  {
82    m_workSuccessCallback = callback;
83  }
84  void
85  MyProtocol::SetWorkFailedCallback(WorkFailed callback)
86  {
87    m_workFailedCallback = callback;
88  }
```

These two public functions can be used to set the private callback objects. Review the class definition to see that `WorkSuccess` and `WorkFailed` are `typedef`ed `Callback<>` classes.

In the following snippet, `MyProtocol`'s two processing functions are defined.

```
90 void
91 MyProtocol::ReceivePacket(Ptr<Packet> packet)
92 {
93   std::cerr << "MyProtocol::ReceivePacket() with "
94             << "Param1=" << m_param1 << ", Param2=" << m_param2 << "\n";
95
96   m_workStartTrace(Simulator::Now(), packet);
97
98   Simulator::Schedule(Simulator::Now() + m_paramTime,
99                       &MyProtocol::FinishWork, this,
100                      packet, Simulator::Now());
101 }
102
103 void
104 MyProtocol::FinishWork(Ptr<Packet> packet, Time workStarted)
105 {
106   // do something interesting with packet.
107   bool workOk = (m_param1 % 2 == 1);
108
109   if (workOk)
110     {
111       m_workSuccessCallback(packet);
112     }
113   else
114     {
115       m_workFailedCallback(packet, 10);
116     }
117
118   m_workEndTrace(Simulator::Now(), Simulator::Now() - workStarted,
119                  packet);
120 }
```

The example protocol will "start" processing a packet in the `ReceivePacket()` function. This imaginary protocol requires some processing time, thus `ReceivePacket()` only indicates start of work by calling the appropriate trace callbacks in line 96. To simulate processing time, an event must be schedules at the time when work ends. The event handler used is `FinishWork()` and must be scheduled to be executed after `m_paramTime` elapsed.

This event is scheduled in lines 98–100 by invoking the `Simulator::Schedule()` function. `Simulator` is a singleton object holding the global simulation coordinator. The arguments to this call are `Schedule(`*`time`*`,` *`function-pointer`*`,` *`object-address`*`,` *`arguments...`*`)`. Note that any function can be called; the event handler need not be derived from a class `Event` as in ns-2. Furthermore any arguments can be passed to the handler, in the example the current time and processed packet are pushed to the handler when invoked.

Thus after `m_paramTime`, counted from the current simulator time, the `FinishWork()` function is called. This function then does some interesting packet processing algorithm and determines whether it was successful. Accordingly either `m_workSuccessCallback` or `m_workFailedCallback` is called.

To allow an experimenter to gather information on the protocol's behavior, the `WorkEnd` trace callback is called with the possibly modified packet as an argument.

The protocol definition is now complete and the next listings will describe how the new protocol can be used.

In the next listing four callback functions are defined: two for the success/failure callbacks and two for the start/end trace sources.

```
122 /* main program */
123
124 void
125 Proto1WorkSuccessCallback(Ptr<const Packet> packet)
126 {
127   std::cerr << "proto1's work succeeded on packet.\n";
```

```
128 }
129 void
130 Proto1WorkFailedCallback(Ptr<const Packet> packet, int reason)
131 {
132   std::cerr << "proto1's work failed on packet, reason: " << reason << ".\n";
133 }
134
135 void
136 Proto1WorkStartTrace(std::string context,
137                      Time start, Ptr<const Packet> packet)
138 {
139   std::cerr << Simulator::Now() << " " << context
140             << " time=" << start << ".\n";
141 }
142 void
143 Proto1WorkEndTrace(std::string context,
144                    Time start, Time duration, Ptr<const Packet> packet)
145 {
146   std::cerr << Simulator::Now() << " " << context
147             << " time=" << start
148             << " duration=" << duration << ".\n";
149 }
```

The signatures of these functions are fixed according to the definitions in the `MyProtocol` class. Only new parameter is `context` to the *TraceSink* functions. `context` is used in large simulations to indicate the object issuing the trace event: it contains node id, protocol and application names, which otherwise would not be available to the TraceSink function.

In the following `main()`, one protocol object is created and its callbacks are connected to the printing functions defined above.

