Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

TCP performance over WCDMA

Hynderick, Olivier; Raes, Sven

Award date:
2004

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 02. May. 2024

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'informatique
Année academique 2003-2004

TCP performance over WCDMA

Olivier Hynderick
Sven Raes

Mémoire présenté en vue de l'obtention du grade de Maître en Informatique

Abstract

This report deals with the analysis of the performance of the widely spread Transport Control Protocol over 3G UMTS networks based upon data provided by simulations executed on Network Simulator Version 2. This paper starts with explaining and describing the underlying UMTS network in further detail. After describing the basic and common features of TCP, the behaviour of the most common and used TCP versions is then analysed, merely based on their theoretical implementations. These theoretical features are followed by an evaluation and more thorough study of their respective behaviours thanks to various simulations in several contexts.

Résumé

Le contenu de ce mémoire tend à analyser les performances du protocole TCP opérant au sein d'un réseau UMTS. L'analyse s'appuyant notamment sur des simulations réalisées au moyen du simulateur de réseau NS2. Dans un premier temps, ce travail tend à analyser, d'un point de vue théorique, la norme UMTS elle-même pour en suite se focaliser sur les principales versions du protocole TCP. Ces études théoriques servant entre outre de base pour l'évaluation du comportement de TCP au sein de différents contextes de simulation choisis pour représenter des situation courantes opérant au sein d'un réseau UMTS.

We would like to thank our supervisor, Mr. Laurent Schumacher
and his assistant Hugues Van Peteghem
for their follow-up during the entire writing of this report.
We would also like to thank the entire staff of the Communication Department
of the Aalborg University,
who stood by us during the first six months of this project.
We also grant our gratification to Leon Bette, Eddy Raes and Fernanda Opdenacker for their
proofreadings of this report.
Finally, many thanks to our families and girlfriends for
their support during the writing of this paper

Chapter I: Introduction 2
Sources 4
Chapter II: UMTS overview 6
II.A Introduction 6
II.B UMTS network architecture 6
II.C UTRAN Radio Interface Protocols 8
The Radio Resource Control Protocol (RRC) 9
The Radio Link Control protocol (RLC) 10
The Medium Access Control Protocol (MAC) 12
II.D Physical layer 13
II.E UMTS link layer characteristics 15
High delays 16
Bandwidth 16
Asymmetric 17
Error patterns 17
II.F Sources 18
Chapter III: NS2 Simulator and UMTS extension 20
III.A Introduction – NS2: The Network Simulator 20
Discrete Event Simulation 20
Network Simulator version 2 [3] 21
Implementation 21
Topology 23
Agents 23
Trace-files 23
III.B Improvements to the "Strathclyde University" NS2 UMTS module 24
Introduction 24
Differences between TCP and UDP w.r.t. SF calculation 25
Differences w.r.t. bandwidth allocation between standard and implementation 25
First set of modifications 27
Second set of modifications 33
Third set of modifications 35
Conclusion 36
III.D Sources 37
Chapter IV: Transport Control Protocol Basics 38
IV.A General TCP features 38
IV.B Reaching an equilibrium 42
IV.C Conservation of the equilibrium 43
IV.D Sources 46
Chapter V: TCP Versions 48
V.A Introduction 48
V.B Overview of the common TCP Versions 48
TCP Tahoe (Distributed with 4.3 BSD UNIX) 49
TCP Reno 51
TCP New Reno [8] 52
TCP Vegas 54
TCP SACK (Selective ACKnowledgement) 55
TCP FACK (Forward ACKknowledgement) 56
V.C TCP behaviour 57
V.D Conclusion 63

V.E Sources 64
Chapter VI: Analysis and Mitigations 66
VI.A Introduction 66
VI.B Metric 66
VI.C Constant error rate environment 67
LL recovery 67
Queue Level 72
VI.D Bandwidth fluctuation 78
VI.E Error rate fluctuation 83
VI.F Proposed Solutions for TCP 89
VI.G Sources 92
Chapter VII: Conclusion 94

Acronym

AM	Acknowledged Mode
ACK	Acknowledgement
AIMD	Additive Increase Multiplicative Decrease
ATM	Asynchronous Transfer Mode
ARQ	Automatic Repeat Request
BDP	Bandwidth Delay Product
BoD	Bandwidth on Demand
BS	Base Station
BSC	Base Station Controller
NodeB	Base station Node
BTS	Base Transceiver Station
BSD	Berkeley Software Distribution
b	bit
BER	Bit Error Rate
bps	bits per second
BLE	Block Error
BLER	Block Error Rate
BCH	Broadcast Channel
BCCH	Broadcast Control Channel
BMC	Broadcast/Multicast Control
B	Byte
Bps	Bytes per second
CS	Circuit Switched
CDMA	Code Division Multiple Access
CCTrCH	Coded Composite Transport Channel
CCCH	Common Control Channel
CPCH	Common Packet Channel
CTCH	Common Traffic Channel
CWND	Congestion Window
CBR	Constant Bit Rate
CRNC	Controlling RNC
CN	Core Network
CRC	Cyclic Redundancy Check
DCH	Dedicated Channel
DCCH	Dedicated Control Channel
DPCCH	Dedicated Physical Control Channel
DPDCH	Dedicated Physical Data Channel
DTCH	Dedicated Traffic Channel
DSCH	Downlink Shared Channel
DRNC	Drift RNC
EDGE	Enhanced Data GSM Environment
ECC	Error Control Coding
ECN	Explicit Congestion Notification
FTP	File Transport Protocole
FIN	Finish
FACH	Forward Access Channel
FACK	Forward Acknowledgment

FEC	Forward Error Coding
FER	Frame Error Ratio
GGSN	Gateway GPRS Support Node
GMSC	Gateway MSC
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communication
GSN	GPRS Support Node
GTP	GPRS Tunnelling Protocol
GUI	Graphical User Interface
HHO	Hard Handover
HS-DSCH	High Speed Downlink Shared Channel
HLR	Home Location Register
I-TCP	Indirect TCP
ISN	Initial Sequence Number
ISDN	Integrated Services Digital Network
IuCS	Interface circuit Switched network
IuPS	Interface Packet Switched network
IMT-2000	International Mobile Telecommunications 2000
ITU	International Telecommunications Union
ICMP	Internet Control Messaging Protocol
IP	Internet Protocol
LL	Link Layer
LTN	Long Thin Network
MSS	Maximum Segment Size
MTU	Maximum Transmission Unit
MAC	Medium Access Control
ms	millisecond
ME	Mobile Equipment
MSC	Mobile Services Switching Centre
MS	Mobile Station
MT	Mobile Terminal
NACK	Negative ACK
NS2	Network Simulator 2
NAK	Not Acknowledged
OSI	Open System Interconnection
OVSF	Orthogonal Variable Spreading Factor
pkt	packet
PDCP	Packet Data Convergence Protocol
PS	Packet Switched
PCH	Paging Channel
PCCH	Paging Control Channel
PDC	Personal Digital Communication
PCPCH	Physical Common Packet Channel
PDSCH	Physical Downlink Shared Channel
PHY	Physical Layer
PRACH	Physical Random Access Channel
P-CCPCH	Primary Common Control Physical Channel
PU	Processing Unit
PDU	Protocol Data Unit
PSTN	Public Switched Telephone Network

QoS	Quality of Service
RAN	Radio Access Network
RB	Radio Bearer
RLC	Radio Link Control
RNC	Radio Network Controller
RRM	Radio Resource Management
RRC	Radio Ressource Control Protocol
RACH	Random Access Channel
RED	Random Early Detection
RTP	Real-Time Transport Protocol
RFC	Request For Comment
RTO	Retransmission Time Out
RTT	Round Trip Time
s	Second
S-CCPCH	Secondary Common Control Physical Channel
SSH	Secure Shell
SACK	Selective Acknowledgment
SAP	Service Access Point
SDU	Service Data Unit
SGSN	Serving GPRS Support Node
SRNC	Serving RNC
SMS	short message service
SRB	Signalling Radio Bearer
sshthresh	Slow Start Threshold
SHO	Soft Handover
SF	Spreading Factor
SIM	Subscriber Information Module
SYN	Synchronize
TE	Terminal Equipment
3G	Third Generation
TSG	Time Sequence Graph
TTL	Time To Live
TCP	Transmission Control Protocol
TTI	Transmission Time Interval
TR	Transparent Mode
TF	Transport Format
TFCI	Transport Format Combination Indicator
TFI	Transport Format Indicator
USIM	UMTS Subscriber Identity Module
UTRAN	UMTS Terrestrial Radio Access Network
UM	Unacknowledged Mode
UMTS	Universal Mobile Telecommunication System
UDP	User Datagram Protocol
UE	User Equipment
VLR	Visitor Location Register
VOIP	Voice Over IP
WCDMA	Wideband Code Division Multiple Access
WAP	Wireless Application Protocol
w.r.t.	with respect to
WWW	World Wide Web

Chapter I: Introduction

More than twenty years of ongoing modifications and optimisations have made TCP or Transport Control Protocol, the most solid transport protocol for computer networks. Today, more than 90% of Internet Traffic is carrying TCP Data. *Figure I.1* depicts the overall domination of this protocol on the Internet. On the other hand, wireless networks are an emerging technology of these days. During the last years, both the Internet and mobile systems grew extremely fast. It is natural that nowadays these two worlds are converging rapidly. That is, future networks will be a combination of high speed wired networks and wireless links. The adaptation of protocols designed for the wired Internet, such as TCP, to the wireless specificity and the design of wireless systems, adapted to Internet protocol stack is an area of wide interest. Next to that, experts say that the amount of data traffic on wireless networks is growing exponentially, whereas the growth for simple voice call traffic is nearly at a halt ([3] pages 167-168). Added to this the fact that, the most interesting services that could be provided on the next generation of mobile wireless networks, will definitely involve data traffic. A good example of such a service would be video conferencing, where the callers could simply see each other on the high definition color screen of their handset. One of the reasons of existence of Third Generation (3G) Wireless networks is the desire of the network operators and service providers to be able to offer those financially interesting services to their customers all over the network. Another important development is the upcoming of Video Phoning or Voice-Over-IP (VOIP). If conventional telephone calls could be routed through a packet switched network, not only could this benefit the overall cost for the consumers. On top of that it will optimise network resources allocations as well. Therefore, it is rather logical to assume that most of the applications that will be provided by 3G networks will be using TCP, as it is well established and the dominant protocol on the Internet.

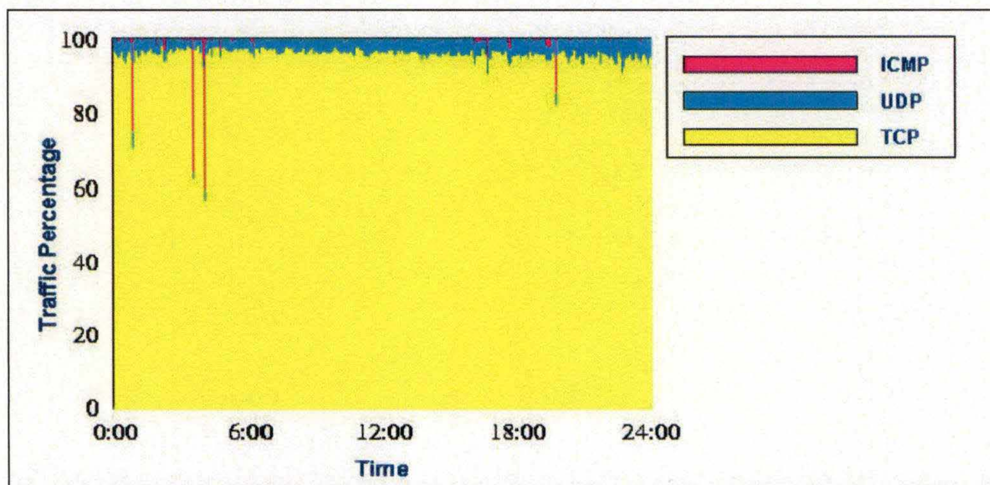


Figure I.1: The Internet traffic distribution in terms of byte over a 24-hour period, gathered from [1].

TCP has been designed as a highly reliable connection oriented protocol between hosts in a packet switched computer communication network [2]. One of the main features integrated in the TCP implementation is its well-designed flow and congestion control mechanisms, which enables it to dynamically adapt to the varying network load. The downside of these mechanisms is that they were designed with the assumption that packet losses were due to network congestion. As long as we

consider wired networks, this is mainly the case, but if wireless networks are taken into account, these assumptions do not hold anymore. Each wireless link and wired networks have different and distinctive transmission characteristics. The wired networks are relatively reliable as compared to the wireless links. Wireless links bring some serious issues as high Bit Error Rate (BER) because wireless hosts use radio transmission for communication. Generally, the BER of wireless links is higher than that of wired networks. Transmitted packets are lost in the wireless links by fading channels, shadowing or other external factors. Or even the lower bandwidth provided, caused by the fact that spectrum is a rare resource for wireless applications. Bandwidth and capacity, for a given application, are limited and therefore result in a low maximum rate at which packets can be transmitted over the wireless link. Available bandwidth in the wireless links may be up to 2Mbps, but that of wired links is about 10 to even 1000Mbps. An important characteristic is the long Round Trip Time (RTT). Wireless links exhibit longer delays than would wired media. In wireless networks, since Radio waves travel at the speed of light, the same as the transmission speeds in wired media, it is the lower bandwidth in wireless networks that increases the time a packet to get transmitted. This degrades the TCP throughput[1] and increases the experienced delay perceived by the user. This is only one of the reasons why TCP may suffer from some serious performance drawbacks if used unchanged in a wireless environment.

The objective of this report is to evaluate the performance of several TCP versions used on a wireless network, a 3G network or UMTS network to be precise. An introduction to the UMTS technology and theoretical description of the TCP algorithms including their different versions will be followed by some performance evaluations based on results provided by our NS2 – The Network Simulator version 2 upgraded by an UMTS simulation add-on module [4], modified to make it suit our needs [5]. Chapter III, section A of this report will deal with a general overview of this widely used simulation tool, whereas section B of this same chapter deals with the modifications that have been made on the add-on UMTS module, together with the reasons for those changes.

---

[1] In the remaining of this document, we define the throughput as the amount of data transferred in one direction over a link divided by the time taken to transfer it. (We agree that this definition may seem informal, therefore a more accurate one will be provided whenever throughput figures are shown)

Sources

[1] Claffy K., Miller G., Thompson K., **The nature of the beast: recent traffic measurement from an Internet backbone**, Proceedings of INET '98, Geneva, Switzerland, 1998.

[2] Postel, Jon. **Transmission Control Protocol - DARPA Internet Program Protocol Specification,** RFC 793, September 1981.

[3] Tanenbaum, A., **Computernetwerken**, Vierde editie, Pearson, ISBN 90-430-0698-X.

[4] Martin P., Ballester P., **UMTS extensions for NS2**.
Available at http://www.geocities.com/opahostil/, Last visited: September 28th, 2003.

[5] Hynderick. O, Raes. S, **UMTS extension for NS2.**
Available at http://www.info.fundp.ac.be/~lsc/Research/ns-2_UMTS/, Last visited: September 28th, 2003.

Chapter II: UMTS overview

II.A Introduction

This chapter gives an overview of the UMTS technology. As a procedure, we will first describe the main components of a UMTS network and their interactions between each others (section II.B). Next, we will be focused on the UMTS protocol stack (section II.C and II.D). In these sections, we will follow a top-down approach by starting with the description of the top most layer until the lowest layer. Finally we will summarise the main features of a UMTS link layer (section II.E), with other words, the characteristics of a path crossing a UMTS network.

II.B UMTS network architecture

The three main parts composing the UMTS Network architecture described below are the Core Network (CN), the UMTS Terrestrial Radio Access Network (UTRAN) and the User Equipment (UE).

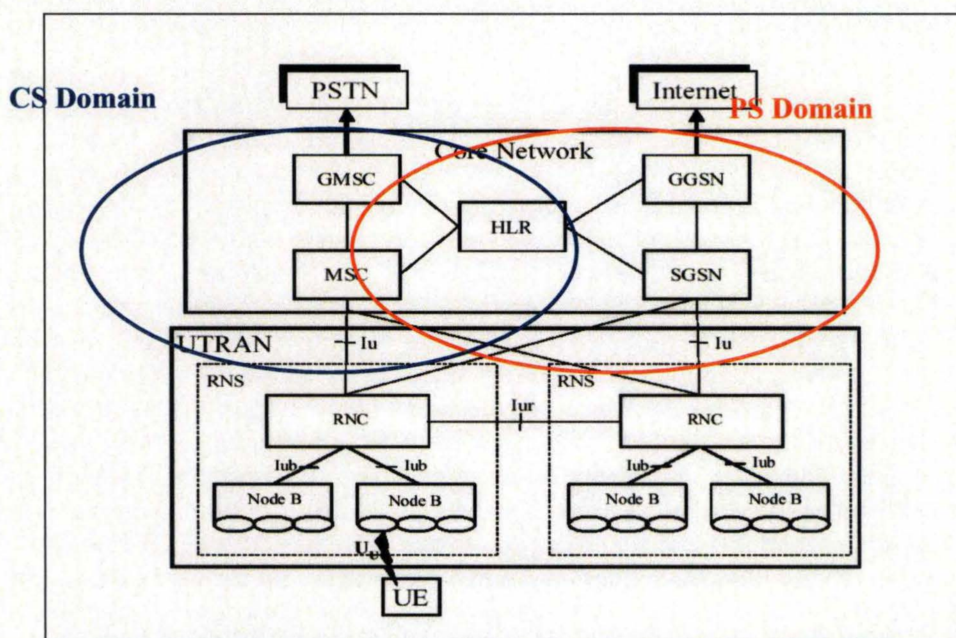


Figure II.2: The UTRAN Architecture. Adapted from [1]

The UE is split between the Mobile Equipment (ME), representing the radio terminal used for communications over the radio interface (in fact, the handset), and the UMTS Subscriber Identity Module (USIM). The USIM is, the in the handset plugged Smart-Card, that contains subscriber specific data. The wireless connection between the UE and the UTRAN is called Uu (also known as "air" or "radio") interface. In case of a cell phone, the UE is called Mobile Station (MS) which is also composed of an ME and a USIM [1], [2].

The UTRAN can be divided into two major components, the NodeB and the Radio Network Controller (RNC). The primary functions of the NodeB components are handling the radio communication link such as coding and spreading and to some extent some basic Radio Resource Management (RRM) operations such as inner loop power control. On the other hand, the RNC manages the radio resources and handles handovers between different NodeBs. These handovers are called soft-handovers, which means that the UE is connected to multiple NodeBs (i.e. cells). Softer-handovers, where the UE is connected to multiple sectors of one NodeB, are managed by the NodeB itself. The interfaces, between the NodeBs and the RNCs are called Iub. The Iub interface is in charge of extending the transport channel capabilities (see section II.C) provided by the (radio) physical layer up to the RNC. The Iur interface is the interface between different RNCs. This link enables soft handovers between two NodeBs that are controlled by different RNCs. The RNC controlling one NodeB (the RNC interfaced directly with the NodeB through the Iu link) is indicated as the Controlling RNC (CRNC) of the NodeB. Its main functions are the congestion control, the admission control and the code allocation for new radio links to be established in its set of cells. The RNC also has a logical role, which can either be Serving RNC (SRNC) or either Drift RNC (DRNC). The SRNC for a UE is the one that terminates the Iu link to and from the CN. So one UE only has one and solely one SRNC while connected to UTRAN. While taking the role of DRNC, the RNC controls the cells used by the UE, but is not the UE's SRNC. So the data path from a UE to the CN using a DRNC has to pass the SRNC just before exiting the UTRAN and entering the CN. In other words, the information is routed from the DRNC, which controls the cells used by the UE, towards the SRNC. It is important to note that each physical RNC normally contains all the CRNC, SRNC and DRNC functionalities.

The CN can be seen as the link between the UTRAN and the standard public telephone and Packet Switched (PS) networks. It is interfaced with the UTRAN through the Iu interface. It mainly consists of switches and databases and is divided into a Circuit Switched (CS) domain (for providing connections like existing telephony) and a PS domain (in order to provide packet data services such as the Internet), cfr. *Figure III.1*. We have to make a distinction between the connection of UTRAN towards the CS network, called IuCS and towards the PS network, the IuPS. Mobile Switching Centres (MSC) form the backbone of the CS domain which is inherited from the GSM networks, and the equivalent for the PS domain is the GPRS Support Nodes (GSN). We have the Serving GSN (SGSN), a switch that is interfaced with the UTRAN through the IuPS interface and directly interfaced with the Home Location Register (HLR). The HLR is a database located in the user's home system that is meant to store the master copy of the user's service profile. This copy contains information about allowed services, forbidden roaming areas and other information of the same kind. It also stores location information about the UE at the level of the serving systems. This info is used for routing incoming calls or SMS. The SGSN has the same function for the PS network as the MSC has for the CS network. The MSC main function is to switch the CS transactions. It is often combined with the Visitor Location Register (VLR), a database similar to the HLR that stores a copy of the visiting user's service profile, as well as more precise location information of the visiting user's UE inside the serving system. The services systems are connected through gateways to the external networks. We speak of Gateway MSC (GMSC) in the case of the CS domain and of Gateway GSN (GGSN) in the case of the PS domain. The GMSC is interfaced to networks as PSTN or ISDN. The GGSN on the other hand is interfaced with PS networks as those based on the Internet Protocol (IP).