```
151 int main(int argc, char *argv[])
152 {
153   Config::SetDefault("ns3::MyProtocol::Param1", IntegerValue(503));
154
155   CommandLine cmd;
156   cmd.Parse(argc, argv);
157
158   Ptr<MyProtocol> proto1
159     = CreateObject<MyProtocol>("Param2", DoubleValue(1001.0),
160                                "ParamTime", TimeValue(Seconds(0.240)));
161
162   proto1->SetAttribute("Param2", StringValue("1002.5"));
163
164   proto1->SetWorkSuccessCallback(MakeCallback(&Proto1WorkSuccessCallback));
165   proto1->SetWorkFailedCallback(MakeCallback(&Proto1WorkFailedCallback));
166
167   proto1->TraceConnect("WorkStart", "main::proto1",
168                        MakeCallback(&Proto1WorkStartTrace));
169   proto1->TraceConnect("WorkEnd", "main::proto1",
170                        MakeCallback(&Proto1WorkEndTrace));
171
172   Simulator::Schedule(Seconds(1),
173                       &MyProtocol::ReceivePacket, proto1,
174                       Create<Packet>(100));
175
176   Simulator::Run();
177 }
```

In lines 158–159, the protocol object is created using the `CreateObject<>` template. During creation, all attribute values defined in `TypeId()` are initialized with their default values. There are many ways to override these default values, of which the example shows the following four:

First method is to use the global `Config` utility as shown in line 153. By changing the default value here, all objects created afterwards will have the new value.

Second method shown in the example is to pass parameters to the `CreateObject<>` call. The parameters are always pairs of string key and `AttributeValue`.

Attribute values can also be changed later by calling `SetAttribute()` on an object, as exemplified in line 162.

A fourth way to override attributes is enabled by parsing the command line in lines 155–156. ns-3 contains a complex command line processor: for example by running the simulation with "`myprotocol-example --ns3::MyProtocol::Param1=300`" the default value of "Param1" can be overridden with 300.

After setting all parameters of the `MyProtocol` object, lines 164–165 connect the two callbacks and lines 167–170 connect the other two trace callbacks. Main difference between both callbacks is that for the event traces multiple callbacks may be registered and for the plain callback only one is allowed.

Not shown in the example is how to connect event traces by using `Config::Connect()`. This is not possible in the example because no network `Node` object is created. In a real simulation containing `Node`s, each node contains a list of attached protocols and these are accessible via `Config`. In such a full-blown simulation, the TraceSource could also be connected with

```
1 Config::Connect("/NodeList/*/$ns3::MyProtocol/WorkStart",
2                 MakeCallback(&Proto1WorkStartTrace));
```

To make the protocol instance `proto1` do some work, the `main()` function manually schedules a packet transmission in lines 172–174. Note the direct reference to `ReceivePacket()` and construction of the packet using `Create<>`.

In line 176 the simulation's main event processing loop is run. The loop terminates when no more scheduled events require processing.

The output of the example program, when run with no parameters, is the following:

```
1 MyProtocol::ReceivePacket() with Param1=503, Param2=1002.5
2 1000000000ns main::proto1 time=1000000000ns.
3 proto1's work succeeded on packet.
4 2240000000ns main::proto1 time=2240000000ns duration=1240000000ns.
```

Note that `Config::SetDefault()` overrides the "Param1" to 503, which allows the protocols work to succeed. Success is signaled via the callback. In the tracing output lines, progress of simulated time can be observed: protocol processing time was 124 ms, as set with "ParamTime" in the `CreateObject<>` call.

This wraps up the example protocol implementation. Not all features shown in this example are required to build simulations. Many aspects are more part of inner workings of ns-3 and their understanding is only required to read model and protocol implementation code. The next example shows how to actually build a simulation.

C.3 Highway Lanes Scenario Code

In this section the crash course continues with a wireless network simulation experiment built using ns-3. The example upon which this short introduction is given, is a simplified version of the highway lanes experiments used to speed test ns-2 and ns-3 in section 8.1. In particular the whole time measurement code was removed to make the example more concise, also Nakagami fast fading was added to demonstrate this new feature.

The described simulation source code is embedded in this PDF: `highway-lanes.cc`. It contains enhancements added to ns-3 in this thesis and thus cannot be directly run with ns-3.4, however, future major versions will include the contributed features. The principle simulation composition method can be applied to other scenarios requiring different models and applications.

The source code file starts, as all previous files, with an emacs directive setting automatic indentation style. After a short description of the simulation scenario, ns-3 headers files are included. Note that, different from previous code examples, here `...-module.h` headers are used. These are automatically generated and include all public header files of the specific component (see section 4.1). Furthermore, three header from the STL are included for output and calculations.

```
1  // -*- mode: c++; c-file-style: "gnu"; indent-tabs-mode: nil; -*-
2
3  /*
4   * Test case: 6*n nodes on a six lane highway
5   *
6   * 6*n nodes are put on a highway with 6 lanes. Each lane is 5 meters apart
7   * from neighboring lanes. Cars are spaced at 90 meters on each lane (15 meters
8   * between two nodes along the x axis) yielding a total of 66.6 nodes per kilometer.
9   */
10
11 #include "ns3/core-module.h"
12 #include "ns3/simulator-module.h"
13 #include "ns3/node-module.h"
14 #include "ns3/wifi-module.h"
15 #include "ns3/helper-module.h"
16 #include "ns3/traffic-application.h"
17
18 #include <iostream>
19 #include <iomanip>
20 #include <numeric>
```

Following the includes preamble, two directive are needed prior to beginning experiment code. First the complete `ns3` namespace is imported instead of reiterating `ns3::` in front of each used class. Second the ns-3 logging facilities are informed that log messages from this file are to be categorized as "`Main`". The example will use ns-3 logging macros, which can be activated or suppressed at run-time.

```
22 using namespace ns3;
23
24 NS_LOG_COMPONENT_DEFINE("Main");
```

In the following snippet, the experiment class is started. The whole experiment setup and running code is contained in one class. This encapsulation allows intuitive scoping of result variables and provisions for running multiple replications of the same experiment with different parameters. To record results the class contains the three variables `m_appTxPackets`, `m_appRxPackets` and `m_phyRxErrors`, which will count transmitted, received and dropped packets across all simulated nodes.

```
26 class Experiment
27 {
28 public:
29
30   static const double m_simulatedTime = 60.0;
31
32   unsigned int m_appTxPackets;
33   unsigned int m_appRxPackets;
34   unsigned int m_phyRxErrors;
```

One iteration of the experiment is executed in the following `Run()` function. The complete simulation setup is contained in this function. All simulated objects are created and destroyed in each replication; they are not reused.

```
36   void
37   Run(unsigned int numNodes)
38   {
39     // Create nodes and store them in the container.
40
41     NodeContainer nodes;
42     nodes.Create(numNodes);
43
44     // Add packet socket handlers.
45
46     PacketSocketHelper packetSocket;
47     packetSocket.Install(nodes);
```

The highway lanes experiment has one parameter: the number of nodes placed on the lanes. These nodes are created in line 42 using the `NodeContainer` helper object. This container can be simply viewed as a vector of `Node` objects. It is the basis for a series of helper classes contained in ns-3 to make common tasks during simulation set up easy. This helper pattern is exemplified by the following lines 46–47, in which the packet socket "stack" is added to each node. Packet sockets are simplified versions of UDP or TCP sockets, except that each packet is transferred with no modifications. No header or trailer, sequence numbering or congestion control algorithm is added.

Instead of using the helper classes `NodeContainer` and `PacketSocketHelper`, the node set up could be written out into a `for` loop containing `CreateObject<Node>()` and corresponding construction of packet sockets. These loops are hidden for convenience in the `Create()` and `Install()` functions. This becomes very useful when creating a wireless channel and adding wireless LAN devices to the nodes:

```
49      // Install wifi devices on the nodes.
50
51      Ns2ExtWifiChannelHelper wifiChannel;
52      wifiChannel.SetPropagationDelay("ns3::ConstantSpeedPropagationDelayModel");
53      wifiChannel.AddPropagationLoss("ns3::ThreeLogDistancePropagationLossModel");
54      wifiChannel.AddPropagationLoss("ns3::NakagamiPropagationLossModel",
55                                     "m0", DoubleValue(1.5),
56                                     "m1", DoubleValue(1.0),
57                                     "m2", DoubleValue(1.0));
58
59      Ns2ExtWifiPhyHelper wifiPhy = Ns2ExtWifiPhyHelper::Default();
60      wifiPhy.SetChannel(wifiChannel.Create());
61      wifiPhy.Set("UseConstantNoiseFloor", BooleanValue(true));
62      wifiPhy.Set("ConstantNoiseFloor", DoubleValue(-99.0));
63      wifiPhy.Set("PreambleCapture", BooleanValue(true));
64      wifiPhy.Set("DataCapture", BooleanValue(true));
65
66      WifiHelper wifi = WifiHelper::Default();
67      wifi.SetMac("ns3::AdhocWifiMac");
68      wifi.SetRemoteStationManager("ns3::ConstantRateWifiManager",
69                                   "DataMode", StringValue("wifia-6mbs"),
70                                   "NonUnicastMode", StringValue("wifia-6mbs"));
71
72      wifi.Install(wifiPhy, nodes);
```

The 802.11 model is highly configurable and thus this part of the simulation set up has the most parameters. For the experiment the `Ns2ExtWifiChannel` and `Ns2ExtWifiPhy` classes created for this thesis are used. Again their actual creation is done by helper wrappers instead of explicitly. Nevertheless, detailed configuration of the models can be done through the helpers, which follow the *factory* design pattern and create object instances with modified parameters.

The channel object is created with `Ns2ExtWifiChannelHelper` and configured to use three-log-distance and Nakagami propagation loss. The default speed-of-light propagation delay model is also applied. For demonstration, the m parameters of the Nakagami model are modified before instantiation.

Wireless network devices are created with `Ns2ExtWifiPhyHelper`. These are attached to the previously configured channel and some model parameters are modified. Frame capture is actually activated by default, but setting these options to true doesn't hurt.

`WifiHelper` is a helper class used to create the remaining MAC layer objects of the 802.11 model, which are located on top of `WifiPhy`. Primary option of these higher layers is the type of wireless station created. In the example a non-QoS ad-hoc station is created, `QosAdhocWifiMac` could be used to add EDCA QoS queues. Each wireless station also contains a `WifiRemoteStationManager`, which is used for rate control and retransmission counters. The rate control algorithm is set in the example to a constant "algorithm" fixing the rate to 6 Mb/s.

After the complex configuration process, wireless network devices (`WifiNetDevices`) are installed on all nodes with just one line of code using `Install()`.

In the next step, "mobility" of nodes is defined. For the highway lanes scenario this means positioning nodes on six parallel lines.

```
74      // Position nodes on to highway lanes.
75
76      Ptr<ListPositionAllocator> positionAlloc
77        = CreateObject<ListPositionAllocator>();
78      for (unsigned int i = 0; i < numNodes; ++i)
79        {
80          positionAlloc->Add(Vector(i * 15, (i % 6) * 5, 0.0));
81        }
82
83      MobilityHelper mobility;
84      mobility.SetPositionAllocator(positionAlloc);
85      mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");
86      mobility.Install(nodes);
```

Again the actual creation of the mobility objects is left to helpers. The mobility class is verbosely named `ConstantPositionMobilityModel`. Initial, and in this case fixed, node positions are taken from a list, which is filled with positions calculated using the formula in line 80. More complex and actually mobile simulations can be created with other mobility models.

Next the traffic generator applications are installed. In this scenario, the `TrafficApplication` packet generator implemented for this thesis is set up to create isochronous beacon-like traffic.