II.C UTRAN Radio Interface Protocols

UMTS allows a user/application to negotiate bearer characteristics that are the most appropriate for carrying information. A bearer is defined as an information transmission path of a defined capacity, delay, BER, etc. A bearer service is the type of telecommunication service that provides the capability of transmission between access points [9]. UMTS also provides the possible to change bearer properties via a renegotiation procedure in the course of the active connection. *Figure II.2* depicts the UMTS bearer service layered architecture consisting of several levels of services with the end-to-end service being on the top most layer. This section describes the protocols controlling the radio bearer services. Those protocols are typically implemented in the UE protocol stack and, regarding to the access network, in the Node or RNC entities.

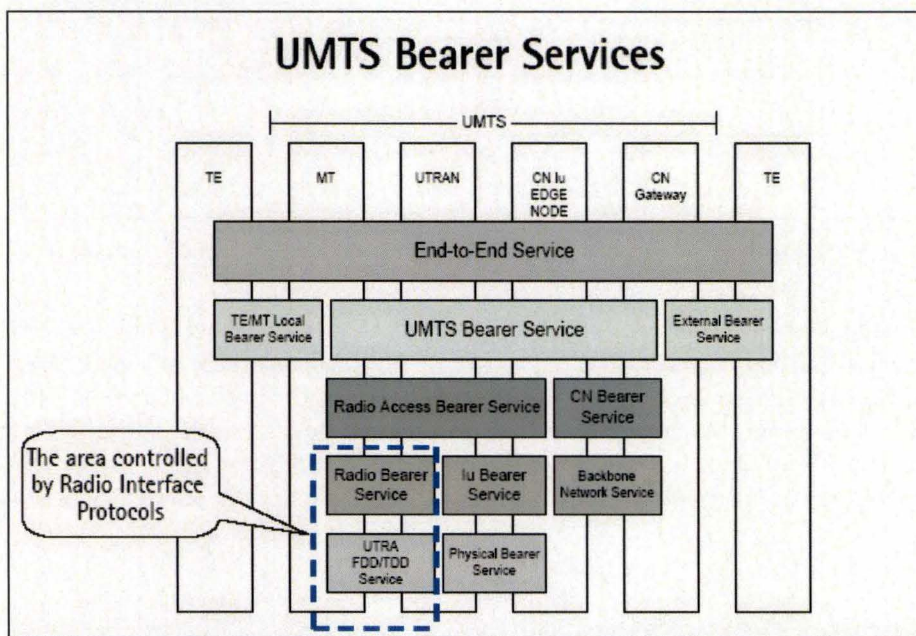


Figure II.3: The UMTS Bearer Services. Taken from [3]

Figure II.3 depicts the overall protocol architecture of the UTRAN radio interface. Although it seems rather different from the OSI model, it is still possible to find some similarities. *Figure II .3* proposes to gather protocols belonging to the radio interface into layer one (called the physical layer), layer two (called the data link layer) and finally layer three (called the network layer). Moreover, the radio interface is divided into two planes. The first plane, called user plane, carries all the data from the user (IP packets, SMS ...) meaning that this plane is completely transparent for UTRAN. The second plane, known as the control plane, handles all signalling between the handset and the network via the RRC layer.

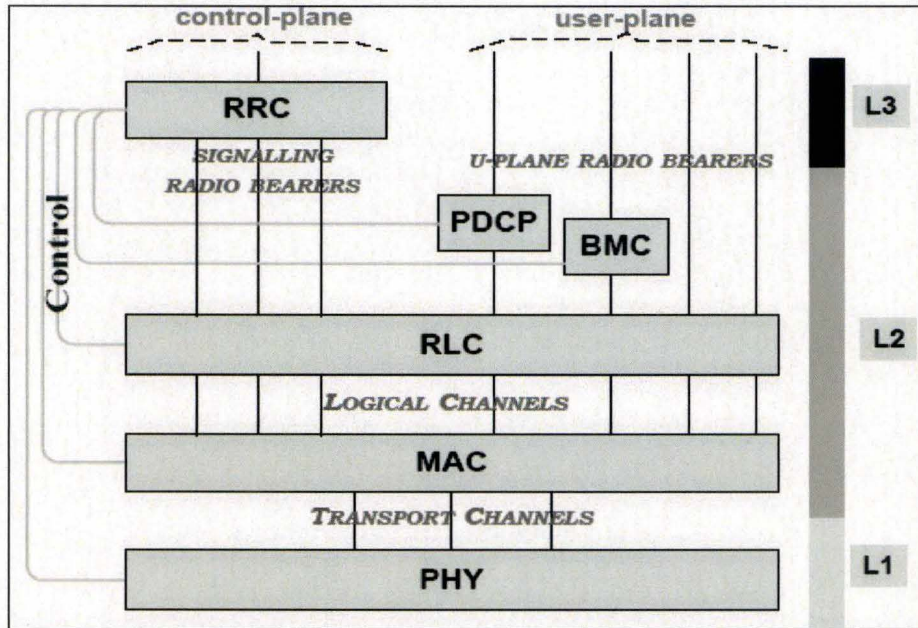


Figure II.4: The UMTS radio interface protocol architecture. Taken from [3]

In a layered architecture, a specific layer offers its services to the layer above and does so with the aid of the sub layer. In UTRAN, the Physical layer offers services to the MAC layer via the transport channels. The MAC layer offers services to RLC layer by means of logical channels. Finally, the RLC layer offers services to higher layers via service access points describing how the RLC layer handles data packets. The RRC layer does not respect perfectly a layered architecture, in a way that it interfaces directly with every lower layer, in order to measurement and configure the whole protocol stack.

The Radio Resource Control Protocol (RRC)

Most of the signalling between UE and UTRAN is contained inside RRC messages. A classification could be made between signalling information from the higher layer and signalling sent during procedures managed by the RRC layer (measurements, handovers, cell updates …). A third category can be added: information carried by RRC packets to lower layers of the stack (layer one and two). With the first category, the RRC layer behaves like a transfer channel, thus data is transparent for UTRAN. With the two last categories, RRC packets are the result of RRC procedures.

The RRC procedures depend on current UE modes and RRC states. The two basic modes of a UE are idle mode and connected mode. The handset stays in idle mode until the transmission of a request for establishing a RRC connection. Otherwise, in this mode, the UE is still able to receive system information (paging information…) or cell-broadcast-messages. To conclude, this mode is well adapted to enable the UEs to enter into sleep state.

The connected mode can be divided into 4 states. Switching between these states during a communication allows the UE to avoid wasting power. These states are Cell_DCH, Cell_FACH, Cell_PCH and URA_PCH.

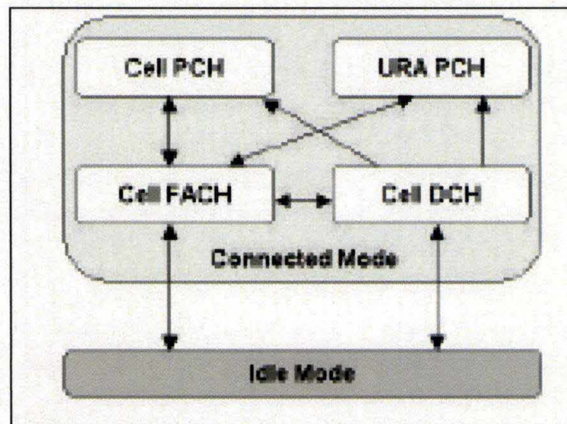


Figure II.5: The RRC states

Cell_DCH:	A dedicated channel is allocated to UE and the UE is known by RNC.
Cell_FACH:	RACH and FACH channels are used instead of a dedicated channel. Mobility control is managed by the UE
Cell_PCH:	UE is still connected but user data can not be sent and the UE can be reached only via PCH channel.
URA_PCH:	This state is close to the previous one, except that UE executes Cell Update procedures only when it switches between UTRAN registration areas

Table II.1: The four states of the connected mode.

Let's focus on web-browsing for example. There, data traffic activity can be separated into two phases. A first one corresponds to the downloading of a web page and a second one occurring when the user is viewing the content of a downloaded page. Web browsing is a succession of both these phases. Obviously, during the second phase, no information is sent. Therefore, it should be interesting to switch constantly between Cell_DCH and Cell_PCH state. During the first state, a dedicated channel is allocated to the UE. The user is thus able to receive data from the Internet. On the other hand, during a period of inactivity, the Cell_PCH allows the user not to consume energy and network resources, while keeping the connection alive for making further request.

The Radio Link Control protocol (RLC)

The Radio Link Control protocol provides segmentation and retransmission for both user data and control data. Retransmission is usually realised to improve the quality of the physical channel. However, this kind of operation may have a bad effect on the transmission delay. Moreover, radio resource is a scarce resource, thus it is very important to retransmit only what is really needed. In this view, the RLC layer provides three different access points corresponding to three different transport modes: the transparent mode (TR), the unacknowledged mode (UM) and the acknowledged mode (AM). The last one allows a high level of reliability for applications such as web browsing. On the other hand, the transparent mode can offer a streaming mode, useful for application like voice over IP.

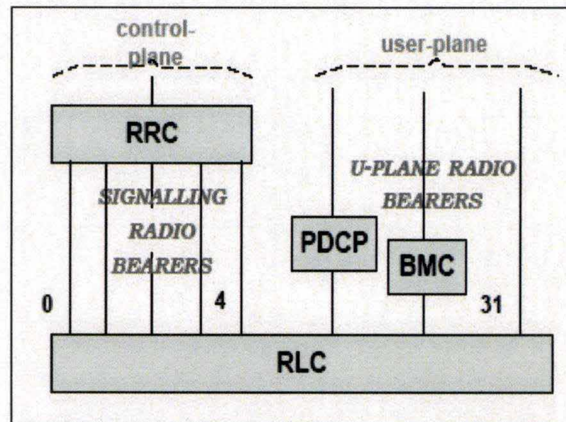


Figure II.6: The Radio Bearers. Taken from [3]

Note that, the service provided by the RLC layer to the control plane is called "Signalling Radio Bearer" (SRB). Moreover, if PDCP or BMC is not used over RLC, the service provided by the RLC layer is called "Radio Bearer" (RB).

1. Transparent mode (TR)

   In this specific mode, a RLC packet is only composed of a data field. This means that no header is added to the data from the higher layer. As no header is found in RLC packets, data segmentation has to be made from a predefined scheme. Therefore, in this mode, segmentation is unusual. Indeed, without header it is impossible to know if some information is missing on the receiver side. That's why transport mode is appropriate for voice data stream with small amounts of data. Erroneous packets can be discarded or marked. Therefore, in this mode, the RLC layer is nearly transparent for the data flow.

2. Unacknowledged mode (UM)

   In this mode, a header with a sequence number is added to the data. However, that does not mean that the retransmission will be carried out. In fact, the sequence information, carried in headers, makes segmentation easier. It is also possible to receive some integrity information about the transmission – like the part of packet that is not received. To conclude, this mode is well adapted, when reliability is not the main preoccupation and the data stream is quite irregular (with bursts).

3. Acknowledged mode (AM)

   In the acknowledged mode, an automatic repeat and request mechanism is proposed. This offers the possibility to improve the reliability of the logical channel. Indeed, thanks to a sliding window we can control the data flow. Moreover, with acknowledgment information, we can resend the information that was corrupted. However, reliability vs. delay can be controlled by the RRC layer. We also have the possibility to allow an out-of-sequence delivery. In conclusion, this mode seems to be adapted for applications like File Transport Protocol (FTP) clients or web browsers.

The Medium Access Control Protocol (MAC)

The MAC layer relies on the transport channels provided by the physical layer for providing logical channels to the RLC layer. Accordingly the main task, which is under the responsibility of the MAC layer, is the mapping between logical and transport channels. Regarding the logical channels, the main idea is that there is a specific channel for each type of data. The first distinction made, is on the differences between control and traffic information, coming from the differences between the control and the user plane. The next distinction made, is between the information belonging to one specific user or a group of users.

			Specificity	Recipient
Broadcast Control Channel:	**BCCH**	Downlink channel used to carry control information over the whole cell	Control	Group
Paging Control Channel:	**PCCH**	Downlink channel transferring paging information	Control	Group
Dedicated Control Channel:	**DCCH**	Point-to-point control channel	Control	User
Common Control Channel:	**CCCH**	Point-to-multipoint control channel	Control	Group
Dedicated Traffic Channel:	**DTCH**	Point-to-point channel designed to transfer user data	Traffic	User
Common Traffic Channel:	**CTCH**	Point-to-multipoint (and so unidirectional) channel for user information	Traffic	Group

Table II.2: The logical channels.

Where the logical channel specifies the kind of data flowing over the network, the transport channel defines how data is transferred. With other words, the MAC layer is the one responsible for selecting the right transport format (TF) for each transport channel. The TF is composed of a static and a dynamic part. The static part contains information (Transport Format Set ...) that will be fixed during the entire time this channel is in use. On the other hand, dynamic characteristics (transport bloc size, Transmission Time Interval, channel coding ...) can change over time.

Let's carry on this MAC layer description by an enumeration of the transport channels. Contrary to the logical channels, they are all unidirectional. Some of them are shared by a group of users (common/shared channel) and others, are specific to one user (dedicated channel). Common channels do not allow soft handover but some of them can have fast power control.

			Direction	Utilisation
Broadcast Channel:	**BCH**	channel designed to carry information for all the cell	Downlink	Shared
Paging Channel:	**PCH**	channel designed to carry paging information	Downlink	Shared
Random Access Channel:	**RACH**	channel designed to carry control information or small amounts of data	Uplink	Shared
Uplink Common Packet Channel:	**CPCH**	a RACH channel extension	Uplink	Shared

Forward Access Channel:	**FACH**	downlink channel used to carry control channel or small amounts of data	Downlink	Shared
Downlink Shared Channel:	**DSCH**	downlink shared channel used to carry dedicated control or user data	Downlink	Shared
Dedicated Channel:	**DCH**	downlink channel used to carry dedicated control or user data when greater bandwidths a required	Downlink or Uplink	Dedicated

Table II.3: The transport channels.

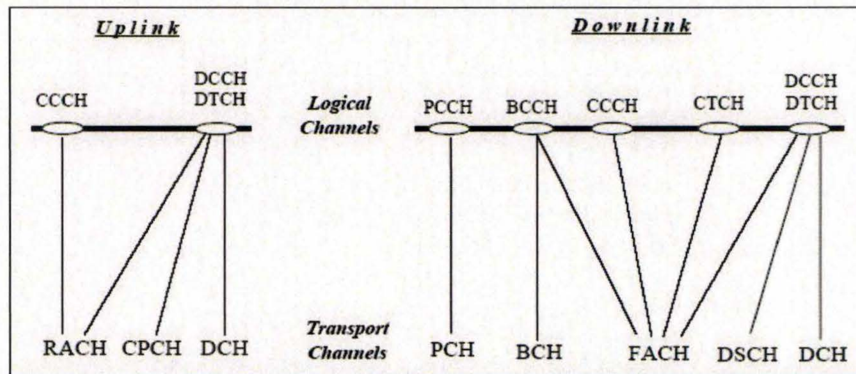


Figure II.7: The mapping between logical and transport channels. Taken from [3]

Figure II.6 shows the mapping between transport and logical channels. This picture shows that things are not predefined in a way that, there are, for some logical channels, several mapping possibilities between transports channels. As an example, data from DCCH and DTCH can be carried by DCH, CPCH or RACH. This mapping is done upon data flow requirements like the transmission rate. For example, on the uplink side, a DTCH may be matched with a DCH if the rate requirement is not too high.

Finally, the MAC layer is also in charge of several others activities like:

The identification of the right recipient on common transport channel
Multiplexing of several logical channels with the same Quality of Service (QoS) parameters into one transport channel
Dynamic transport channel switching
Some QoS management activities in order to provides priority between logical channels
Traffic volume monitoring

II.D Physical layer

The MAC layer by matching each logical channel to the right transport channel and selecting the right TF for each transport channel has already decided how to carry the data. The physical layer now has to support variable bit rate transport channels in order to carry data at the pace they are coming in. In other words, the physical layer is in charge of providing bandwidth-on-demand (BoD). It is important to underline that in a UMTS network architecture, the interface between the physical layer and higher

layers is represented by the Iub-interface (see section II.B).

Let's begin with the mapping between the physical and the transport channels

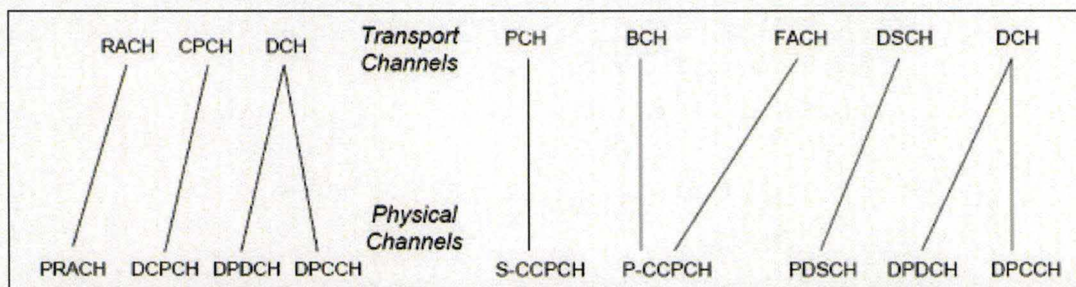


Figure II.8: The mapping between transport Channels and physical channels.

Primary Common Control Physical Channel :	**P-CCPCH**
Secondary Common Control Physical Channel :	**S-CCPCH**
Physical Random Access Channel :	**PRACH**
Dedicated Physical Data Channel :	**DPDCH**
Dedicated Physical Control Channel :	**DPCCH**
Physical Downlink Shared Channel :	**PDSCH**
Physical Common Packet Channel :	**PCPCH**

Table II.4: The physical channels.

Figure II.7 depicts the mapping between the transport and the physical channels. It shows that even for the DCH (where the channel is mapped into two physical channels), the mapping is predefined showing that the manner data is carried is decided somewhere else in the protocol stack. It is worth mentioning that on *figure II.7*, only physicals channels carrying information from/to transport channels are showed. Beside those channels, there are other channels carrying information relevant for the physical layer procedures.

One of the main tasks for the physical layer is the multiplexing between the transport channels (whatever their QoS parameters are). As a matter of fact, each transport channels carries self-describing information called the Transport Format Indicator (TFI). While multiplexing, the physical layer combines the TFI information from different transport channels to the Transport Format Combination Indicator (TFCI). As TFI may change from block to block, TFCI can be assimilated to a data stream. By the way of the multiplexing procedure, two flows are generated. The first one is carrying the multiplexed transport blocs and the second one carrying control information about the first one. Usually, both flows are carried by the same physical channel, excepted for the DCH. There, control and data flows are separated into DPCCH and DPDCH. Moreover, it might happen that more than one DPDCH is needed to at the same time. There, all the control information corresponding to those channels is still carried by only one DPCCH.

In order to provide all these physical channels with variable bit rates, the UMTS physical layer makes use of the Wideband Code Division Multiple Access (WCDMA) as its air interface. In this system, the user information bits are spread over a wide bandwidth by multiplying the user data with quasi-random bits (called chips) derived from spreading codes. In UMTS, the chip rate is fixed at 3.84 Mcps leading to a carrier bandwidth of approximately 5 MHz. In order to provide highly variable user data rates,

WCDMA splits the time in frames of 10ms, during which the user data rate is kept constant. However, the data capacity can change from frame to frame. Given the fact that in UMTS the chip rate always remains constant, the physical layer makes use of standard Orthogonal Variable Spreading Factor (OVSF) codes in order to adapt the frame capacity. Indeed, by using sequences with different lengths, several processing gains can be used, according to the user data rate. In UMTS, OVSF codes are obtained from a code tree depicted in figure II.8.

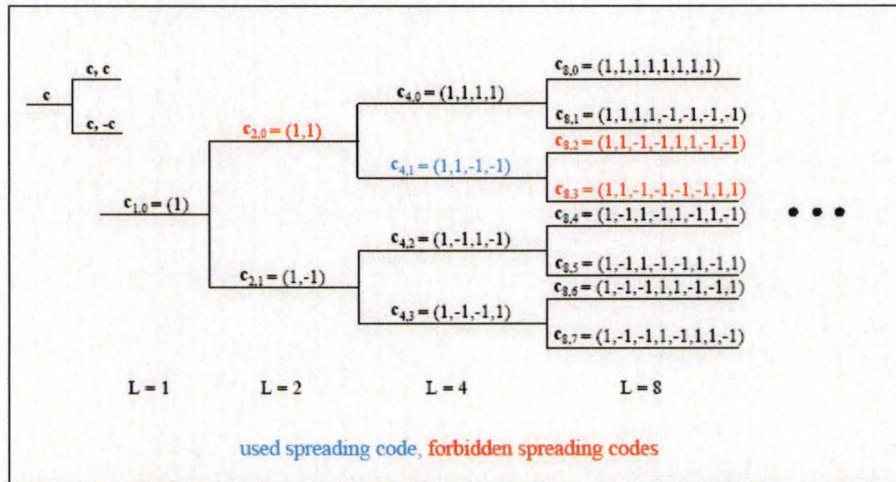


Figure II.9: The channelization codes three. Taken from [4]

These codes are also called channelization codes. Indeed, in WCDMA orthogonal codes are used to separate transmissions from a single source. More precisely, in uplink, the different (physical) channels from a same user are separated by way of channelization code. On the other hand, on the downlink, channelization codes are also used to separate users between each other. It is important to note that channelization codes are rare resources in a way that not all codes from the tree can be used at the same time i.e. are orthogonal between each other. *Figure II.8* shows that a first level code can be divided into 2 second level codes or a 1 second level code and 2 third level codes. Of course increasing the level code, by the same way, increases the processing gain and thus reduces the frame capacity. UTRAN provides four level one codes meaning that, in downlink, four users can make use of a level 1 code at the same time. If another user also places a request for a level one code, one first level code must be divided into 2 second level codes. In other words, instead of having 4 first level codes we get 3 first level codes and two second level codes. To conclude, it is imperative, in order to avoid wasting bandwidth, to handle cleverly the allocation of those codes.

II.E UMTS link layer characteristics

The scenario of concern is the wireless link between the UE and the NodeB in the UMTS network. It is the last hop to the end user. Therefore, the whole data traffic coming from or going to the UE passes through the NodeB, the last node before the wireless link. Besides wireless connections, the UMTS network will also provide mobility and BoD. Accordingly, the characteristics of the UMTS link layer (in terms of error rate, data rate, delay…) depend, of course on its air interface, but also on the way mobility or bandwidth allocation is handled by the network. Indeed, as long as the air interface of the

wireless link has an impact on the QoS provided by the network, it is also true for mobility or bandwidth allocation. As an example, the link layer can experience an increase of its error rate during a handover. In this section, we will present some aspects of the link characteristics of a UMTS network.

High delays

Typically, the delays on an UMTS link are quite larger than what can be expected in wired networks. That is mostly due to the complexity of the physical layer. Indeed, a wireless interface does not provide any protection against interferences. The physical layer also has to deal with multi-path propagations. Therefore, in order to provide wide coverage with enough reliability, complex error recovery or interleaving mechanisms need to be applied. Besides the latency coming from the wireless interface, other delays coming from the access or the core network have also to be taken into account. Accordingly, a typical RTT varies between a few hundred milliseconds and one second [6]. More specific figures about delays can be found in [5].

The delays inside a UMTS network are not only high, but also can vary a lot. As a matter of fact, in those kinds of networks, delay spikes are known to occur often. Delay spikes are a sudden increase of the latency inside the communication path. In a UMTS link, it is likely to experience an increase of the typical RTT by several times [6]. Delay spikes can occur due to mobility questions like handovers or a link recovery (see section II.C) due to temporal losses of the radio coverage.

Bandwidth

Initial UMTS networks are expected to have a bit rate up to 64kbps in the uplink and up to 384kbps in downlink [6]. According to the bandwidth required by an application and provided by the network, the system belongs to the so-called Long Thin Network (LTN). That means that we have a long delay along with moderate bandwidth. The bandwidth delay product (BDP) is a way to characterize the topology in terms of how much data the pipe can hold and how long delays it introduces. BDP is defined as the bandwidth multiplied by the end-to-end delay. This causes much information to stay in the network for some time during the transfer. This will affect the communication between the sender and the receiver, since signalling is delayed. This can therefore limit the performance of communication that relies on that kind of signalling. Many of these effects are described in [7] and [8].

UMTS networks belong to the kind of wireless networks that implement BoD. BoD is a mechanism implemented in the link layer (LL) providing a fair and efficient sharing of the available resources among a large number of users moving inside one cell. In a UMTS network, as long as an application does not send enough information, a shared channel can be used. Doing so, the user will experience quite a short access delay but a low bandwidth capacity will be offered to him. By requiring much transfer capacity in uplink, downlink or both, the user may force the wireless device to open a dedicated channel (see section II.C). Such a channel can provide much more bandwidth capacity but also can adapt itself to a changing demand from the application, on the transfer capacity point of view. In this case, the user will experience a quite longer time to open the channel. Moreover, the allocation for this kind of channel could be done for a specific amount of time, or as long as the application is sending enough data. Typically, the request for opening of such a channel will be done on the ground of the

buffer occupancy or the expected buffer occupancy. If the sender can not anticipate the arrival of additional traffic, poor performance will be experienced during the time it takes to setup the channel.

The transfer capacity provided by the network may also change accordingly to some mobility factors or simply because of other users. As the network has to provide a fair enough repartition of the available resources among users, arriving and departing users can increase or decrease the available bandwidth inside one cell (see section II.D). UMTS networks offer the possibility to the users to move from one cell to another without resetting their dedicated connections. All of these features allow data to flow over the wireless interface without interruption during handover procedures (due to re-opening time for example). In this specific situation, the scenario of concern will be the bandwidth the user will get in the new cell in comparison with the one he got in the previous cell.

Asymmetric

A UMTS network may run asymmetric uplink and downlink connections. Mostly that is due to some power consumption reasons. In addition, it is an efficient way to allocate bandwidth that only gives the necessary resources to an application. We should bear in mind not to slow down the control information of the transport protocol used over the channel.

Error patterns

One key issue for data communication flowing over a wireless segment, is the error characteristic of wireless links. On wired networks, since there are no interferences, few transmission errors occur. On the other hand, for a wireless segment, due to the weakness of its transport means, the opposite is true. Beside this error characteristic, in wireless networks providing mobility, it is possible to distinguish two typical error patterns. The first one can be described as a transient random error pattern. The second one can be seen as a burst error of a relatively persistent nature. Mostly, the second pattern happens in a situation of a fading channel due to some topography reason. Without any loss-recovery mechanisms, the probability to see transmission errors occurring is called Bit Error Rate (BER). Fortunately, the physical layer that is handling the wireless interface, provides some mechanisms to allow partial recovery. If some bit errors cannot be recovered, it will result in a Block Error (BLE). The probability to see this happen is the BLER (Block Error Rate).

II.F Sources

[1] González Parada, E., Cano García, J.M., Aguilar González, J.A., Reyes Lecuona, A. and Díaz Estrella, A., **Wireless Internet Access based on UTRA TDD**, Proceedings of European Wireless 2002 (EW2002), Florence, Italy, February 2002. Available at http://docenti.ing.unipi.it/ew2002/proceedings/105.pdf, Last visited: September 28th, 2003.

[2] Holma H., Toskala A., **WCDMA for UMTS – Radio Access For Third Generation Mobile Communications**, Second Edition, Wiley, ISBN 0-470-84467-1.

[3] Vialén J., **UTRAN Radio Interface Protocols**, Lecture. Available at http://www.comlab.hut.fi/opetus/238/lecture7_RadioInterfaceProtcols.pdf, Last visited: September 28th, 2003.

[4] Mann Pelz R., **WCDMA Transmission schema**, Lecture. Available at http://www.ant.uni-bremen.de/teaching/nsfzm/Chapter-2-2c.pdf, Last visited: September 28th, 2003.

[5] 3GPP, TR 25.853, **Delay Budget within the Access Stratum**, March 2001. Available at http://www.3gpp.org/ftp/Specs/archive/25_series/25.853/25853-400.zip, Last visited: September 28th, 2003.

[6] Inamura H., Montenegro G., Ludwig R., Gurtov A., Khafizov F., **TCP over second (2.5G) and third (3G) generation wireless networks**, RFC 3481, February 2003.

[7] Montenegro G., Dawkins S., Kojo M., Magret V., Vaidya N., **Long Thin Network**, RFC 2557, January 2000.

[8] Allman M., Dawkins S., Glover D., Griner J., Tran D., Henderson T., Heidemann J., Touch J., Kruse H., Ostermann S., Scott K., Semke J., **Ongoing TCP Research Related to Satellites**, RFC 2760, February 2000.

[9] 3GPP, TS 21.905, **Vocabulary for 3GPP Specifications**, October 2001. Available at http://www.3gpp.org/ftp/Specs/2001-12/Rel-4/21_series/21905-440.zip, Last visited: August 30th, 2004.

Chapter III: NS2 Simulator and UMTS extension

This chapter gives an introduction to network simulator NS2 and its internal functionalities. Once this event driven simulator is described more thoroughly, we will handle the UMTS add-on module. This module, provided by [4], had to be adapted in order to suit our simulation needs [2]. All modifications that were integrated are summed up and explained in section III.B.

III.A Introduction – NS2: The Network Simulator

NS2 is an object-oriented, discrete event driven network simulator developed at UC Berkeley written in C++ and OTcl. NS2 is primarily useful for simulating IP networks. NS2 implements network protocols such as TCP and UDP, traffic source behaviour such as FTP, Telnet, or CBR traffic generators.

We can consider NS as an OTcl script (Object-oriented Tcl Script) interpreter that has a simulation event scheduler, a network component object library, and network setup module libraries. To setup a network simulation, an OTcl script that initiates an event scheduler should be written. It sets up the network topology using the network objects and the network setup module functions (or plumbing functions), and tells traffic sources when to start and stop transmitting packets through the event scheduler. In NS2, the event scheduler keeps track of simulation time and launches all the events in the event queue scheduled for the current time by invoking appropriate network components.

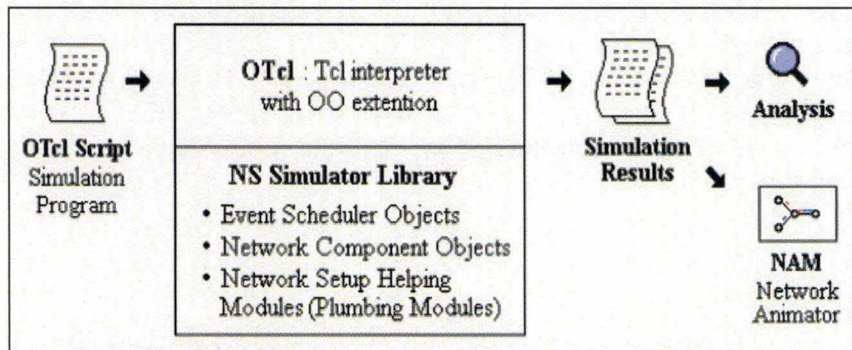


Figure III.10: NS2 simplified architecture. Taken from [1]

Discrete Event Simulation

Simulation is used in this thesis to study how proposed solutions mitigate the well-known problems associated with TCP over wireless media (see chapter VI). Discrete event simulation is appropriate to use since the simulated network is IP-based, where each handling of a data packet will generate an independent event. There are two main advantages with simulations for research and evaluation purposes. First, simulations provide an easy way to study how different assumptions in the model affect the final result, without having to implement the model in a real system. Second, the implementation of a model is often quite less time consuming than that of a real system. Furthermore, discrete event

simulations can be executed very fast, since only the actual events have to be simulated and not the time in between. A discrete event simulation can be therefore regarded as a system moving from one state to another in discrete steps. Discrete time simulation is suitable in our case since we wish to simulate a packet-based network. One thing common for all simulations is that it only reproduces the behaviour of the real world and that of course some characteristics may differ between different implementations and simulations. There are other simulators for simulating IP networks available but one of the most famous is definitively the Network Simulator NS2.

Network Simulator version 2 [3]

We will only use its capabilities for simulating IP networks but other types, like ATM, can also be simulated thanks to add-on modules. The main advantage of NS2 is that it is widely used, especially in the academic community. There is a great range of extensions to NS2 available on the internet. The fact that it is license-free and open-source, encourages users to share information about bugs and features, while the development is in constant progress. The simulator is mainly written is C/C++ and it is implemented according to the object oriented paradigm. The behaviour of the links, nodes and other functionalities are implemented in C++ but an interpreter is used as a front-end. The interpreter enables to set up a topology in a rather smooth manner, instead of having to edit the C/C++ code and recompile for every new simulation. The front-end is written in OTcl. This object oriented tool command language is an interpreted language with an integrated support for objects. The objects in C++ are linked to the OTcl objects in such way that the behaviour of the model objects in C++ can be controlled by OTcl objects. This approach offers the benefit of the powerful and fast executing features of C/C++ but also the flexibility that a script language like OTcl provides. There are of course drawbacks with the shared object approach, especially that procedures are necessary in order to make the C++ objects visible to the Tcl part of the system. One of the major problems of NS2 is that it is rather poorly documented or that some parts are completely out of date. This leads to problems for users wanting to learn how to use NS2 in a more advanced way. The only way to fully understand it is to look at the source code itself, which is of course very time consuming.

Implementation

As mentioned earlier NS2 makes use of a shared object implementation. As shown in *Figure III.2*, every object in OTcl has a corresponding object in C++. Since the objects are shared, their attributes can be accessed from both the Tcl and the C++ interface.

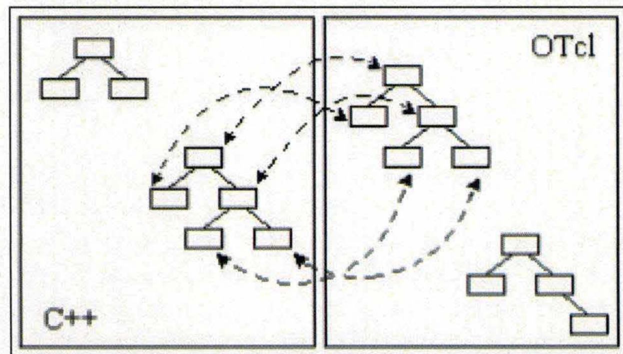


Figure III.11: C++ and OTcl: The duality. Taken from [1]

The scheduler, one of the most important functionalities in NS2, is the core of the simulation. It provides functionalities for handling events and assures that the correct code will be executed at the right time. The scheduler class offers several features, but one of the most useful is the one providing a way to schedule self implemented events. There are three main types of schedulers: *calendar*, *heap* and *list*, which differ only in the manner how they are implemented. There is also a real-time scheduler which uses the list implementation type. The real-time scheduler provides synchronisation of the simulation time with real time events. This can be very useful when NS2 is used as a component in a larger environment where not all parts are simulated but may instead be real world equipment connected to the Internet for example.

The packet class is a sub-class of the event class. The classes are therefore in many ways similar and in fact almost all events are packets being sent and received. Each event can be scheduled individually. When an event is ready for dispatch, the handler is called. The handler executes its code and takes appropriate action before passing the event i.e. packet on. The description of the packet format is illustrated in *Figure III.3*

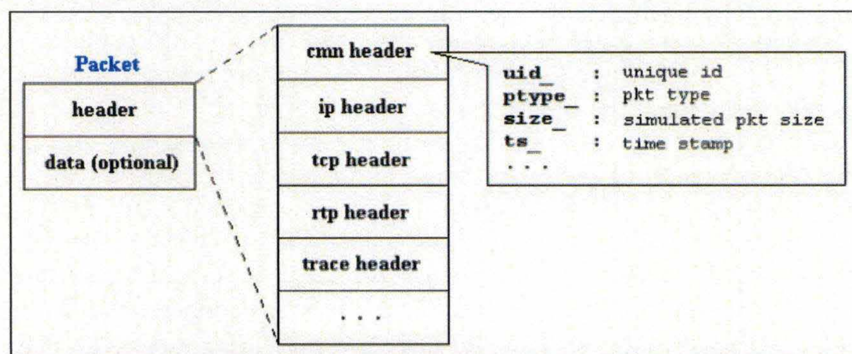


Figure III.12: NS Packet Format. Taken from [1]

All the headers that are available in NS2 are included in the packet representation. This is in contrast to real IP packets where only the headers that are used are present. To compensate for this, the size of the packet is not at all dependent on these headers, instead the size of the packet is set in a special field in the *cmn* part of the header. The data fields in the packet are not used at all most of the time. This means that only headers are passed around in the network without actually carrying any data. Since NS2 is only a simulator this is acceptable, and everything that depends on the size of the packets, uses the *cmn*

header field.

Topology

The first step in a simulation is to set up the topology, composed of nodes and links. The links interconnect the nodes and one node can have multiple incoming and outgoing links. To determine what incoming packet should be sent out to what outgoing link an internal mechanism of "routing" is used. The mechanisms are the classifiers who have the capacity to determine what to do with an incoming packet. Each object that ever handles a packet has in common the way that the packet is received. These objects receive not only the packet but also a link handler. The handler is a pointer to the object that will execute if the receiver chooses to call it. The handler can be, used to send information in the opposite direction, i.e. the direction where the packet is coming from. While the link delay introduces a delay corresponding to the transfer time over the link, the queue simply queues incoming packets before sending them on the link. The Time-To-Live (TTL) module is responsible for calculating the new TTL value for every packet. All packets must pass these modules before being passed on. The queuing of packets is normally done in the incoming stream of a router. In NS2 this is done in the link instead. The main disadvantage of the strategy used in NS2 is that if a node has several incoming queues the node has no way of doing advanced queue management taking into account the characteristics of all queues. The reason for this is simply that the different queues are located in different links. In our case, this will not cause any problem because each node has only one incoming "two-way" link.

Agents

To insert traffic into a simulation NS2 uses the agents. There are many types of agents but they have one thing in common: they all operate at an end-point of the network, i.e. they work as terminals. Furthermore, they have the responsibility of "constructing" and "destroying" packets. There are several agents supporting various protocols implemented in NS2. One of those agents is the TCP agent. It sends out TCP traffic and actually is only the sender side implementation of TCP. For receiving TCP traffic and responding in a correct way there is a need for another agent handling this. This particular agent is called the TCP Sink. This division of TCP into one sender and one receiver is really a simplification. Later functionalities were added to have both sides act as both sender and receiver ([3], page 266). This new agent is called Full TCP but it is still in an experimental phase. TCP may be the dominant in the number of variants supported but there is also support for UDP as well as RTP. The TCP and UDP agents do not produce data. Instead a traffic generator needs to be connected to the agents. The least complicated one is the CBR (constant bit rate), but there are many others as well. FTP is implemented as a traffic generator and when connected to TCP it can be configured to send a specific amount of data to another agent in the topology.

Trace-files

There is built-in support in NS2 for tracing the data flow generated during the simulations. The tracing is done by writing data, describing various events, to a text file. Since the simulator is used to simulate

networks, most things involving packets are written in this particular file. If the packet is associated with a specific protocol, e.g., TCP, data from the protocol header can also be traced. In *Figure III.4* we can see an example of a trace file. The leftmost column indicates what type of event occurred; the next column is the time when that event occurred. After this there are two columns containing information about the source and destination addresses of the transmitted packet. The next thing to come, still going left to right, is the type of packet followed by the packet size. The most common symbols in the leftmost column are the r, + and -. The r symbol represents the event that a packet is received at a node, while + and – symbols are related to queuing. The + means that a packet was inserted in a queue while – means that a packet was taken from the queue.

event	time	from node	to node	pkt type	pkt size	flags	fid	src addr	dst addr	seq num	pkt id

```
r : receive (at to_node)
+ : enqueue (at queue)                src_addr : node.port (3.0)
- : dequeue (at queue)                dst_addr : node.port (0.0)
d : drop    (at queue)

        r 1.3556 3 2 ack 40 ------- 1 3.0 0.0 15 201
        + 1.3556 2 0 ack 40 ------- 1 3.0 0.0 15 201
        - 1.3556 2 0 ack 40 ------- 1 3.0 0.0 15 201
        r 1.35576 0 2 tcp 1000 ------- 1 0.0 3.0 29 199
        + 1.35576 2 3 tcp 1000 ------- 1 0.0 3.0 29 199
        d 1.35576 2 3 tcp 1000 ------- 1 0.0 3.0 29 199
        + 1.356 1 2 cbr 1000 ------- 2 1.0 3.1 157 207
        - 1.356 1 2 cbr 1000 ------- 2 1.0 3.1 157 207
```

Figure III.13: NS2 Trace Format Example. Taken from [1]

III.B Improvements to the "Strathclyde University" NS2 UMTS module

Introduction

The purpose of this section III.B is to explain the major modifications of the UMTS extension for Network Simulator, NS2 developed by Pablo Martin and Paula Ballester [4] at Strathclyde University, United Kingdom, during the academic year 2002-2003. It is worth mentioning that this module has been developed with the aim at offering an experimental platform for routing protocol developments. Moreover, it only supports UDP. As we were studying the performance of various TCP protocol versions over a UMTS network, we have experienced its inability to provide a support to this transport protocol as a major limitation. This section hence documents the improvements which have been realised in order to make this extension suited to our simulation requirements.

First of all, we will start by stressing the differences between the two transport protocols TCP and UDP, as they influence the functioning of the UMTS extension. Following this, we will then discuss fixes to the inadequate behaviours of the extension in specific scenarios. Afterwards, we will address the changes to the module, aimed at making it compliant with TCP. Finally, improvements motivated by our own investigations will be presented in the last section.

In the remaining part of this chapter, specific names are given to the data units in the application, transport and RLC layers, "flow", "segment" and "packet" respectively.

Differences between TCP and UDP w.r.t. SF calculation

As a foreword, it is important to keep in mind the differences between the sending behaviour of an application relying either on TCP or on UDP, as they may explain "surprising" results when using the UMTS extension. Indeed, "surprising" results obtained with TCP may be mistakenly regarded as bugs, whereas they are mainly due to the module design.

TCP aims at providing reliability to the application layer by trying to evaluate constantly the congestion status between the sender and the receiver. On the ground of this evaluation, the sending rate of the transport layer is adapted. On the other hand, as UDP has no way of evaluating the available resources of the network, the amount of data being sent at any given point in time with UDP depends merely on the application layer. Consequently, while it is possible to define a specific sending pattern for a UDP data flow from the application layer, this cannot be achieved likewise for a TCP data flow. This can lead to unexpected consequences in UMTS environments.

A UMTS network has indeed to provide BoD, while performing resource sharing among all the users within a cell. This means that every user sending data over the air interface is allocated a fixed bandwidth for an amount of time. In the extension developed by Pablo Martin and Paula Ballester, bandwidth allocation is based on the parameters of the Tcl script that sets up the simulation. As an example:

```
$node_(0) 2222 ftp rate_ 140.0k
```

A dedicated bandwidth of 140 kbps is allocated to flow #2222 from node_(0).