```
88      // Use broadcast packet address for applications.
89
90      PacketSocketAddress socketBroadcast;
91      socketBroadcast.SetAllDevices();
92      socketBroadcast.SetPhysicalAddress(Mac48Address::GetBroadcast());
93      socketBroadcast.SetProtocol(1);
94
95      // Install TrafficApplication on each node.
96
97      Ptr<SimpleTrafficPacketFactory> packetFactory
98        = CreateObject<SimpleTrafficPacketFactory>("Size", UintegerValue(400));
99
100     TrafficHelper trafficApp("ns3::PacketSocketFactory", socketBroadcast);
101     trafficApp.SetAttribute("PacketFactory", PointerValue(packetFactory) );
102     trafficApp.SetAttribute("OnTime",
103                             RandomVariableValue( ConstantVariable(m_simulatedTime) ));
104     trafficApp.SetAttribute("OffTime",
105                             RandomVariableValue( UniformVariable(0.0, 0.1) ));
106     trafficApp.SetAttribute("Interval",
107                             RandomVariableValue( ConstantVariable(0.1) ));
108
109     ApplicationContainer app = trafficApp.Install(nodes);
110     app.Start( Seconds(0.0) );
111     app.Stop( Seconds(m_simulatedTime) );
```

Again a helper class `TrafficHelper` is used. Each traffic generator must be configured with a destination address, which in this case is the broadcast address. Because the packet socket stack is used, a `PacketSocketAddress` must be used (instead of say an `Ipv4Address`). Destination address of the socket created in lines 91–94 is the MAC broadcast address.

Packet creation in `TrafficApplication` follows the *abstract factory* pattern: the traffic generator must be configured with an instance of `TrafficPacketFactory`. In this example the predefined `SimpleTrafficPacketFactory` is used to created 400 byte sized zero-filled packets.

Traffic stream properties are set in lines 103–108: packets are generated at 10 Hz by setting "`Interval`" to a constant random variable. For Poisson traffic, an `ExponentialVariable` can be used. Start of the isochronous packet sequences is randomized for each node by setting the "`OffTime`" parameter to a uniform variable. The traffic generator starts in an *off* state of random duration, then it switches to *on* and starts transmitting packets for `m_simulatedTime`.

As before with other helpers, the configured traffic application is installed on all nodes in line 110. However, the installed application objects must be set up to start and stop at specific simulated time points. This

starting and stopping is common to all applications and is not related to the *on*/*off* state alteration in `TrafficApplication`. As `NodeContainer` groups `Nodes`, `ApplicationContainer` contains many `Applications`, which are all started at simulation time start and stopped after the targeted time.

To collect information for evaluation of the experiment, callbacks must be connected to trace sources of interest.

```
113     // Add Trace callbacks to gather statistics.
114
115     Config::Connect("/NodeList/*/ApplicationList/*/$ns3::TrafficApplication/Tx",
116                     MakeCallback(&Experiment::AppTxTrace, this));
117     Config::Connect("/NodeList/*/ApplicationList/*/$ns3::TrafficApplication/Rx",
118                     MakeCallback(&Experiment::AppRxTrace, this));
119
120     Config::Connect("/NodeList/*/DeviceList/*/Phy/State/RxError",
121                     MakeCallback(&Experiment::PhyRxErrorTrace, this));
```

Three callbacks are attached to trace sources available in the scenario. The actual trace callback functions are defined in one of the following code parts. Syntax of the trace source path specifier is somewhat obscure and it is best to reuse existing examples.

The two traces exported by `TrafficApplication` are hooked: transmission and reception of a packet. Note that the traffic generator also automatically accepts all received packets, otherwise they would be queued indefinitely. For collision statistics, the WifiPhy trace source "`RxError`" is hooked.

The simulation set up is now complete and the event loop can start.

```
123     // Zero counters and run simulation.
124
125     m_appTxPackets = 0;
126     m_appRxPackets = 0;
127     m_phyRxErrors = 0;
128
129     Simulator::Run();
130     Simulator::Destroy();
131   }
```

Prior to starting the event loop, the statistics counters are zeroed. With `Simulator::Run()` the event loop is entered. It runs until no more events are scheduled and then terminates the experiment. All created simulation objects are destroyed and allocated memory is freed with `Simulator::Destroy()`. This concludes the `Experiment::Run()`, statistical processing is left to outside code.

These statistics are base on the packet counters incremented by following three trace callbacks.