Bandwidth allocation falls within the responsibility of the RRC layer. How can this layer evaluate and therefore allocate the right bandwidth for every incoming flow? Of course, several flows with different requirements can be set up at each node. Ensuing this, the RRC layer will take into account the sum of these requirements to open a dedicated channel between the UE and the NodeB providing the minimum bandwidth fitting the requirements of the outstanding flows. More precisely, the bandwidth provided by the dedicated channel will depend on the SF computed by the RRC layer on the ground of the sum of the flow requirements. But, in most cases, the computed SF will open a dedicated channel providing more bandwidth than the one required by the flows. In UDP scenarios, this does not have any impact, as the sending pattern of UDP only depends on the application layer, and not on the available bandwidth! On the other hand, the TCP protocol, by way of mechanisms like Slow-Start and Congestion-Avoidance, always tries to reach the upper boundary of the available resources. As a result, the experienced bandwidth can, "surprisingly", be really wider than the one declared in the Tcl script.

Differences w.r.t. bandwidth allocation between standard and implementation

Let's have a look the available spreading factors in uplink as well as in downlink.

DPDCH spreading factor	*DPDCH channel bit rate (kbps)*	*Maximum user data rate with ½ rate coding (approx.)*
256	*15*	*7.5 kbps*
128	*30*	*15 kbps*
64	*60*	*30 kbps*
32	*120*	*60 kbps*
16	*240*	*120 kbps*
8	*480*	*240 kbps*
4	*960*	*480 kbps*

Table III.5: Uplink DPDCH data rates.

The above table depicts the capacity of the dedicated channel that can be provided by the network in uplink. For instance, let's take the Tcl line given above. If node_(0) was a UE, the required bandwidth would have been 140 kbps. This would have lead to SF = 8 providing a potential bandwidth of 240 kbps. With the TCP protocol, the effective bandwidth used by the flow 2222 could indeed reach 240 kbps. Despite the lower bandwidth specified in the Tcl script, this is the one given by the right column of the above table which will be achieved.

DPDCH spreading factor	*DPDCH channel bit rate (kbps)*	*Maximum user data rate with ½ rate coding (approx.)*
512	*15*	*7.5 kbps*
256	*30*	*15 kbps*
128	*60*	*30 kbps*
64	*120*	*60 kbps*
32	*240*	*120 kbps*
16	*480*	*240 kbps*
8	*960*	*480 kbps*
4	*1920*	*960 kbps*

Table III.6: Downlink DPDCH data rates.

It is worth mentioning that the capacity of the dedicated downlink channels provided by the module does not follow completely the UMTS standard. In the downlink, control and data information are time multiplexed, whereas they are sent through separate channels in the uplink. The UMTS extension use

separate channels for both down- and uplink. As a result, the bandwidth achieved in the downlink side can exceed the value of the standard, as shown in *Table III.2*. The maximum user data rates do not correspond with the values given in [5, p. 96], but rather with the uplink values given in *Table III.1*.

First set of modifications

This first set of modifications was performed to enhance the general working of the module. It mostly provides bug fixes.

- ***Modification #1: Spreading factors***

Problem statement: The original module allowed to set a spreading factor of 512 for an uplink dedicated channel. This is not possible according to the standard and has been corrected in this version.

Problem resolution: SF = 512 no longer available in uplink.

Part of the code modified: LLUE::get_sf

- ***Modification #2: Handling packet loss at RLC layer – Flush receive buffer***

Problem statement: As explained in the UMTS standard, the RLC layer is the one responsible for the enhancement of the transmission reliability. Its way to proceed is first to split every incoming segment (TCP or UDP), belonging to one flow, in smaller RLC packets before handing them down to the lower layers for transmission. The behaviour of the RLC layer handling this data flow on the receiver side depends on the requirements of the flow. Two kinds of quality of service can be provided: with or without LL recovery. From the RLC layer point of view, a transmission error corresponds to a packet loss, even if a single bit has been corrupted. If the LL has to provide reliability, every packet reported as missing triggers a retransmission request. Meanwhile, all received RLC packets are stored in a buffer. Reassembly is performed when all RLC packets belonging to one segment (TCP or UDP) are correctly transmitted or retransmitted. Without LL recovery, all the RLC packets following the loss are discarded, until the first RLC packet of the next transferred segment is received. Reassembly can therefore only be performed if all the information of a segment has been correctly transmitted.

The receive buffer was handled erroneously by the RLC layer, as it was possible to get several instances of the same RLC packet in the buffer when the transmission error rate was reaching high values. In a context of high transmission error rates, the number of retransmission requests from the receiver is getting high. Thus the average time between the moment the repeat request is sent and the moment the RLC packet is finally correctly received can become significant. In this situation, the RLC layer may time out, assuming that the packet has been lost again, and will apply for its retransmission. This leads to the appearance of several instances of the same RLC packet into the receive buffer of the RLC layer. When the RLC packets are reassembled, only one of those instances is used, the other ones remaining in the buffer due to the absence of a clearing mechanism.

In normal conditions, due to the fact that the module reassembles the packets upon arrival, the buffer

would be non empty only when a packet is missing for reassembly. This missing packet is assumed to be the one preceding the packet with the smallest sequence number in the buffer, and that sequence number is mentioned in the acknowledgement of the last received packet.

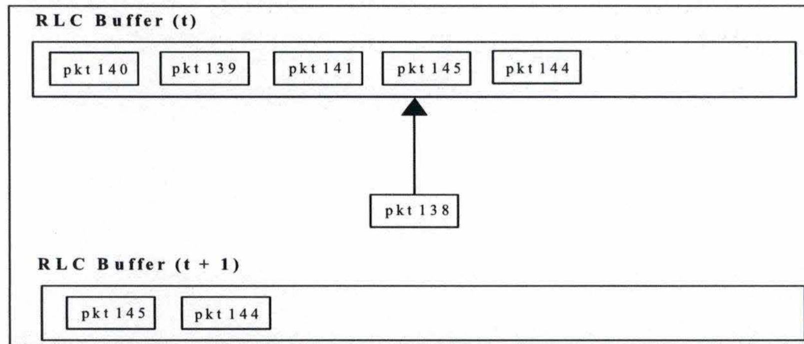


Figure III.14: RLC Buffer Status.

Figure III.5 depicts the case where packet 138 is missing. All the RLC packets preceding this packet have already been reassembled and therefore removed from the buffer. In order to carry on the reassembly, packet 138 is expected, and requested through acknowledgements. Upon its arrival, the reassembly process is carried on up to the next missing packet, being packet 143 in this specific case.

However, with uncleared instances of previously reassembled packets, if the packet with the smallest sequence number is actually a replicate of a packet *n* already correctly received and sent to reassembly, the presence of duplicate(s) triggers the transmission of erroneous acknowledgements requesting the packet *n*-1 already received.

Problem resolution: Every duplication of a RLC packet in the receive buffer is now removed whenever reassembly is performed. If received in sequence, packets are immediately reassembled and never stored into the RLC buffer for the sake of efficiency.

It is important to keep in mind that it would have been better to try to adapt constantly the rate of the wireless channel to the extra data coming from retransmission requests during all the lifetime of the connection, such that the bandwidth experienced by the application layer would have been the actual required bandwidth. But this would have required updating the SF during the simulation. This feature is not supported yet. Another option would have been to give the priority to the transmission made on the ground of repeat requests. Unfortunately, this meant modifying important parts of the code.

Part of the code modified: RlcUmts::sendUpDATA
RlcUmtsNodeb::sendUpDATA

- ***Modification #3: Handling packet loss at RLC layer – Correct reception check***

Problem statement: When no link layer reliability is provided, all the RLC packets following a transmission error are discarded, until the first RLC packet of a new segment is received. Consequently, as a RLC packet is missing, the reassembly of the RLC packets into the transmitted segment will no longer be possible. In NS2, simulations are usually done with dummy data. As a result, if parts of the information to be transmitted are lost on the way, the receiver can still manage to reassemble a segment

and to forward it to the higher layer! Actually, this was achieved by the original version of the UMTS module, as it did not check whether all the RLC packets of a given segment had been correctly transmitted when receiving the last RLC packet of that segment. Therefore, in scenarios where the LL recovery was not activated, the higher layers (RRC, transport and application) were detecting a transmission error (missing segment) only if it affected the last RLC packet of a segment. The impact of transmission errors was therefore considerably reduced.

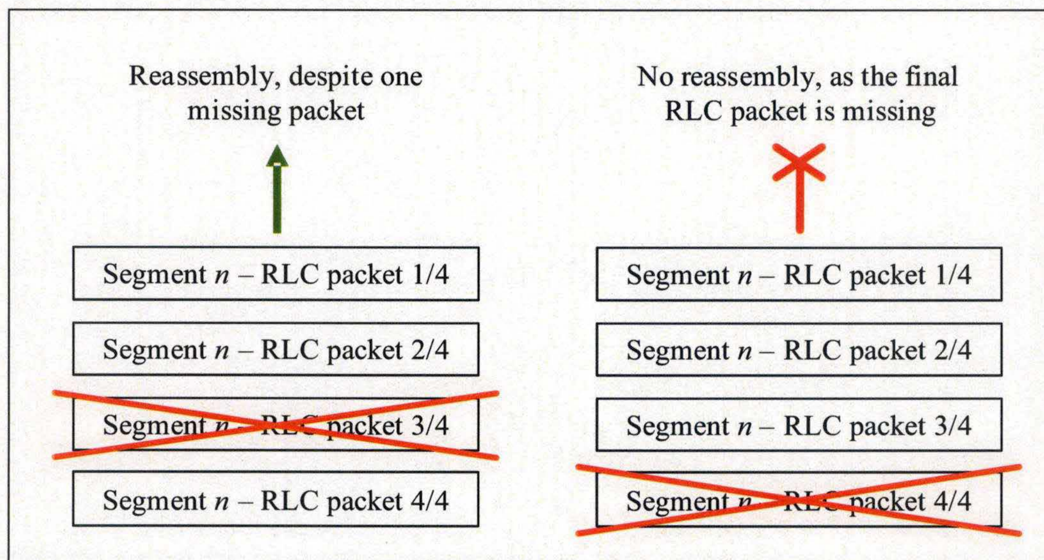


Figure III.15: Erroneous RLC packet handling.

Problem resolution: Every transmitted RLC packet is now checked for correct reception.

Part of the code modified: RlcUmts::sendUpDATA
RlcUmtsNodeb::sendUpDATA

- ***Modification #4: Timeouts at the RLC layer***

Problem statement: The RLC layer is in charge of link layer recovery. Thanks to packet numbering, the RLC layer can notice when a RLC packet is missing. As the UMTS network provides an in-order delivery of the RLC segments, if the packet received is not the one expected, it means that a loss did occur during the transmission. Moreover, to provide reliability, the layer has also to pay attention to flow interruptions. But an interruption in the transmission flow can have two opposite meanings. In the positive case, the flow is interrupted because the flow has been successfully transmitted. In the negative case, the interruption of the flow is due to bursts of errors. Since a timer is set every time a new RLC packet is received, the RLC layer times out, either when a RLC packet has been lost, or when the transmission has been completed. This way of providing link layer reliability has the advantage of being quite simple. Unfortunately, it has the drawback that the RLC layer times out each time the transmission of a data flow is completed. For instance, in case of a simulation with a number of short transmissions over the UMTS connection (telnet, ssh for instance), each of these transmissions will end with a loop of repeat requests from the receiver.

Problem resolution: It has been chosen to avoid restarting the timer when the last RLC packet of a transport segment is fully received. Of course, this has got some impact on the LL reliability level. More precisely, in case a transport segment completely fits into a single RLC packet, and if it is not followed by another segment, the loss of this packet will not trigger any retransmission request. This is exactly what happens when a SYN segment (TCP connection opening segment) gets lost.

Part of the code modified: RlcUmts::sendACK
RlcUmtsNodeb::sendACK

- ***Modification #5: RLC feedback***

Problem statement: RLC feedbacks to the sender provide a way for the sender to know the next packet expected at the receiver side. In the original extension, feedback was generated on the ground of the last RLC packet received if the buffer is empty, or the RLC packet with the smallest sequence number otherwise. It means that if the received packet was the one expected, the feedback will mention the RLC packet just following this last one, meaning that the packet has been correctly received and the receiver is waiting for the next packet. Otherwise, feedback will mention the latest RLC packet delivered in order plus 1. Unfortunately, this system has got some limitations when the reception of one missing RLC packet enables to reassemble several transport segments waiting for that packet arrival to be reassembled, and when that reassembly empties the receive buffer. This happens for instance when the reassembly of TCP segment *n* is delayed, as one of its RLC packet has not been correctly received, whereas the following TCP segment (*n+1*) has already been correctly received. In such a case, when receiving the missing RLC packet, the original extension was only acknowledging the reception of the TCP segment that RLC packet belongs to, *n* in the example, but not (*n+1*) if there are no further RLC packets stored in the buffer beyond segment (*n+1*). This bug triggered unnecessary retransmissions.

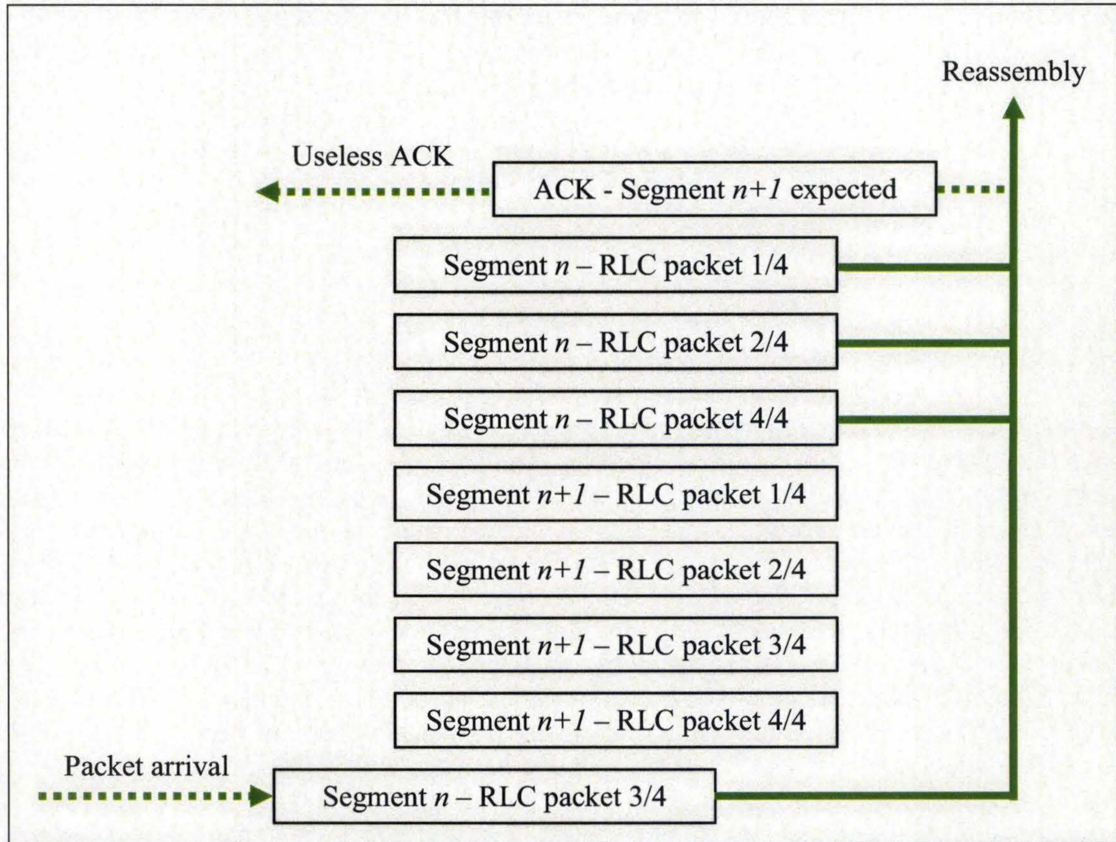


Figure III.16: Erroneous RLC packet handling.

Problem resolution: The acknowledgement is systematically based upon the last RLC packet of the last reassembled transport segment.

Part of the code modified: RlcUmts::sendUpDATA
RlcUmtsNodeb::sendUpDATA

- ***Modification #6: Synchronisation at PHY layer***

Problem statement: The UMTS standard specifies that the CCTrCH can rely on at least two dedicated physical channels, one for the control information, the DPCCH, and at least one for the data information, the DPDCH. To decode the whole information, the receiver needs to get the control information from the DPCCH as well as the data information carried by each DPDCH. If the data carried by a single dedicated channel is missing at the end of one slot, the entire data can not be decoded and has to be dropped.

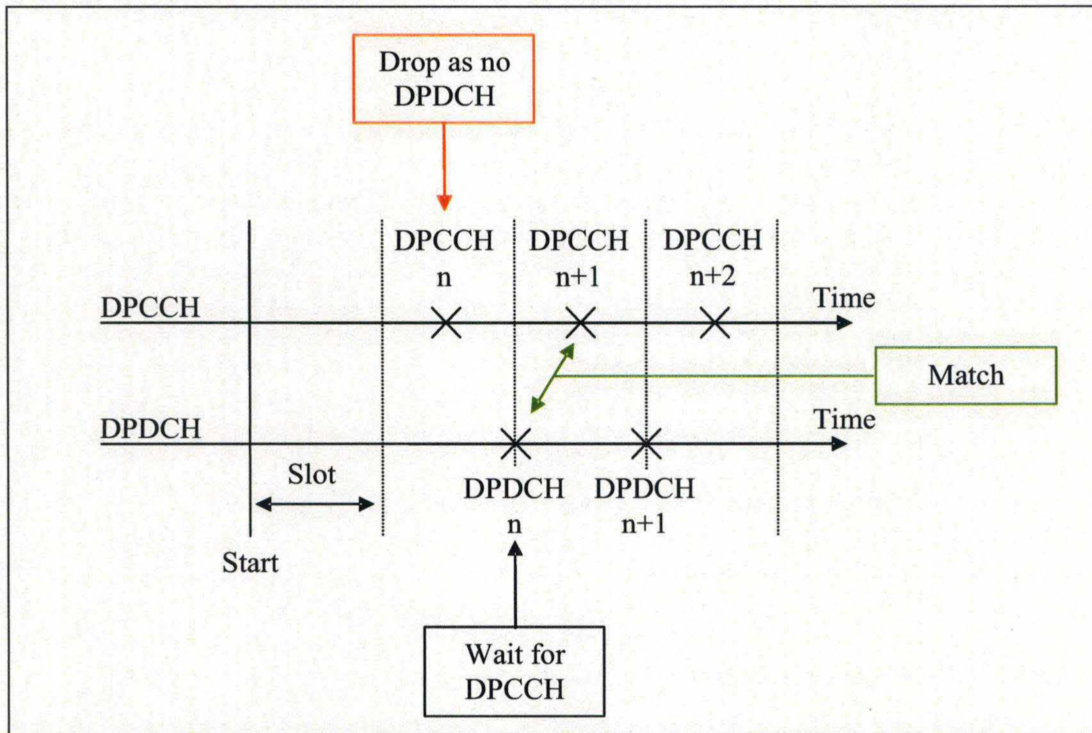


Figure III.17: Dedicated channel synchronisation.

In the original extension, the reception check of each dedicated channel was not performed at the end of every slot but instead at the reception of the control information. Unfortunately, the DPDCH and the DPCCH had different transmission delays. The transmit time of the control information was shorter than a slot. Therefore, it was not possible to send anything at all over a dedicated channel. Actually, when the DPCCH arrival was triggering the reception check of the DPDCH, the latter was missing, and only arrived afterwards. The useless control information was then dropped. Consequently, at the next reception check triggered by the arrival of a new DPCCH, the previous DPDCH was used for that check. In other words, the data information received during the slot *n* was matched with the control information received at slot *(n+1)*. As a matter of fact, a(n erroneous) match could be performed during all slots but two: the first slot, because no data information had been sent yet, and the last slot because control information was no longer available! As a consequence, opening a TCP connection was hardly possible, due to the difficulties to send the first SYN packet. On the other hand, all the connections finished in a loop of repeat requests because it was never possible to send the last RLC packet.

As shown on *Figure III.8*, the arrival of the control information precedes the one of the data. This makes it impossible to decode the entire information. Also, during the last slot, when the data arrives, no control information is still present at the receiver side, thus the data will never be decoded.

Problem resolution: Re-synchronize DPCCH and DPDCH. Moreover, control as well as data will take the entire time slot to be sent.

Part of the code modified: PhyUmts::TX_Time
PhyUmtsNodeb::TX_Time

- ***Modification #7: Mobility***

Problem statement: During the lifetime of a connection, the protocol stack of a UE and the one of a NodeB store information in state variables. For example, the RLC layer keeps track of the sequence number of the next expected RLC packet for each flow, and the UE PHY layer has to remember the address of the Node B it is connected to. When handovers occur, all this information related to the previously attached NodeB is of course removed. Unfortunately, in the original version, the state information stored inside the RLC layer was not removed in such a case. In other words, all the information was effectively removed but the one inside the RLC layer. As a consequence, it was not possible to return to the previous NodeB and to keep on sending data of a flow applying for LL reliability, as the state information was still referring to a previous part of the flow (for example the next expected packet just before the handover).

Problem resolution: RLC status information is now refreshed just after a handover.

Part of the code modified: RlcUmtsNodeb::sendUpDATA

- ***Modification #8: Interference handling at NodeB***

Problem statement: Although it has been possible to decode the data after the transmission (CCTrCH) the simulator has still to define whether the interference threshold was not too high and thus whether the data was not corrupted. For this reason, the PHY layer sends to the MAC layer an array holding the BLER value computed on the ground of the interference level for each connected UE. Unfortunately the MAC layer did not look at the right place in this array. In most situations, erroneous values were taken into account.

Problem resolution: The MAC layer is now looking for the information at the right place.

Problem statement: PhyUmtsNodeb::blerHandler

Second set of modifications

The second set of modifications aims at making the module compatible with the TCP protocol.

- ***Modification #1: Ambiguity in TCP and RLC headers***

Problem statement: One of the main reasons of the incompatibility of the original extension with TCP was due to ambiguities between TCP and RLC layers. Each packet in NS2 gets a basic header called "*cmn* header" (*Figure III.3*). One field of this header is used to give a general description of the packet. The ambiguity came from the fact that TCP and RLC acknowledgement packets used exactly the same description. As a result, a conflict appeared every time a UE or a NodeB received a TCP acknowledgement packet. It was erroneously interpreted as a RLC acknowledgement packet by the RLC layer.

Problem resolution: Avoid using this specific field of the "*cmn* header" in the identification of each incoming packet and to use rlctype_ field belonging to the RLC packet header extension.

Part of the code modified: RlcUmts::recv
RlcUmtsNodeb::recv

- ***Modification #2: Computation of the spreading factors***

Problem statement: Each time the RLC layer is not working in TR (see chapter II, section C), every segment coming from the above layer is split up into smaller RLC packets. In the way the original module had been implemented, one RLC packet could not carry more than one TCP segment. It does imply that if a TCP segment or the last part of a TCP segment (according to the segment's split up) cannot completely fill a RLC packet, the remaining space is wasted. This is the main reason why the segment sizes as well as the bandwidth requirements were taken into account to compute the spreading factor in the original extension. Unfortunately, this computing was made on the ground of the first incoming segment. With UDP, this has no impact as changing the size of the segment during the connection is not possible within NS2. However, things are more complicated with TCP. The first TCP segment sent over the connection is usually a SYN segment dealing with the connection opening. The size of this segment is not related to the size of the following segments belonging to the connection. For most scenarios, this problem lead to the computation of a spreading factor completely wrong. The more bandwidth the application required, the more significant the impact of this miscomputation was.

Problem resolution: Avoid taking into account the segment size in the calculation of the spreading factor. The spreading factor computation is performed on the sole ground of the bandwidth requirements set by the Tcl script (TCP bitrate).

Part of the code modified: LLUE::get_rate
LLNodeb::phy_rate

- ***Modification #3: Opening/closure of dedicated channels***

Problem statement: According to the UMTS specification, downlink and uplink dedicated channels have to be opened and closed simultaneously. The original module did not apply that rule. The system allowed handling both channels (uplink and downlink) with more freedom.

The opening of a channel is initiated by an incoming segment and the closing phase is triggered by a too long idle time (no information sent over the channel). Therefore, a channel can be open in down-/up-link without up-/down-link counterpart, and the same thing can happen at closure. It is even possible to open one channel (for example the downlink) without the other (for example the uplink) if the latter is not used. From a simulation point of view this does not have much impact but for TCP, when one channel (for example the uplink) is closed suddenly (in the middle of the connection), but the other remains open. This situation may occur in link layer recovery scenarios, when the transmission of a packet is delayed because of retransmission, implying an interruption of the TCP ACK flow in the opposite way. A too long interruption can lead to the closing of the return channel whereas the other one remains open.

Problem resolution: Take into account the incoming of one segment for the computation of the idle time in both ways (uplink and downlink). Whenever a segment arrives in the downlink, both down- and uplink timers are reset.

Part of the code modified: LLUE::recv
LLNodeb::recv

Third set of modifications

This final set of modifications was performed in order to make the module more suited to our needs. These modifications were aimed at creating the desired simulation contexts, by simplifying the way the configuration has to been set up.

- ***Modification #1: BLER setup***

Problem statement: The module decides if a packet is corrupted based upon the noise and the interference level inside the cell. If several users or several base stations close to a specific user are emitting simultaneously, the user will very likely have to cope with an increase of transmission errors. For each user, the module calculates the Eb/N0 ratio (Bit energy on noise and interference spectral density). This ratio will be used to index an array in order to retrieve the corresponding BLER value. In order to evaluate the efficiency of a transport protocol like TCP, it is imperative to create a noisy environment. Setting up the simulation environment in order to be able to evaluate the transport protocol in noisy conditions is not an easy task. The requested interference level can indeed not be set directly, but has to be reached by indirect means. Several interfering sources have indeed to be dropped in the cell in order to produce the requested noise level. Even if adding such sources inside a cell is rather easy, it remains however difficult to predict precisely the level of interference that will be achieved.

Problem resolution: The modification enables to set the BLER value directly from the Tcl script. This value can either be constant or randomly distributed according to a specific distribution between two given limits.

Instructions added: set setbler_ x
set bler_ y

The first instruction takes two values: x is 0 or 1, depending whether the BLER value is set manually (0) or not (1). The second instruction is meant to set the BLER value (y is the given BLER value, described as a constant or as a random function). This is valid for one single frame, and can thus be modified between frames. These instructions have to be applied on the physical layer.

- ***Modification #2: RRC layer timeout***

Problem statement: The timer on the RRC layer is used to close a dedicated channel when there is no activity during a specific time frame. The problem of this specific layer is that RLC retransmission

traffic is not considered as actual RRC traffic, since no information goes through this layer. As a consequence, for simulations with high error levels, and therefore many RLC retransmissions, the RRC layer was eager to cut the connection even when there were a lot of activity between two nodes.

Problem resolution: The timeout value is a parameter of the simulation.

Instructions added: set ctimer_ x

This instruction makes it possible, on the link layer, to give a specific value (x) to the timer.

Conclusion

The modifications to the code of the UMTS extension proposed by Pablo Martin and Paula Ballester have been presented. They aim at implementing correctly the UMTS standard, fixing bugs and facilitating the simulation environment setup in order to get accurate results. The extension can therefore be used now, not only for routing purposes, but also for studying the efficiency of a transport protocol like TCP in a given context.

III.D Sources

[1] Clung J., Claypool M., **NS by example**.
Available at http://nile.wpi.edu/NS/, Last visited: September 29th, 2003.

[2] *Hynderick. O, Raes. S,* ***UMTS extension for NS2.***
Available at http://www.info.fundp.ac.be/~lsc/Research/ns-2_UMTS/, Last visited: September 28th, 2003.

[3] **The NS2 manual**, September 2003.
Available at http://www.isi.edu/nsnam/ns/doc/ns_doc.pdf, Last visited: September 29th, 2003.

[4] *Martin P., Ballester P.,* ***UMTS extensions for NS2****.*
Available at http://www.geocities.com/opahostil/, Last visited: September 28th, 2003.

[5] Holma H., Toskala A., **WCDMA for UMTS – Radio Access For Third Generation Mobile Communications**, Second Edition, Wiley, ISBN 0-470-84467-1.

Chapter IV: Transport Control Protocol Basics

This chapter deals with basic features of TCP and their implementation. First of all we will discuss some of its general features and how it became one of the most reliable protocols used over the Internet. Its robustness and ability of fair sharing of the available network resources makes it ideal for large networks such as the Internet. Section IV.B, introduces the mechanisms TCP can rely on, in order to reach an equilibrium during transmission. Once the equilibrium reached, the way TCP handles its conservation will be explained in section IV.C.

IV.A General TCP features

TCP is the most widely used protocol when speaking about the data flowing over the internet. With traffic distributions of the internet averaging 80% for data flows and even more than 90% for bytes, it's clear that TCP is a well established protocol [3], which deserves closer attention. We will illustrate some of this dominant protocol's specific features:

- Connection oriented :

The Transmission Control Protocol, TCP, is a connection-oriented transport protocol that provides reliable data delivery. A connection oriented transport protocol means that any data transfer is only possible once a connection between the two communicating entities is established. These entities are often referred to as the client and the server. The client is usually the one initiating the connection establishment. This procedure is called the three way handshake. The client emits a SYN segment specifying the IP address and the Port Number of the server which it tries to connect on. The initial sequence number is set thanks to the TCP Clock. The server will respond with its own SYN segment containing its own initial sequence number set the same way. This segment will also acknowledge the previous SYN, from the client, by adding 1 to the client's initial sequence number. The connection between the client and the server is considered opened once this packet has been received by the client. As for the server to client connection, it will be considered opened from the moment the server receives the acknowledgement for his SYN packet. Only when this procedure is terminated, we have established a two-way connection. (Shown in *Figure VI.1*).

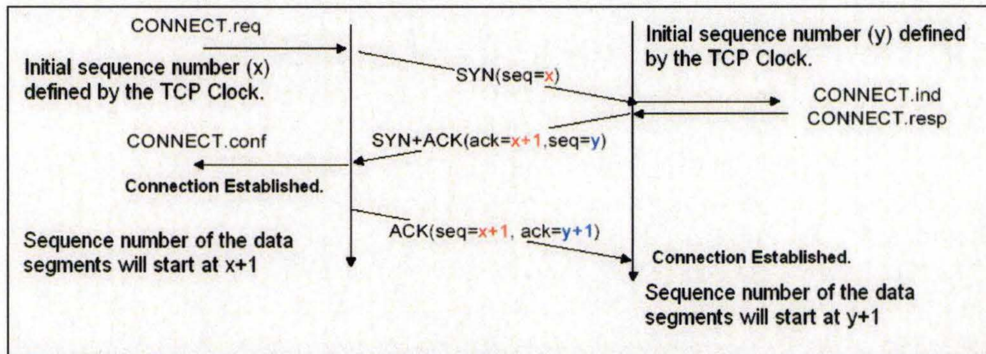


Figure IV.18: TCP Three-Way Handshake. Adapted from [1]

In a similar way TCP has to close the connection. To do so, it closes one side at a time by sending a FIN segment. (Shown in the *Figure IV.2*).

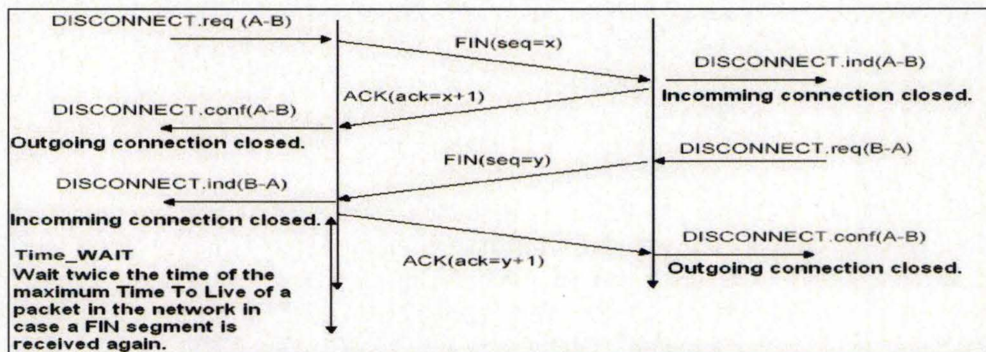


Figure IV.19: TCP Connection Termination. Adapted from [1]

After receipt of a FIN segment, the receiving side knows it doesn't have to expect anymore data from the other side. The possibility for it to send data still remains until it sends its own FIN segment.

- Flow Control :

To avoid data losses TCP uses a flow control mechanism which prevents flooding the receiver with data that it can not handle. To achieve this flow control, TCP uses a sliding window mechanism. In order to increase performance, TCP tries to send several packets at a time. The receiver can acknowledge these frames, if received in sequence, with only one cumulative ACK. The main issue is to know how many packets to send at a time, i.e. the sending window size. This window is called the advertised window. Each sender will set the *Window Size* field, as seen in *Figure IV.4*, a non optional 16-bit field in the TCP header, to the free data capacity of its receiver buffers. Due to the limited size of the field, the window size is limited to a maximum size of 65,535. In case the receiver buffers are full, the window size will be set to zero and therefore notify the sender to pause its data sending. If space is freed in the buffers, a window update ACK is sent letting the sender know it can resume transmitting.

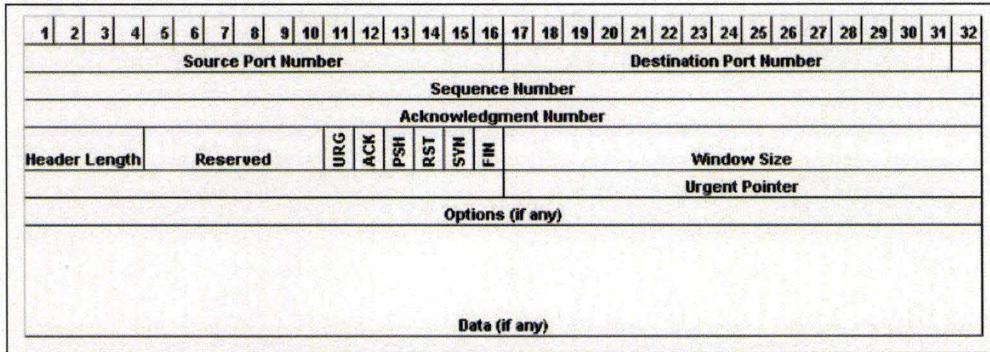


Figure IV.20: TCP Packet Format.

The advertised window informs us about the receiver buffer status. The Congestion Window (CWND) is a window used by the sender to probe the network capabilities of handling the traffic. In order to send data, TCP chooses the minimum of the CWND and the advertised window. By doing so, it tries not to flood the receiver's buffers and to avoid packet loss due to congestion. At the initialisation, the size of the CWND is usually set to one.

TCP also has a built-in congestion avoidance mechanism. When congestion occurs, meaning the total traffic offered exceeds the network capacities, packets will get dropped. TCP will try to recover by retransmitting. In these scenarios congestion is not solved and even gets worse. In order to avoid this vicious circle, TCP interprets packet loss as an indication of congestion. The transmission rate is slowed down every time packet loss is detected. When packets are delivered correctly, the transmission rate will be speeded up, within the limits of the sliding window.

- Reliable data transfer and congestion control :

TCP can recover from lost packets, duplicated packets, delayed packets, corrupted data and traffic congestion. To achieve this reliable data delivery, the TCP receiver sends acknowledgement control messages (ACKs) to the TCP sender to guarantee the successful receipt of the data. The ACKs are generally carried onboard other TCP packets. This method is known as "Piggybacking". However, even if the application on the sender side has nothing to transmit, it must transmit ACKs for each packet it receives. The ACKs can be sent either just after the packet has been received or when a data segment has to be sent. TCP has chosen a compromise. If a segment is received in sequence, and if it is the only one to be acknowledged, TCP waits for the possibility to piggyback or otherwise for a timeout. The timer is set, for example, to 50ms [1]. If there is a segment waiting for acknowledgement, the ACK is sent right away when the timer expires. In the case of a segment that is out of sequence the ACK is sent directly to permit Fast-Retransmit. The Fast-Retransmit mechanism will be explained in more detail in the V.B.2 section.

If a packet is lost on the way to the receiver, it will never be acknowledged and the sender has to be aware of that, in order to ensure reliability. TCP uses a Retransmission-Timer. Every time a data segment is sent, a timer is started. If the segment is acknowledged before the timer expires, the timer is stopped and reset. Otherwise, the segment is sent again when the timer elapses. Consequently, we can easily see why the choice of the timeout period is critical. If it is too high, the sender waits too long to receive acknowledgements and the throughput can be reduced drastically. If, on the contrary, it is set

too low, there will be too many retransmissions which increase traffic and thus the likelihood of congestion. This retransmission mechanism is based on the Retransmission-Time-Out (RTO). The value of this RTO derives from the RTT. The RTT measures the time between the instant a segment is sent and the instant of receipt of its acknowledgement. The value of the RTT can vary during the connection time and thus needs various adjustments. Additionally, the fact that each network can have different RTT values makes it obvious that a fixed RTT value will be impossible to give.

The following formulas are used for estimating the RTT and calculating the RTO [1], [2]:

RTO := Average (RTT) + 4 * Deviation (RTT)
Average (RTT) := (1-x) * Average (RTT) + x * Current RTT
Deviation (RTT) := (1-y) * Deviation (RTT) + y * |Current RTT – Average (RTT)|

The value of x is set to (0.125) and y to (0.25)

An upper boundary is set to avoid a too big value for the RTO, yet this limit should at least be 60 seconds [4]. The maximum RTO is mostly set to 64 seconds [6]. Also, there may be a minimum RTO value, corresponding to the precision of RTO calculation. For example, in the popular BSD implementation of TCP, the minimum RTO is set to 0.5 second [5]. Problems can also arise around the evaluation of the current RTT, especially after retransmissions. When an ACK arrives just after a retransmission, it's impossible to know whether the ACK is for the last retransmission or for the previous transmission. This case is shown on the Time Diagram on *Figure IV.4.* Not being sure, we can either underestimate or overestimate the RTT. The simplest solution to avoid these mistakes is not to calculate their RTT. This way of working is called Karn's Algorithm [7]. It simply states that the RTT is not updated on any segments that have been retransmitted and that the RTT measurements should only restart after an ACK is received for a segment that is not retransmitted. Also, the RTO should be doubled after retransmission. This is called Exponential-Back-Off (or binary exponential back-off). Why is that necessary? When the RTO is smaller than Real RTT, TCP will use the previous RTO forever. Thus all packets will be retransmitted and no new RTT measurements will ever happen again. But it is important to keep in mind that traditionally, one RTT sample has to be taken at a time [4], typically once per RTT. However, when using the timestamp option (explained in section V.B.4), each ACK can be used as an RTT sample. In [9] it is suggested that TCP connections utilizing large CWND should take many RTT samples per window of data to avoid aliasing effects in the estimated RTT. A TCP implementation MUST take at least one RTT measurement per RTT (unless that is not possible by Karn's algorithm). Either way, lost packets and their retransmission increases latency and the end-to-end delay. Thus, in order not to disturb the RTT calculation, data losses should be avoided.

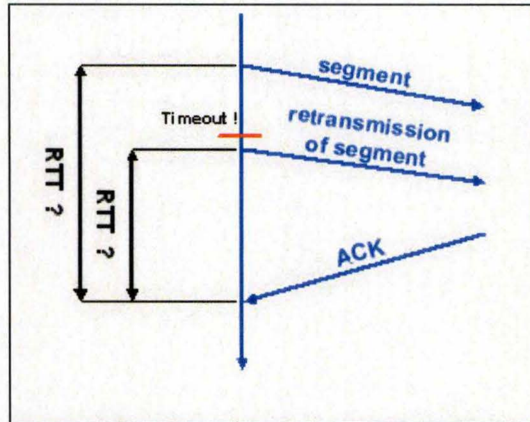


Figure IV.21: Ambiguous RTT estimation.

IV.B Reaching an equilibrium

Since its introduction TCP has undergone several modifications mainly in order to enhance its ability to face network congestion. In October of '86, the Internet had the first of what became a series or 'congestion collapses'. As an example, during this period, the data throughput from LBL (Lawrence Berkley Laboratory) to UC Berkeley dropped from 32 Kbps to 40 bps [10]. All of this shows how congestion control is imperative for the well-being of the network. By nature, congestion control mechanisms tend to prevent the network from being overwhelmed. This is achieved by limiting the number of packets in transit. More precisely, congestion control aims at reaching network equilibrium by finding local equilibrium at every connection. Each TCP connection has to limit its outstanding data in a way to provide a fairness sharing of the channel's bandwidth and avoiding network bottleneck [8] overflowing. At the time, congestion collapses were occurring, it was frequent to see 10% of the traffic dropped at internet gateways because of local buffer overflows [10]. To improve this situation, several new algorithms have been put into TCP. The most important are:

- RTT variance estimation
- Slow-Start
- Dynamic window sizing on congestion
- Fast-Retransmit

They spring from the observation that TCP flows should obey to the principle of 'packet conservation' [10]. For every connection 'in equilibrium', or with other words, running with a full window of data in transit, a new packet isn't put into the network before an old packet leaves. This view really shows what TCP aims for: flow stability as to the amounts of data in transit.

When equilibrium is reached, given the fact that the receiver can't generate ACKs faster than data can go through the network, the protocol is self-clocked. To start the clock, an algorithm, called Slow-Start, has been designed. The equilibrium is expected to be reached when the first hints of network congestion start to occur. This algorithm aims at a gradual increase of the amount of data in transit in order to reach the equilibrium. During this phase, at each ack received, 2 packets are generated. The size of the CWND is thus multiplied by 2 every RTT, leading to an exponential growth in time.

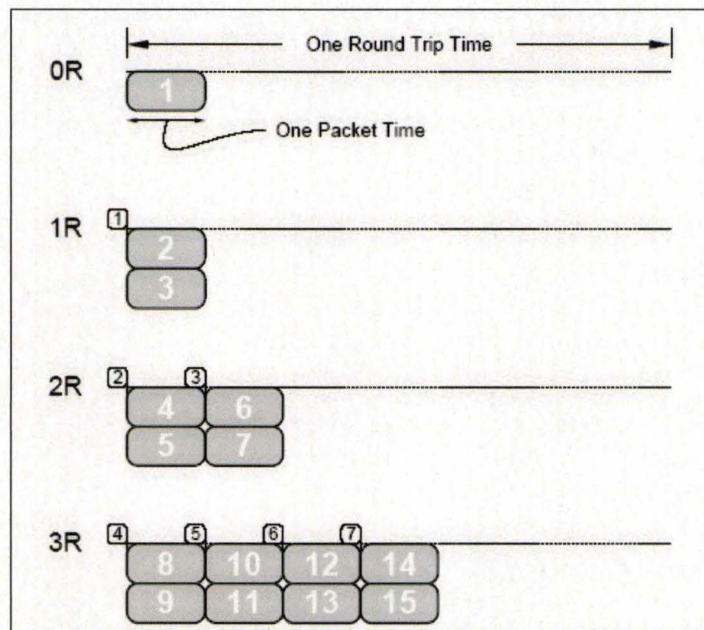


Figure IV.22: The slow-start phase. Taken from [10]

In *Figure IV.5,* the horizontal direction is time. The grey squares are the TCP packets and the white ones are the corresponding ACKs. From this figure, it is clear why an add-one-packet-to-window policy leads to an exponential increase in time. At one RTT, two consequent ACKs are separated by the time needed by one packet to go through the bottleneck, i.e. the slowest segment of the path. Accordingly, sending two packets on the ground of one ack will stack at least one of them on the top of the bottleneck's buffer (when the buffer is empty, both packets if not empty).