```
133   void
134   AppTxTrace(std::string context, Ptr<const Packet> p)
135   {
136     NS_LOG_DEBUG(context << " TX size=" << p->GetSize());
137     ++m_appTxPackets;
138   }
139
140   void
141   AppRxTrace(std::string context, Ptr<const Packet> p, const Address& from)
142   {
143     NS_LOG_DEBUG(context << " RX from=" << from << " size=" << p->GetSize());
144     ++m_appRxPackets;
145   }
146
147   void
148   PhyRxErrorTrace(std::string context, Ptr<const Packet> p,
149                   Ptr<const WifiPhyTag> phytag, WifiPhy::RxErrorReason reason)
150   {
151     NS_LOG_DEBUG(context << " PHYRXERROR"
152                  << " reason=" << WifiPhy::RxErrorReasonToString(reason)
153                  << " phytag={" << *phytag << "} p={" << *p << "}");
154     ++m_phyRxErrors;
```

```
155   }
156 };
```

These callbacks were connected to their corresponding trace sources during simulation set up. Their parameter list must match the required callback signature exactly, otherwise a verbose run-time error is thrown and the experiment is not run.

The callbacks in this example only increment the corresponding packet counter and optionally outputs a debug message. These logging macros can be activated in ns-3 debug mode by setting the environment variable `NS_LOG=Main`.

The collected packet counts will be processed using the following three statistical functions.

```
158 template <typename Container>
159 double meanValue(const Container& c)
160 {
161   return std::accumulate(c.begin(), c.end(), 0.0) / c.size();
162 }
163
164 template <typename Container>
165 double standardDeviation(const Container& c)
166 {
167   double squareSum = 0.0;
168   double sum = 0.0;
169
170   for (typename Container::const_iterator ei = c.begin();
171        ei != c.end(); ++ei)
172     {
173       squareSum += (double)(*ei) * (double)(*ei);
174       sum += *ei;
175     }
176
177   double mean = sum / c.size();
178   return sqrt( (squareSum / c.size()) - (mean * mean) );
179 }
180
181 template <typename Container>
182 double errorMargin(const Container& c)
183 {
184   return 2.576 * standardDeviation(c) / sqrt(c.size());
185 }
```

Mean value and 99% error margin will be calculated over a number of independent replications.

In the following `main()` function, the `Experiment` class is instantiated and results are collected.

```
187 int main(int argc, char *argv[])
188 {
189   CommandLine cmd;
190   int replications = 1;
191   unsigned int fixedNumNodes = 0;
192   cmd.AddValue("Replications", "Perform independent replications.", replications);
193   cmd.AddValue("NumNodes", "Run for a fixed number of node.", fixedNumNodes);
194   cmd.Parse(argc, argv);
```

To make the experiment program more flexible, two special parameters are added to the already highly versatile ns-3 command line options. One is the number of independent replication performed, and the other can be used to test only a fixed number of nodes, which is useful to run multiple experiments processes in parallel.

The final code snippet contains the rest of the `main()` function.

```
196   for (unsigned int numNodes = 6; numNodes <= 180; numNodes += 6)
197     {
198       if (fixedNumNodes != 0 && numNodes != fixedNumNodes) continue;
199
200       std::vector<unsigned int> appTxPackets;
```