This leads to several observations:

- The Slow-Start algorithm probes the network by sending short packet bursts. However, sending packet bursts involves some network resources requirements in terms of buffer capacity. More precisely, opening a window of W packets requires a buffer size of W / 2 packets.

- The size of CWND tends to the BDP. Whether a minimum is required, bigger buffers also artificially affect the BDP. Indeed too many network buffer resources increase the amount of outstanding packets without improving the experienced throughput. An enlarged RTT will be experienced.

- The Slow-Start phase occurs during a time $R \log_2 W$, where R stands for the RTT and W, the CWND expressed in packets.

IV.C Conservation of the equilibrium

Once data is flowing reliably, the equilibrium can be maintained as long as no new packet is injected

before an old one has exited. Assuming that the ACK system is well implemented, the TCP protocol can only fail in keeping the equilibrium because of RTO failure. Hence, a reliable RTT estimator and a RTO taking into account the RTT deviation, are two of the major features for a congestion control mechanism geared at efficiency in the context of a heavy network load. For more details, it's probably worth looking at [11]. When RTO is working well, it can be stated that timeouts mean packet loss and not a broken timer. Packets can get lost because of two reasons: they are damaged in transit or the network is congested, meaning there was insufficient buffer capacity somewhere in the network. At the time this mechanism was designed, losses due to damage were assumed to be scarce (less than 1 %) [10]. Yet this is true, excepted for wireless networks. In fact, later on, research has focused on the way TCP faces congestion in order to keep the flow at equilibrium. The first "modern TCP version" comes up with Fast-Retransmit. Afterwards, mechanisms like Fast-Recovery or Selective-Repeat were included. More detailed explanations of those algorithms are provided in Chapter V, dealing with the various TCP versions.

Several remarks are to be made.

- So far, the equilibrium is present as a phase where the number of outstanding data is kept constant as long as no congestion occurs. In this case, TCP would be unable to adapt itself to an enhancement of network availabilities. Actually, TCP applies a CWND increase policy of one packet every RTT. Its adapting ability has become $\Delta W / RTT$ instead of $R \log_2 W$ during the Slow-Start phase. That underlines the fact that TCP tends to an optimal value of CWND rather than constantly reaching the best use of network resources. Accordingly, TCP may be rather ill-suited in highly variable environments.

- The throughput of any window flow control protocol is quite sensitive to damage losses. In TCP Tahoe, packet loss empties the window and so, a loss probability of P degrades the throughput by a factor of $(1 + 2PW)^{-1}$ [10]. Actually, this formula is not completely fair. Indeed the probability P is about packet (for example: one mistake every ten packets) and thus favours connections using a larger packet size. When the probability is given in term of bits, which is closer to the notion of Block Error Rate (BLER) we get: $(1 + 2PSW)^{-1}$ where S is the packet size in bits.

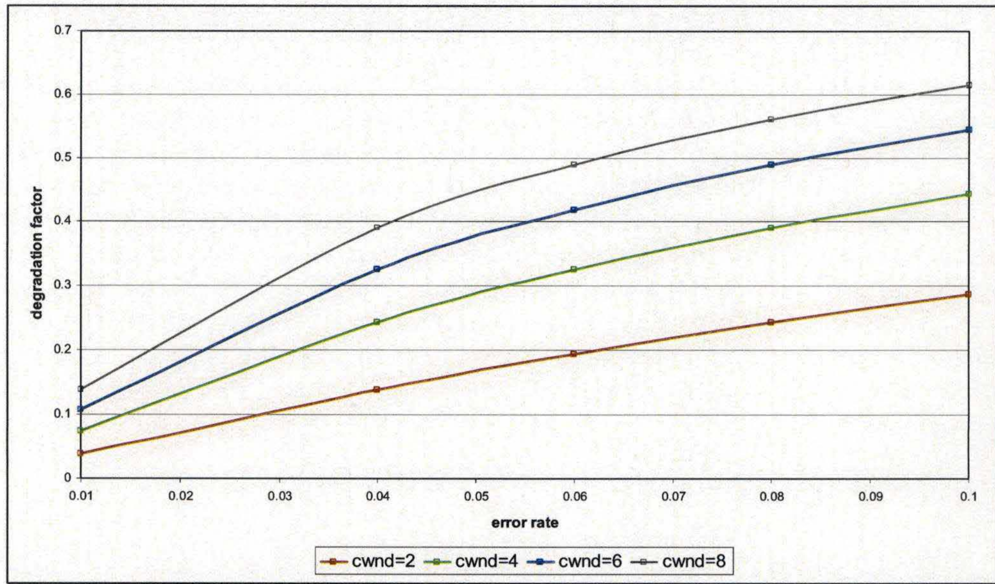


Figure IV.23: Impact of the error rate (packet level) on the throughput

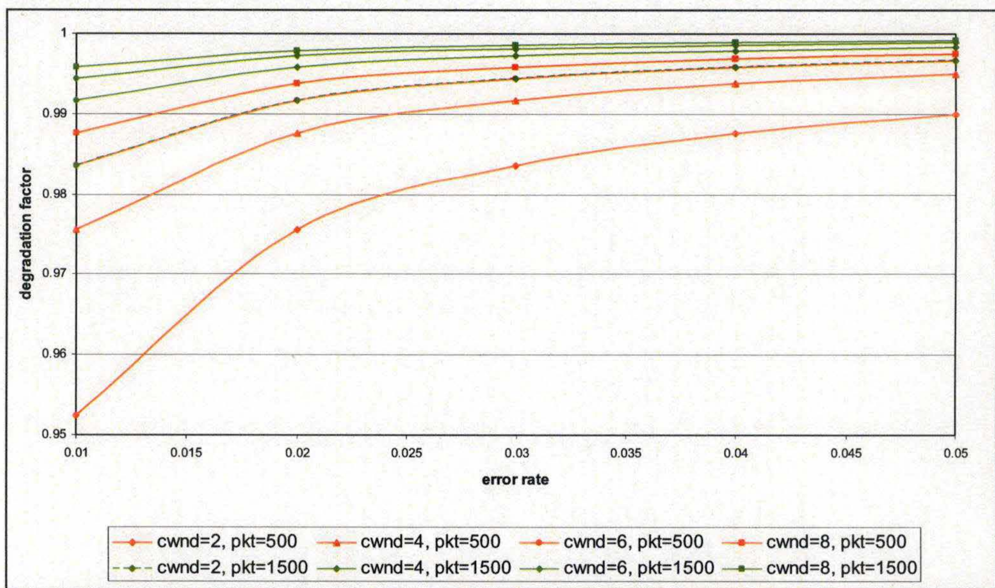


Figure IV.24: Impact of the error rate (on bit level) on the throughput

IV.D Sources

[1] Bonaventure O., **Téléinformatique et Réseaux**, Lecture.

[2] Van Peteghem H., **TCP: contrôle de congestion**, Lecture.

[3] Ahn J.S., Danzig P.B., Liu Z., Yan L., **Evaluation of TCP Vegas: Emulation and Experiment**, Proceeding of ACM SIGCOMM '95, Augustus 2000.

[4] Paxson V., Allman M., **Computing TCP's Retransmission Timer**, RFC 2988, November 2000.

[5] Braden R., **Extending TCP for Transactions – Concepts**, RFC 1379, November 1992.

[6] Wright G.R., Stevens W.R., **TCP/IP Illustrated**, Volume 2, Addison Wesley, ISBN 0-201-63354-X.

[7] Karn P., Partridge C., **Improving Round Trip Time Estimates in Reliable Transport Protocols**, Proceeding of ACM SIGCOMM '87, Augustus 1987.

[8] Allman M., Dawkins S., Glover D., Griner J., Tran D., Henderson T., Heidemann J., Touch J., Kruse H., Ostermann S., Scott K., Semke J., **Ongoing TCP Research Related to Satellites**, RFC 2760, February 2000.

[9] Jacobson V., Braden R., Borman D., **TCP Extensions for High Performance**, RFC 1323, May 1992.

[10] Jacobson V., **Congestion avoidance and control**, Proceedings of ACM SIGCOMM '88, Augustus 1988.

[11] Postel J., **Transmission Control Protocol Specification**, RFC 793, September 1981.

Chapter V: TCP Versions

V.A Introduction

Third generation wireless networks, commonly referred to as 3G, provide a higher user bit rate for both CS and PS connectivity. Higher bit rates will naturally bring about new services. So far, GPRS, the packet-oriented data service for GSM, already offers the users the ability to run several internet applications on their terminal. With UMTS, things will go further. For example, video telephony or fast downloading of data could now be offered to the network customers. And, if there is to be a killer application, it is most likely something based on a quick access to information and its filtering appropriate to the user's location. Actually, in many cases, the information provided by these services will come from the Internet. According to some measurements [1], the majority of the Internet traffic still relies on the TCP protocol. To be more general, TCP seems to get spread everywhere where end-to-end reliability is required. Therefore, most of the interactive and background services to be addressed by WCDMA networks are expected to belong to the group of applications employing a TCP/IP platform. So, this defies UMTS to properly handle the TCP/IP traffic. Regarding TCP, it was designed to provide reliable data transfer over almost any transmission medium regardless transmission delay and its variation, duplication or segment reordering as discussed in chapter IV. This can still seem to be suitable for UMTS but additionally, it has mainly been focused on avoiding congestion collapse. Every packet loss is taken as a hint of congestion meaning that TCP makes a strong reliability assumption on the subnetwork. This assumption comes from the time when the first TCP version was designed. At that time, and thanks to the commonly high reliability level provided by wired networks, losses due to damage were estimated less than 1% [2]. Today, network communications increasingly involve mobility. Therefore, the ubiquity of the wired environment has moved on to a new kind of mixed wired/wireless environment. One of the features shared by wireless networks and especially by those providing mobility, are the general impairment of the link layer reliability setting aside the reliability postulate of TCP. Whether TCP can still provide end-to-end reliability, its ability to get the most of such networks can seriously be put into question.

We have tested several TCP flavours to see which of them are the most adapted to the wireless medium. Every version implements some improvements on the basic TCP implementation explained in Chapter IV. After the overview of the used TCP versions, in section V.B, this chapter aims at analysing the TCP's congestion algorithms in order to show what the assumptions and their impacts are. We finally draw a conclusion upon how TCP behaves on a TCP network and what can be done to improve it.

V.B Overview of the common TCP Versions

In this section we developed the most commonly used and widely spread TCP versions. They are:

- TCP Tahoe
- TCP Reno
- TCP New Reno
- TCP Vegas

- TCP SACK
- TCP FACK

The timer based detection of the lost segments, as explained in chapter IV, is the only mechanism that was defined in the original TCP specification [7]. All TCP implementations support it. This feature is minimally necessary to ensure a reliable data transmission. We will specify the additional mechanisms implemented in the different TCP versions used.

TCP Tahoe (Distributed with 4.3 BSD UNIX)

Early TCP versions, like the ones explained in chapter IV, section A, utilized the go-back-n model using cumulative acknowledgments. Lost data packets are only retransmitted after the expiration of the retransmission timers. These versions had little impact on the network congestion and the packet losses induced by it [4]. TCP Tahoe performs congestion control and RTT estimation in a way similar to the version of TCP released with the 4.3BSD Tahoe UNIX system release from UC Berkeley [9]. The congestion window is increased by one packet per new ACK received during slow-start and is increased by (1/ CWND) for each new ACK received during congestion avoidance [9].

To guarantee a reliable data transmission and congestion control, TCP Tahoe added the following mechanisms on the earlier TCP implementations:

- Slow Start: This method is used to probe the network capacities by increasing exponentially the size of the CWND. Right after the establishment of the connection, Slow-Start is enabled and used to set the Slow-Start-Threshold *(sshthresh)* at the moment the first packet loss occurs. Initially, *sshthresh* is set to the 65,535, the maximum size of the Advertised-Receiver-Window [4]. After the very first packet loss, Slow-Start will be interrupted when the CWND reaches the size of the *sshthresh*. Afterwards Congestion-Avoidance (explained in next section) is used in order not to flood the network's capacity too rapidly.

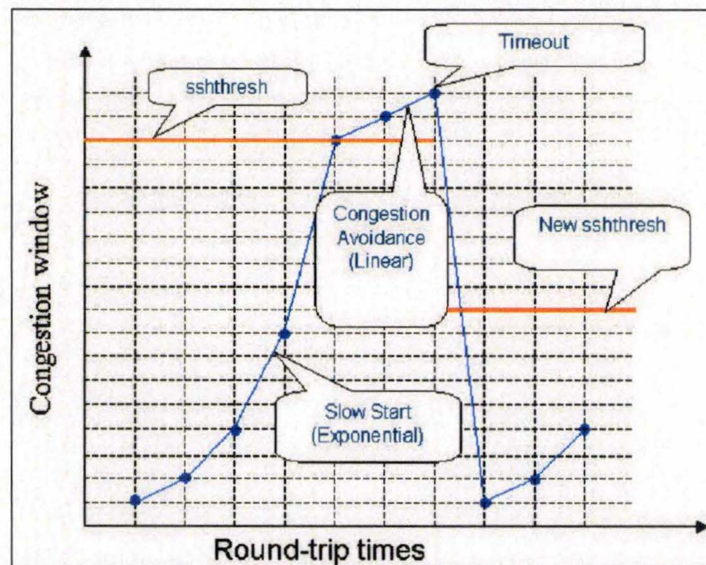


Figure V.25: Slow Start and Congestion Avoidance. Taken from [11]

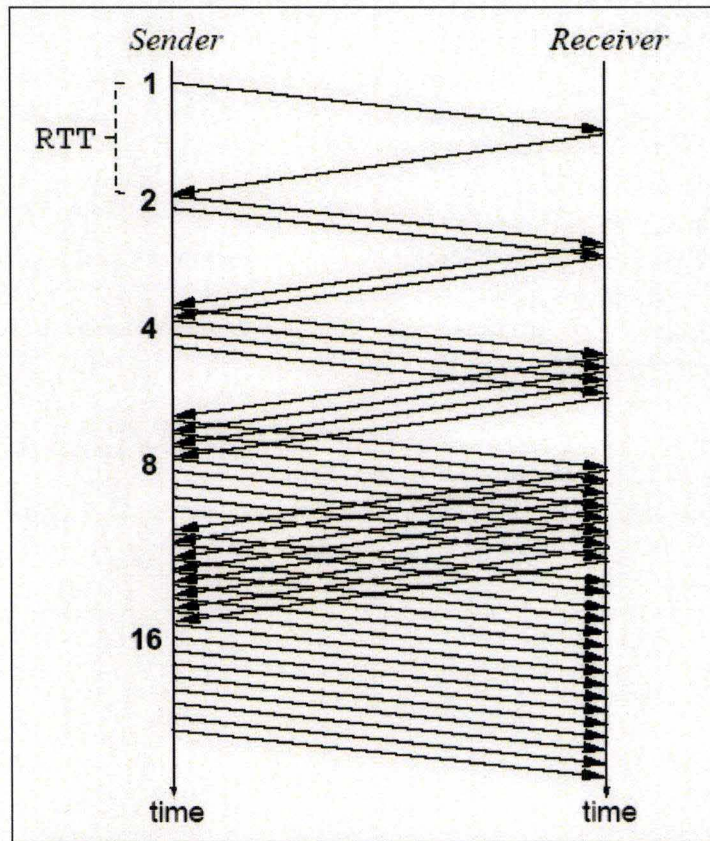


Figure V.26: Exponential Increase Time Diagram. Taken from [12]

- Congestion-Avoidance also known as Additive-Increase-Multiplicative-Decrease (AIMD) [10]: To control the traffic between the sender and receiver, the system of sliding windows, as explained before, is used. The receiver uses the Advertised-Window, which performs flow control according to buffer capacities. The CWND performs congestion control at the sender's side according to the perceived network congestion. When a packet loss occurs and the retransmission timer times out, the packet that was not acknowledged will be retransmitted. At the same time, the CWND is set to one; the sshthresh is set at the maximum between one and the half of the previous CWND size. When a new ACK arrives and if the size of the CWND is smaller than the size of the sshthresh, Slow-Start is launched again. Meaning the increase of the CWND will be exponential. Otherwise, when the CWND is larger than the sshthresh, the CWND will grow additively; this is called Congestion-Avoidance.

- Fast-Retransmit: This is one of the main improvements that increase TCP's efficiency. Instead of waiting for the retransmission timer to expire, a packet known to be lost can be retransmitted more quickly. This feature is triggered when a certain number of duplicated acknowledgments have been received. This value is generally called *tcprexmtthresh* and is often set to three Dup-ACKs. Dup-ACKs are ACKs acknowledging the last packet received in sequence and are sent every time a packet is received out of sequence. As shown in *Figure V.3*, the packets received out of sequence (127, 129, 131) are stored in the receiver's buffer. Once the lost packet 123 has been retransmitted

and received, an acknowledgement (ACK133) for the packets previously stored in the buffers can be sent. After retransmitting the dropped packet, TCP Tahoe enters again in a Slow-Start phase.

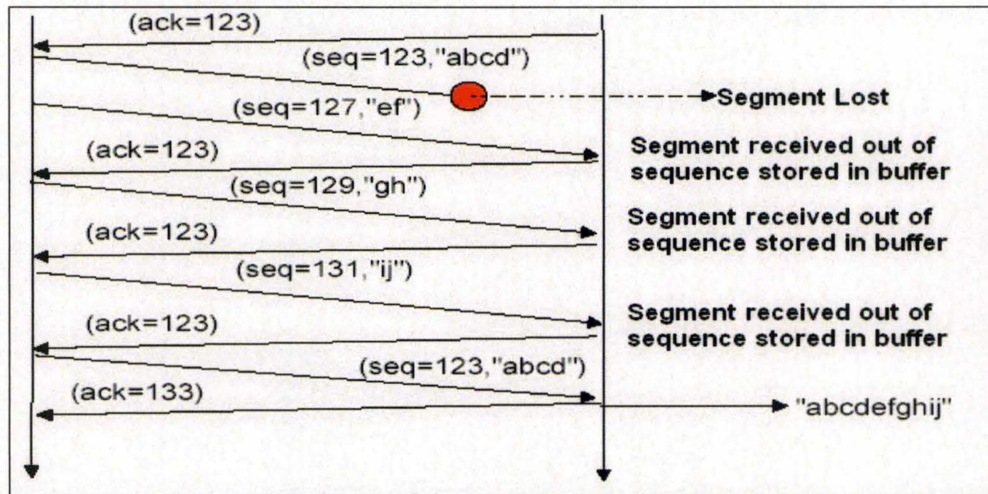


Figure V.27: TCP Fast Retransmit. Adapted from [4]

TCP Reno

TCP Reno adopts all the TCP Tahoe Mechanisms and adds improvements such as Fast-Recovery. Fast-Recovery can be seen as a modification that has an effect on the Fast-Retransmit mechanism for TCP Tahoe. The main purpose of this improvement is to keep the data between the two entities flowing. To reach that aim, the TCP Reno implementation does not return to Slow-Start during a Fast-Retransmit phase. This is justified by the fact that the receipt of *tcprexmtthresh* (usually set to three) [3] Dup-ACKs means that data is still flowing between the sender and receiver. Rather, it reduces the CWND to half the current window and resets the threshold to match this new value and adds the value of *tcprexmtthresh*. As a matter of fact, to evaluate at what size the CWND should be set; TCP Reno takes the minimum of the advertising window size and the CWND size plus *ndup*. The latter one contains zero, unless the number of duplicated ACKs reached *tcprexmtthresh*. As long as duplicated acks are still received, the CWND is inflated by steps of one. Only after the first ACK acknowledging new data arrives, we will reinitialise the CWND to the size of the *sshthresh* value and *ndup* is reset to zero [4]. This ACK is known as the "recovery ACK". We have to reduce this CWND, because the previous growth was done by considering the out-of-order received packets on the other side that were kept in the buffer. Once this stage is finished, we can go back to the usual Congestion-Avoidance. This can improve considerably the throughput in case of single packet drop inside a same window. The side effect of this feature is the loss of performance when multiple drops occur in a same data window [3].

The evolution of the CWND in the case of TCP Reno can be noticed on the *Figure V.4*.

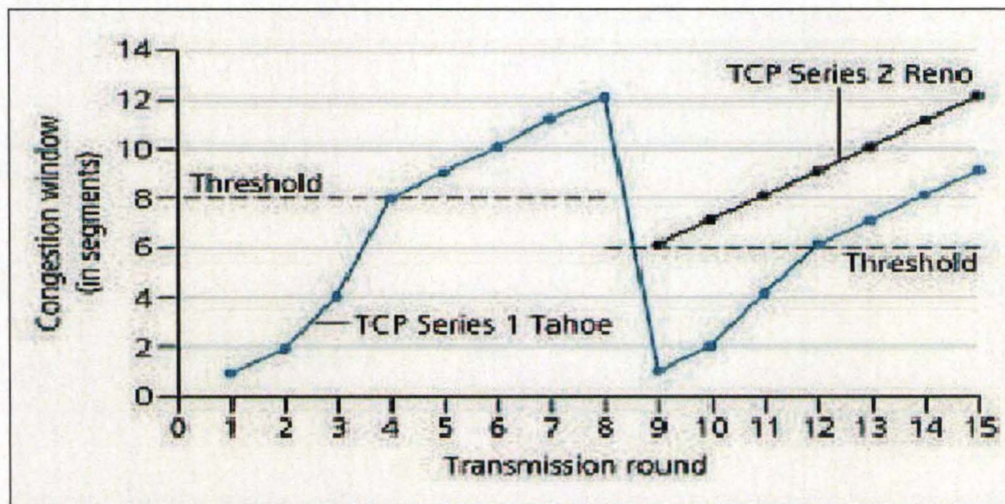


Figure V.28: TCP Reno: Fast Recovery. Adapted from [5]

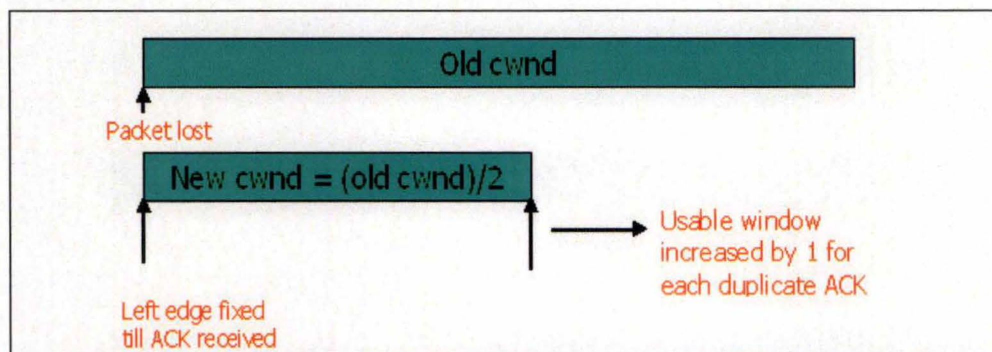


Figure V.29: Congestion window evolution.

TCP New Reno [8]

The main problem with TCP Reno is when multiple losses occur in the same data window. Then it will wait for the retransmission timer to expire. The modification TCP New Reno introduces is situated at the sender side and takes effect during the Fast-Recovery phase. TCP Reno is pulled out of Fast-Recovery when partial ACKs reach the sender side. By partial ACK we mean, an ACK which acknowledges new data, but not all of the packets that were outstanding at the beginning of the Fast-Recovery period [4]. As seen in *Figure V.6,* this induces a shrinking of the current window to the size of the size of the CWND. In other words, the ACK that arrives after retransmission, the one called partial ack, could indicate that a second loss has occurred. New Reno times out when there are less than three acks for the first loss or when a partial ACK is lost. In order to exit Fast-Recovery, the sender must receive an ACK for the highest sequence number sent. Thus, new partial ACKs (which represent new acks but do not represent an ACK for all outstanding data) do not reduce the window. With other words, in case of multiple packets dropped from a single window of data, the first new information

available to the sender comes when the sender receives an acknowledgement for the retransmitted packet (that is the packet retransmitted when Fast-Retransmit was first entered). If there had been a single packet drop, then the acknowledgement for this packet would have acknowledged all of the packets transmitted before Fast-Retransmit was entered (in the absence of reordering). However, if there had been multiple packet drops, then the acknowledgement for the retransmitted packet would have acknowledged some, but not all of the packets transmitted before the Fast-Retransmit. We call this packet a partial acknowledgment.

In brief, these are the main differences between Reno TCP and New Reno TCP:

- In TCP Reno, the first partial ACK will bring the sender out of the Fast-Recovery phase. This will result in the requirement of timeouts when there are multiple losses in a window, and thus stalling the TCP connection.

- In New Reno, a partial ack is taken as an indication of another lost packet and as such the sender retransmits the first unacknowledged packet. Unlike Reno, partial acks do not take New Reno out of Fast-Recovery. This way, it retransmits one packet per RTT until all the lost packets are retransmitted, and avoids requiring multiple Fast-Retransmits from a single window of data. Confer *Figure V.7.*

The downside of this is that it may take many RTTs to recover from a loss episode, and you must have enough new data around to keep the ack clock running.

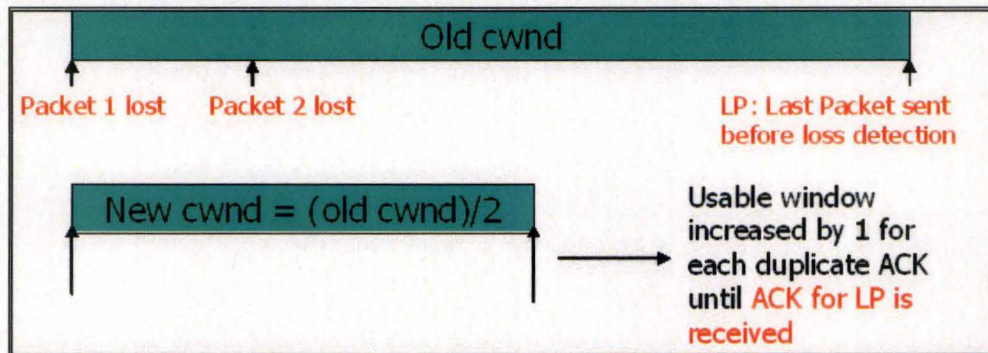


Figure V.30: Congestion window evolution with partial acks.

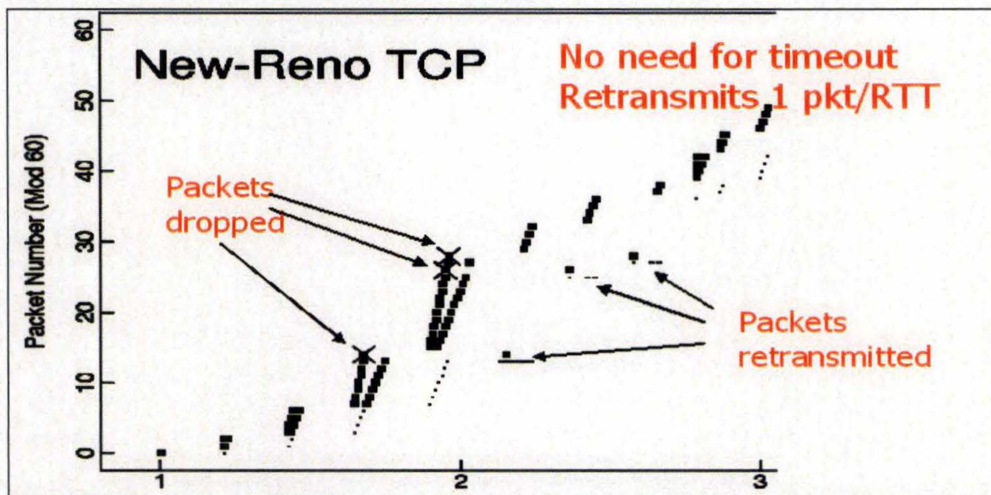


Figure V.31: TCP New-Reno's retransmission.

Figure V.7 as well as several others graphs in this chapter are "traced packets graphs". For each graph the X-axis shows the packet arrival or departure time in seconds. The Y-axis shows the packets number modulo 60. Packets are numbered starting from packet 0. Each packet's arrival and departure is marked by a square on the graph. The packet delay is thus represented by the distance between the two co-linear marks for a constant packet number. Drops are indicated by an "x" on the graph for each packet dropped. Returning ACK packets are marked by a smaller dot.

TCP Vegas

The modifications and additions that TCP Vegas introduces on TCP do only affect the sender side and thus remains compatible with previous versions [6]. The lack of precision during the estimation of the RTT and its variations in the previous TCP versions influences their performance. TCP Reno, for example, retransmits not only when a timeout occurs, but also when *n* (generally 3) duplicated ACKs arrive. TCP Vegas extends that retransmission mechanism by using Timestamp*s*. Time is read and recorded each time a packet is sent. By subtracting that value from the time read at the moment the ACK for that particular packet arrives, it knows the RTT. This accurate value of RTT will be used to decide whether or not a packet has to be retransmitted. The Figure V.8 shows two cases:

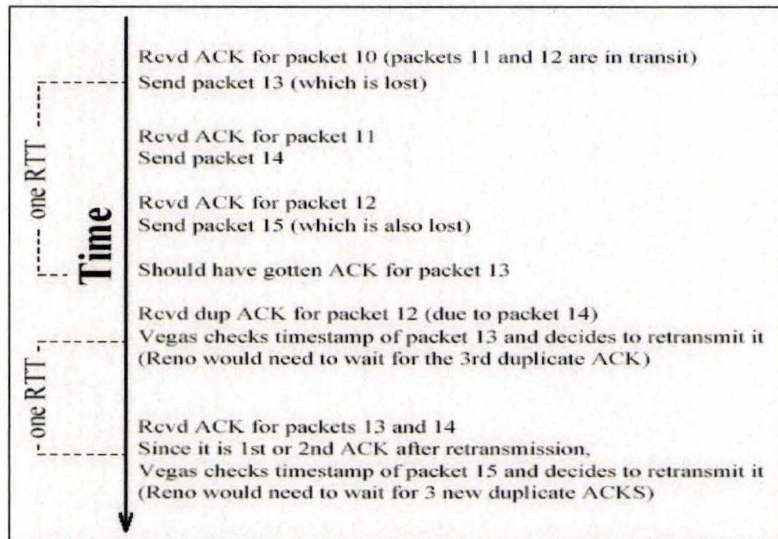


Figure V.32: TCP Vegas: Example of Retransmit Mechanism Taken from [8]

- When a duplicate ACK is received, TCP Vegas checks if the difference between the current time and the timestamp recorded for the relevant segment is greater than the timeout value. If it is, then TCP Vegas retransmits the segment without having to wait for three Dup-ACKs. This improves the performance in case losses are so frequent or the window is so small that the sender will never receive three duplicate ACKs, and therefore, Reno would have to wait for the timeout of its Retransmission timer.

- When an ACK is received which is not a Dup-ACK, and it is the first or second one after a retransmission, TCP Vegas again checks if the time interval since the segment was sent is larger than the timeout value. If so, TCP Vegas retransmits the segment. This will catch any other segment that may have been lost previously to the retransmission without having to wait for a duplicate ACK.

TCP Vegas implements also a new Congestion-Avoidance Mechanism and some modifications to the Slow-Start Procedure.

TCP SACK (Selective ACKnowledgement)

The implementation of the Selective Acknowledgements in TCP does not change the basic underlying congestion control mechanisms. The SACK TCP implementation preserves the robustness of TCP Tahoe and TCP Reno when dealing with out-of-order packets, and uses retransmission timeouts as the last possible recovery method. The main difference can be seen when multiple packets are dropped from one and the same window of data. As in Reno, the TCP SACK enters Fast-Recovery when the data sender receives *tcprexmtthresh* (three) Dup-ACKs. The sender retransmits a packet and divides the CWND to half its size. During Fast-Recovery, SACK maintains a variable called *pipe* that represents the estimated number of packets outstanding on the link [4]. The sender only sends new or retransmitted data when the estimated number of packets in the path, pipe, is less than the size of the CWND. TCP SACK increments the *pipe* variable by one every time the sender sends a packet. This can be a

retransmitted "old" packet or a new packet. TCP SACK subtracts one from the value of the pipe variable when the sender receives a Dup-ACK packet with a SACK option reporting that new data has been received at the receiver. The sender maintains a scoreboard that remembers previous acknowledgments [4]. When the sender is able to send a packet, it retransmits the next packet from the list of packets known to be missing at the receiver side. If there are no such packets and the receiver's advertised window is sufficiently large, the sender sends a new packet. When a retransmitted packet gets dropped, TCP SACK implementation will detect this thanks to the retransmission timeout. It will retransmit the dropped packet and then enter the phase of slow starting. The condition for SACK to quit Fast-Recovery is to receive a cumulative ACK that is acknowledging all packets that were outstanding at the moment the Fast-Recovery state was entered. An example of the SACK TCP's retransmission can be seen of *Figure V.9.*

The main advantage of TCP SACK is its Scoreboard or Bit Mask in which it stores the sequence numbers of the packets that are already acknowledged. This permits TCP SACK to only retransmit packets the receiver doesn't already have in its buffer. This way of avoiding useless retransmissions improves the Goodput. The Goodput is considered as the throughput without counting the retransmissions.

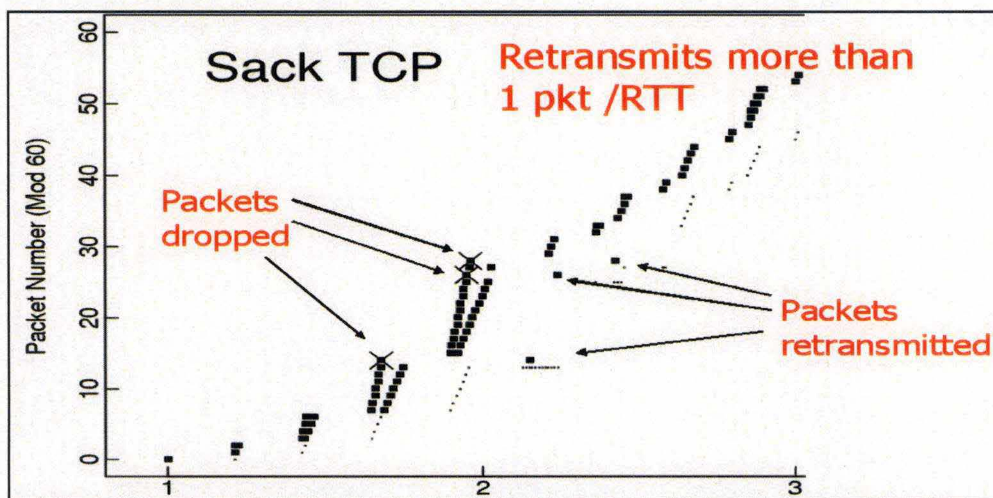


Figure V.33: TCP SACK's retransmission.

TCP FACK (Forward ACKknowledgement)

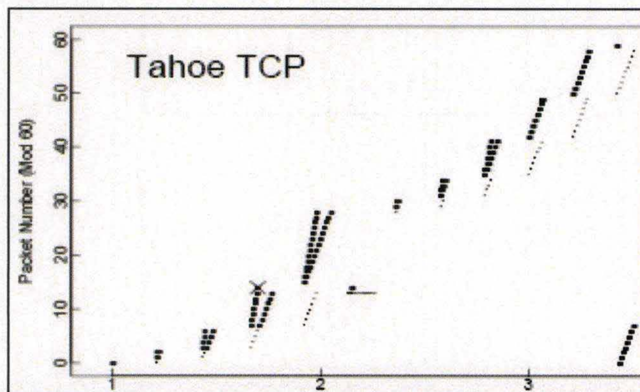
This version can be seen as an extension of TCP SACK. TCP Fack estimates the amount of outstanding data, by comparing the sequence numbers in the SACK packets i.e. TCP SACK packets containing the number of retransmitted packets. If the congestion window is greater than the outstanding data, the remaining data will be sent [10].

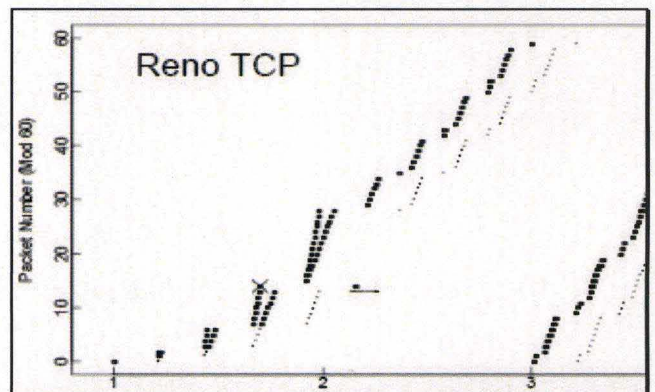
V.C TCP behaviour

TCP Tahoe was the first "modern" TCP version. Since then, all versions that came out were designed to enhance the ability of the TCP congestion control mechanism to face several packet losses within a CWND and so inside one RTT. In this part we were focused on TCP Tahoe, Reno, New Reno and finally TCP SACK. The other version (TCP Vega and TCP FACK) were discarded because NS2 did not included a fully reliable version of them. To be able to compare those versions, we run on our NS2 simulator the simulations executed by the command *test-sack*; proposed by [3]. The simulated topology was made up of a sender and a receiver separated by a finite-buffer drop tail gateway. The link between the sender and the gateway provided a bandwidth of 8Mbps and a delay of 0.1ms. The link between the gateway and the receiver was playing the role of a bottleneck with a throughput of 0.8Mbps and a delay of 100ms. Accordingly, depending on the queue size, the experienced RTT should be slightly over 200ms. In order to obtain the desired drop pattern, 2 connections, in addition to the one studied, were flowing over the network. Even if on this scenario, only drops due to congestion occurred, the comparison between those TCP versions still remains instructive. Indeed, for whatever reason the drop occurs, (congestion or damage) it will always lead to mistakes inside the CWND. Moreover, as high BDP occurs in UMTS networks (see section II.E) leading to high a CWND, it really is our point of concern.

Figure V.34: The simulated topology.

- **TCP Behaviour when one drop occurs**





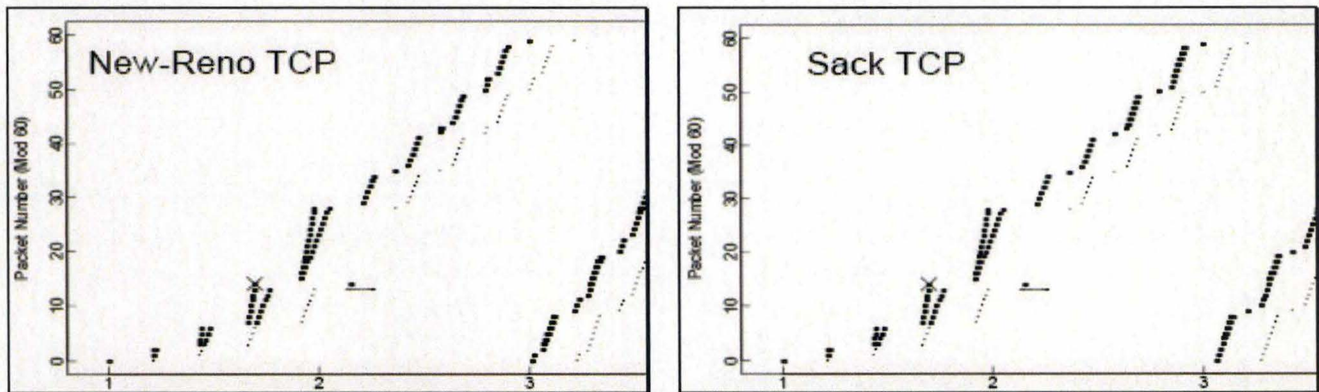


Figure V.35: One dropped packet.

Figure V.11 shows TCP Tahoe, Reno, New Reno and SACK with one dropped packet inside the CWND. Tahoe required for Slow-Start to recover from this drop, while the others, by way of Fast-Recovery, are able to recover smoothly from this error.

With **TCP Tahoe**, according to the Slow-Start algorithm, the CWND increases exponentially from 0 to 28 packets. The packet 14 corresponding to the last packet of the fourth CWND is dropped. This opens a fifth CWND reaching 14 packets instead of 16. The sending of this window yields 14 duplicate acknowledgements of the packet 13. The third duplicate ACK, meeting the ACK threshold invokes:

- Fast-Retransmit retransmitting the segment 14
- Slow-Start resetting:
  - The *sshthresh* at 7 (half of the fifth CWND)
  - The CWND at 1

As packets 15 to 28 have already been sent but not yet acknowledged and the CWND's size is now kept at one. The sender has to wait for those acknowledgments to be able to send anything else. The receiver has already cached packet 15-28, and upon receiving the retransmitted packet 14 acknowledges packet 28. The ACK for packet 28 causes the sender to increase its CWND by one reaching a size of two. The Slow-Start phase carries on with packet 29 till packet 35. Then, the sender reaches the *sshthresh* and enters the Congestion-Avoidance mode. From throughput point of view, this is quite bad: the pipe goes empty and a new equilibrium needs to be started from scratch.

With **TCP Reno, New Reno and SACK**, Fast-Recovery gives optimal performance in this context. The third duplicate acknowledgments triggers:

- The retransmission of packet 14
- The reduction of the CWND at 7 (half of the current CWND)
- The resetting of the *sshthresh* at 7
- The setting of the sender into Fast-Recovery mode

During the Fast-Recovery, the 14th duplicate ACK inflates the CWND to 21 (one packet per duplicated ACK received). The inflated window from the last six duplicated packets allows the sender to send packet 29 to 34. The beginning of the CWND, being filled in by packet 14 to 28, is still waiting for acknowledgment. Upon the receipt the acknowledgment of packet 28, the sender exits Fast-Recovery

and continues in Congestion-Avoidance with a CWND brought at 7.