```
201       std::vector<unsigned int> appRxPackets;
202       std::vector<unsigned int> phyRxErrors;
203
204       for(int rep = 0; rep < replications; ++rep)
205         {
206           SeedManager::SetRun(rep);
207
208           Experiment experiment;
209           experiment.Run(numNodes);
210
211           appTxPackets.push_back( experiment.m_appTxPackets );
212           appRxPackets.push_back( experiment.m_appRxPackets );
213           phyRxErrors.push_back( experiment.m_phyRxErrors );
214         }
215
216       std::cout << std::fixed
217                 << numNodes
218                 << " " << meanValue(appTxPackets) << " " << errorMargin(appTxPackets)
219                 << " " << meanValue(appRxPackets) << " " << errorMargin(appRxPackets)
220                 << " " << meanValue(phyRxErrors) << " " << errorMargin(phyRxErrors)
221                 << std::endl;
222     }
223
224   return 0;
225 }
```

The default experiment configuration iterates over all node counts from 6 to 180 with 6 nodes added in each step. Line 198 was the easiest way to set a fixed number of nodes without refactoring the whole experiment code.

Experiment replications are run in the `for` loop in lines 204–214. Independently seeded random streams are created by using `SeedManager::SetRun()` function. In each iteration of the `for` loop a new experiment object is created, run and results are saved in the three vectors. Average and error margins of these results are then outputted for further processing, e.g. with gnuplot.

In figure C.1 the results of this example experiment are plotted. Note that the number of sent packets is indicated by the left y axis, whereas the number of received packets and reception errors is scaled by the right y axis. This representation was chosen, because the somewhat oversimplified total packet count metric reaches high numbers for lots of broadcasting nodes.

As expected for this experiment, the number of sent packets is proportional to the number of broadcasting nodes. Successfully received packets also increases proportionally, which indicates that the medium is not saturated. Error margins on the received packets are rather large due to probabilistic Nakagami fast fading. Reception errors are indicated by ns-3 for a number of different reasons, which include low signal strength, interrupting preamble or data capture or concurrent transmission. These reasons are not broken down into different figures, but are all contained in the increasing reception error plot line.

This concludes the crash course into basic ns-3 simulation architecture and scenario structure. Simulation source code using other models like Ethernet can be found in the `examples` directory of the ns-3 distribution. This directory also features some scenarios built using the Python bindings. More tutorials, talks and examples are collected on the official ns-3 website [26].

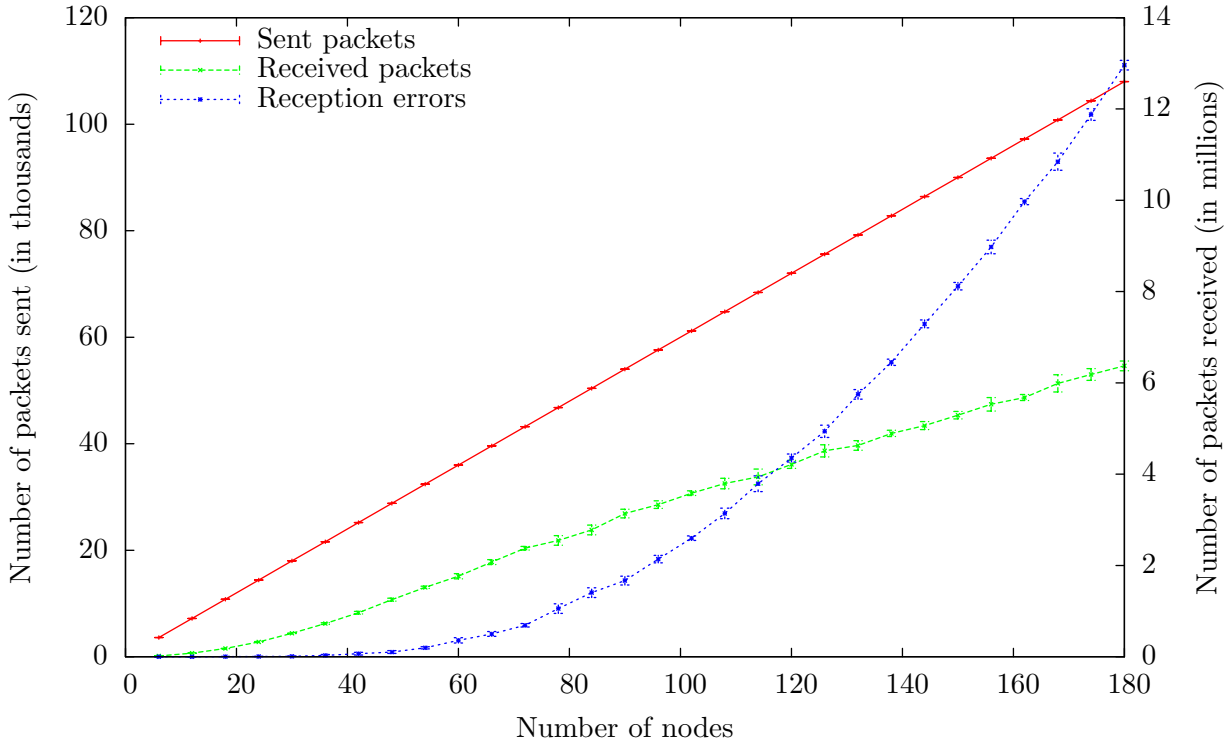


Figure C.1: Results from the highway example experiment