- **TCP Behaviour when two drops occur**

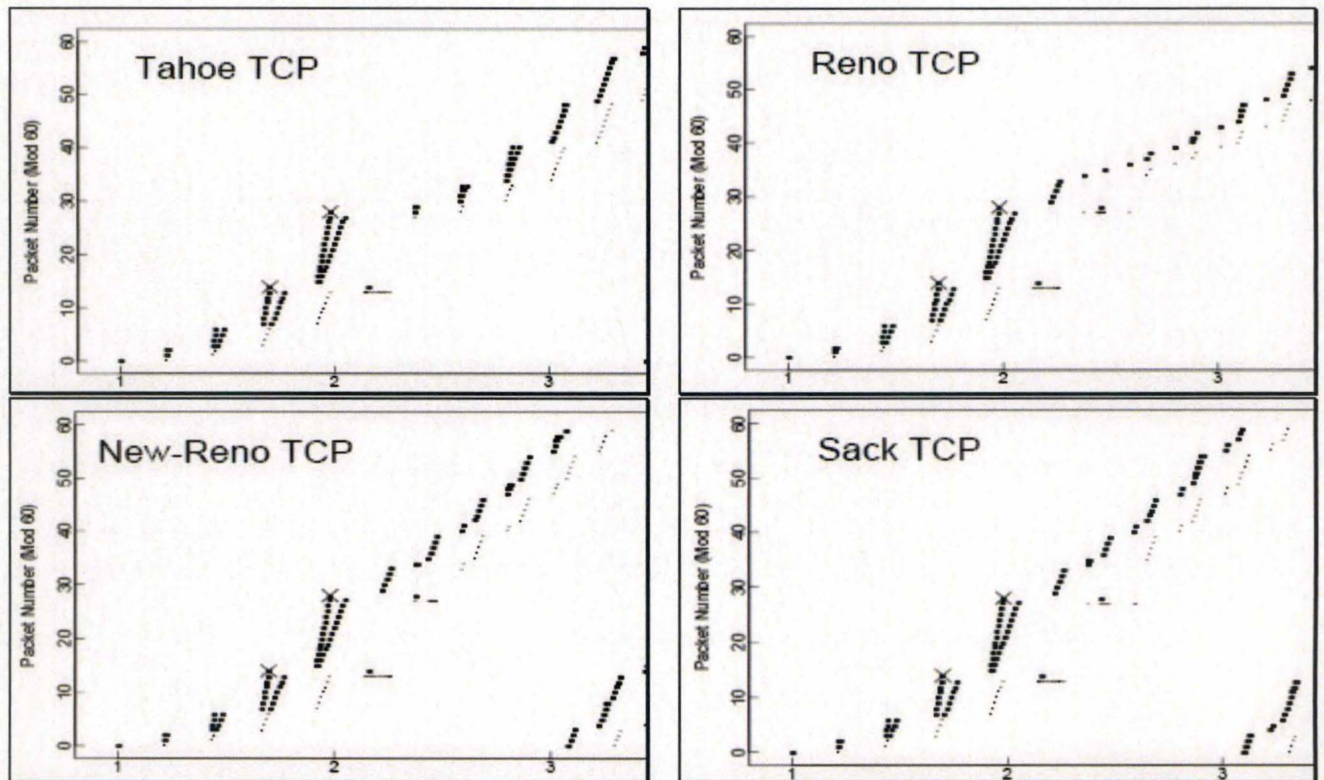


Figure V.36: Two dropped packets.

Figure V.12 shows Tahoe, Reno, New-Reno and TCP SACK with two dropped packets. As already mentioned, Tahoe recovers from the packet drops by way of Slow-Start. So, for Tahoe, two drops instead of one involves only few changes. TCP Reno recovers with some difficulties, while both New-Reno and SACK recover smoothly.

With **TCP Tahoe**, the response to the loss on packet 14 is already described in the single loss case. Instead of heaving 14 duplicated acknowledgments, the sender gets 13. But it is still above the ACK *threshold*. With Tahoe, even if packets 15 to 28 have already been sent, these facts are forgotten when it receives the three duplicate ACKs and restarts the Slow-Start phase. The sender simply resends packet 14 and is now waiting for acknowledgment. Given the fact that packet 28 got lost, ensuing the retransmission of packet 14, the sender receives an ACK for packet 27. Following the acknowledgment, a window of 2 packets is opened and packet 28 and 29 are sent.

With **TCP Reno,** the sender recovers by using twice Fast-Recovery. The first drop triggers 13 duplicated ACKs (rather than 14 in the previous situation) from the packet 14. This allows the sender to enter in a Fast-Recovery mode with an enlarged CWND of 20 (rather than 21). After the retransmission, set off by the 3 first duplicated ACKs, 15 packets of the CWND are waiting for being acknowledged. So, this allows the sender to send packet 29 to 33. Ensuing the retransmission of the packet 14, an ACK

for packet 27 is received. This puts the sender out of the Fast-Retransmit mode with a window of 7 packets. Six packets are still waiting for acknowledgment (28-33), so with a CWND of 7 packets, the sender is still allowed to send packet 34. As packet 28 got lost, 6 duplicates ACKs are triggered from packets 29-34. This sets off the retransmission of packet 28 and invokes again the Fast-Recovery mode with a CWND of 3. The reception of the six duplicated ACKs inflates the CWND up to 9 allowing the sender to send packet 35 and 36. The sender receives an ACK for packet 34 as a result of the retransmission of packet 28. This brings the sender out of the Fast-Recovery mode with a CWND and an *sshthresh* of 3.

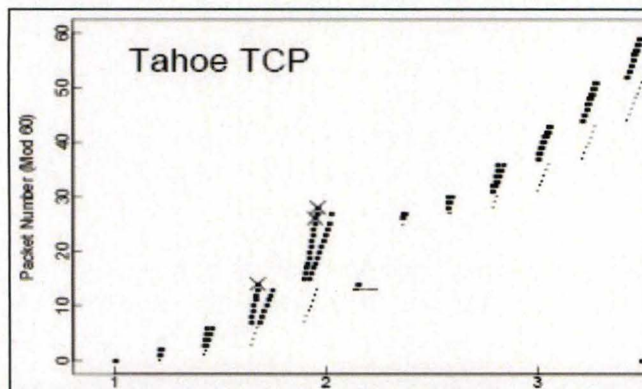
New Reno TCP behaves similar to TCP Reno until the sender receives the first ACK for packet 27. This is a partial ACK and causes New Reno to transmit packet 28 immediately and not exiting Fast-Recovery. With the arrival of the five duplicated ACKs coming from packets 29-33, the sender sends packets 35-38. The ACK for packet 33 causes the sender to exit Fast-Recovery with a CWND of 7 and continues in Congestion-Avoidance.

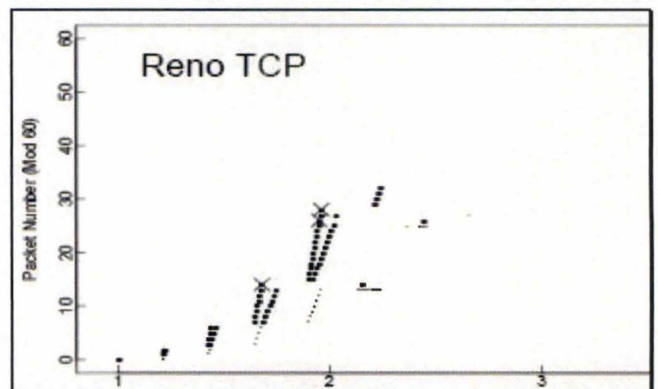
SACK TCP behaves similar to Reno until the third ACK for packet 13 is received. This triggers:

- The protocol to initiate a variable pipe as follows:
  Pipe := CWND – ndup =15 - 3
- The CWND at 7 (half of the current CWND)
- The retransmission of the packet 14.

The variable *pipe* represents an evaluation of the number of packets still being present inside the network. Every time an ACK is received (duplicated or not), a packet is assumed to have left the network. The sender can keep sending packets as long as the value of this variable is lower than the CWND size. The next 10 duplicated ACKs decrease the value of the pipe variable to 2 packets. The sender is now being able to send packets 29-33. Ensuing this, pipe value has now 7 packets. The first ACK for packet 27 is recognized as a partial ACK meaning that packet 28 is assuming lost. As a result the pipe is decreased by two. Packets 34 and 35 are sent. The five additional duplicated ACKs (packet 29-33) allow the sender to send packets 36-39, and so on up to packet 41. The acknowledgment of the packet 35, thus including packet 28, brings the sender out of the Fast-Recovery phase with a CWND of 7 and continues. The sender now carries on under Congestion-Avoidance.

- **TCP Behaviour when three drops occur**





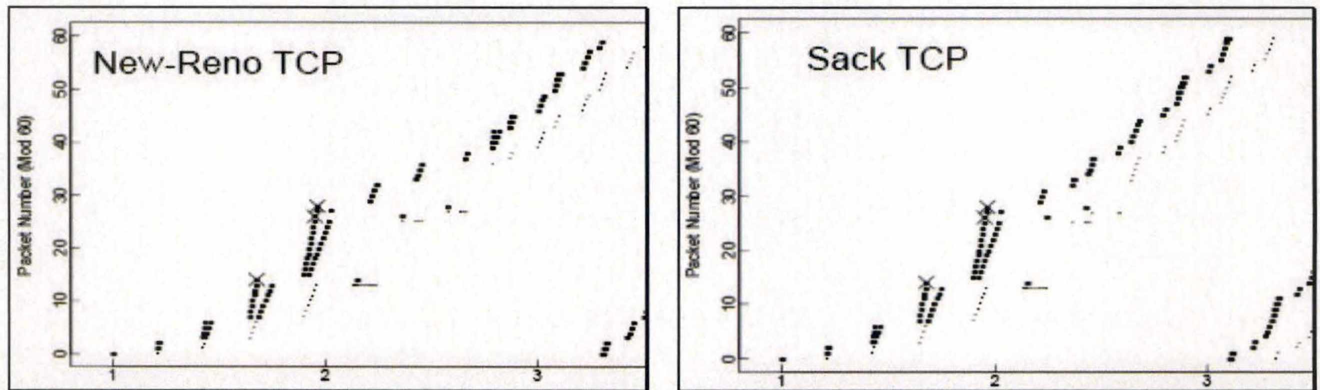


Figure V.37: Three dropped packets.

With **TCP Tahoe**, the response to the loss of packet 14 is the same as described for Tahoe in the single loss case. As in the two packet loss case, even though packets 15-28 were sent, this is not taken into account by the sender. After retransmitting packet 14 and receiving 12 duplicated ACKs, the sender receives an ack for packet 25. The sender, in Slow-Start, opens a window of 2 packets and sends packets 26 and 27. Note that even though packet 27 has already been sent successfully, the sender continues in Slow-Start until packet 37. This situation highlights another Tahoe drawback: it may happen that a packet already successfully transmitted is retransmitted again, wasting some valuable network resources.

Again with **TCP Reno** Fast-Recovery will be applied twice. The additional drop causes only 12 duplicate acks. So packet 14 is resent, and the sender is now in Fast-Recovery mode with a CWND of 7 packets and a usual window[2] of 15 (packet 14-28). The 12 duplicate acks inflate the CWND up to 19 allowing the transmission of 4 packets (29-32). With the arrival of the first ACK for packet 25 (coming from the resending of packet 14), Reno exits Fast-Recovery but after reaching the ACK threshold again, re-enters in it with a window of 3 and a usual window of 7 (packets 26-32). The receipt of the 4th duplicated ACK (from packets 29-32) for packet 25, inflates the CWND up to 7. So the sender is still unable to send any additional packets (the seven packets of the CWND are unacknowledged). The reception of the ACK 27 (coming from the resending of packet 26) puts the sender out of Fast-Recovery with a CWND of 3. Five packets are still waiting for acknowledgment (packet 28-32). The sender is stalled and is waiting for retransmitting timeout.

TCP New-Reno with 3 dropped packets operated like Reno until the receipt of the first ACK for packet 25. This is a partial ACK and so New-Reno retransmits immediately packet 26. Now the usual window is set at 7 (packets 26-32). The four subsequent ACKs (coming from packet 29-32) for packet 25 allow the sender to send packets 33-36. The next ACK acknowledging packet 27 is a partial ack. This causes the sender to retransmit packet 28 and reduces its usual window to 9 (packets 28-36). The sender is unable to send additional data until the receipt of the fourth ACK for packet 27 (coming from packets 33-36). So the sender is allowed to send packets 37 and 38. The ACK for packet 36 brings the sender out of Fast-Recovery with a CWND to 7.

With **TCP SACK**, the sender is aware of the entire state of the transmission. Rather than sending packets 29-32 like Reno, SACK sends packets 26 and 29-31. The handling of the pipe is similar to SACK with 2 packet drops. Except that upon arrival of the duplicated ACKs, pipe value is decreased by

[2] i.e. the number of outstanding packets.

12 rather than 13. Moreover, the retransmission of the packet 26 is accounted for.

- **TCP Behaviour when four drops occur**

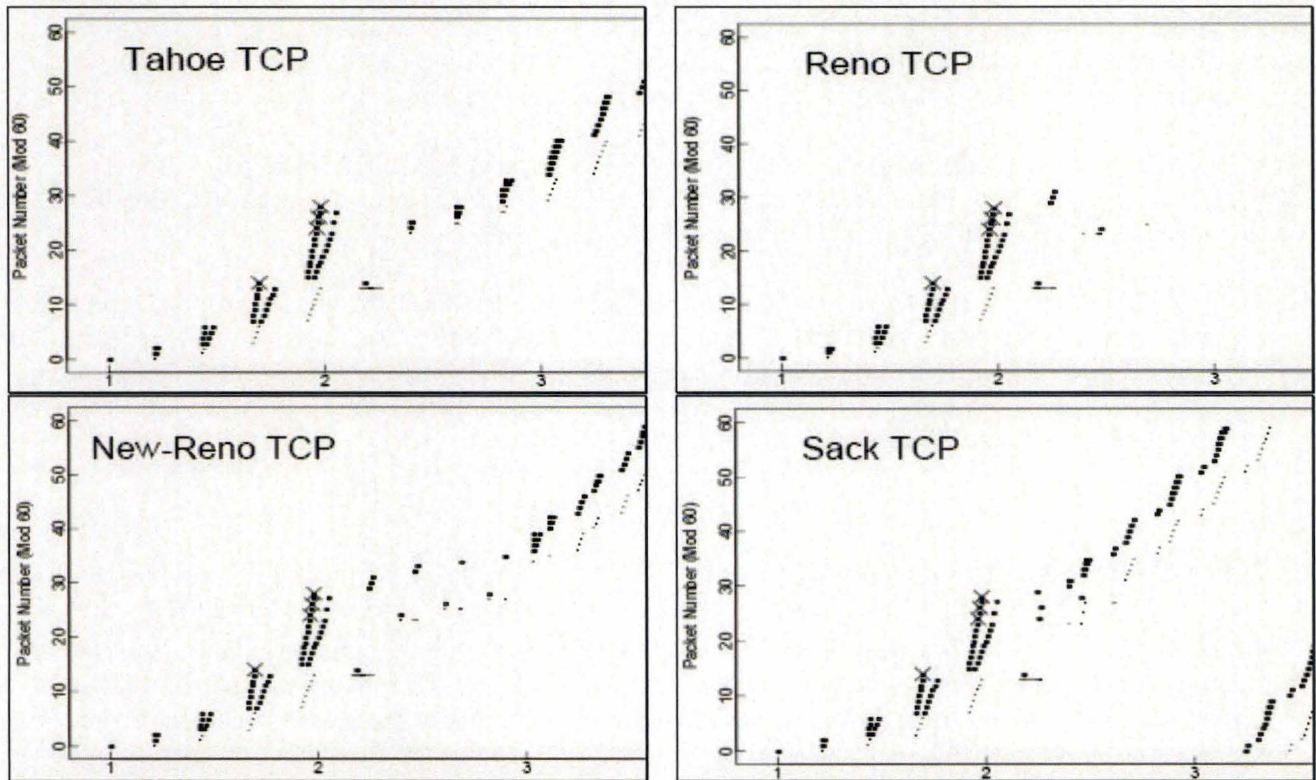


Figure V.38: Four dropped packets.

As in the previous situations, **Tahoe TCP** recovers from the packet drops with Slow-Start. So this situation is rather similar too.

As far as **TCP Reno** is concerned, similar as in the previous situation, the sender has to wait for a retransmit timer to recover from the dropped packets.

TCP New Reno makes the difference with Reno after the receipt of the first partial ACK acknowledging the packet 23.

With **TCP SACK**, the receiver is able to indicate everything via packet 24 and 26. The sender can directly retransmit those packets.

Some comments can be made:

- In context of congestion, SACK has proved its ability to handle packet drops. However, for all the versions, the congestion strategy led to a decrease of the sending behaviour on the ground of packet drops. For some versions, the pipe even goes empty (Tahoe and sometimes Reno). Our first

conclusion is: whatever version, as long as losses are taken as a hint of congestion, TCP performance will rather be poor when transmission damages occur.

- Generally speaking SACK performs better. However when numerous drops occur in bursts (in a same window or in two subsequent windows), its performance is quite similar to TCP Tahoe. Due to the fact that under SACK TCP, when packets are required for retransmission, the sender has no proof that the packet has really left the network. It might simply be caused by delay. The sender cannot "replace" those packets by "new" packets and so more RTTs are needed to recover from.

- TCP New Reno needs one RTT per drop. Four RTTs are required to recover from 4 drops, caused by the fact that New Reno needs a partial ACK to notify that an error has occurred. This leads to rather poor performance in context of errors occurring in burst.

- Regarding Reno TCP, given the fact that it often needs Retransmit-Timeouts to recover from packet drops, it is not the best choice. It is worth mentioning that the situation, in which the sender has to wait for a retransmit-timeout, might also happen with SACK, when a retransmitted packet is lost again. As with TCP Reno, this will result in a very poor performance. Fortunately, the probability to end up in this situation is more unlikely than with TCP Reno.

V.D Conclusion

As explained in Chapter II section E, UMTS network represent a highly variable environment for data flowing over it. Moreover, it is what is called a LTN. Accordingly, TCP is forced to open a rather big CWND in order to flood the network. This increases the probability to see a drop occurring into it. But the question is: despite this, which would be the best TCP candidate? In context of short-term burst error, Tahoe would be the best choice. It is the only one that can recover, no matter what, in only one RTT. Unfortunately, afterwards the use of the Slow-Start leads to a big underutilisation of the network resources. Accordingly in context of constant (and low) error rate SACK should be preferred. The several steps of improvements to the initial TCP algorithm described above are really beneficial as long as drops can be assumed coming from congestion. Whatever version used, those ameliorations completely loose their effect and in some case, could worsen the protocol's performance. Finally, all those versions still react to the drops by launching a downward readjustment of the used bandwidth. Obviously, this downsizing behaviour is not at all desired in a way that no congestion is about to occur. One option could be to render TCP aware of those kinds of errors. This has a major drawback: all the TCP stacks around the world would need to be changed. Another option could be to provide LL recovery in order to enhance the reliability of UMTS network.

V.E Sources

[1] Claffy K., Miller G., Thompson K., **The nature of the beast: recent traffic measurement from an Internet backbone**, Proceedings of INET '98, Geneva, Switzerland, 1998.

[2] Jacobson V., **Congestion avoidance and control**, Proceedings of ACM SIGCOMM '88, Augustus 1988.

[3] Fall K., Floyd S., **Simulation-based Comparisons of Tahoe, Reno, and SACK TCP**, Proceedings of ACM SIGGCOMM '96, July 1996.

[4] Bonaventure O., **Téléinformatique et Réseaux**, Lecture.

[5] Van Peteghem H., **TCP: contrôle de congestion**, Lecture.

[6] Brakmo L.S., Peterson L.L., **TCP Vegas: End to End Congestion Avoidance on a Global Internet**, IEEE Journal of Selected Areas in Communication, October 1995.

[7] Postel J., **Transmission Control Protocol Specification**, RFC 793, September 1981.

[8] Floyd S., Henderson T., **The New Reno Modification to TCP's Fast Recovery Algorithm**, RFC 2582, April 1999.

[9] Jacobson V., **Congestion avoidance and control**, Proceedings of ACM SIGCOMM '88, Augustus 1988.

[10] Breen J., **TCP Performance Issues**, Lecture. Available at http://www.csse.monash.edu.au/~jwb/subjects/cse5803/tcpperf/tcpperf.html, Last visited: September 29th, 2003.

[11] Teyeb O., Wigard J., **Emulation of TCP Performance Over WCDMA**, June 2003. Available at http://cpk.auc.dk/FACE/documents/deliverable_2.1.pdf, Last visited: September 29th, 2003.

[12] Park K., **Congestion Control**, Lecture. Available at http://www.cs.purdue.edu/homes/park/cs422-congestion-2.pdf, Last visited: September 28th, 2003.

Chapter VI: Analysis and Mitigations

VI.A Introduction

As chapter V concluded, TCP has some major drawbacks that make it rather inefficient when used over a UMTS LL. In brief, in a UMTS environment, TCP has to face (see section II.E):

- drops due to damage
- bandwidth and delay variation
- high BDP

In this chapter we will study the ability of TCP to carry data in three different contexts. The first one will simply reproduce an environment where error and transmission rates remain constant during the whole lifetime of the connection. We will show that LL recovery can seriously enhance the overall performance of TCP but at the cost of serious impact on the delay and its variation. Afterwards we will try to evaluate the ability of TCP to adapt itself to bandwidth oscillations. We will see that TCP can rather handle them as long as we remain in "reasonable margins". Finally, we will simulate the performance of TCP running over LL exhibiting some error rate oscillations. In all these situations, we will note that besides the LL recovery mechanism, the buffer size is the major feature that influences TCP.

VI.B Metric

The issue of the TCP evaluation over the above-mentioned environment raises the question of the performance metric used. One classical strategy leads to analyse how well TCP will fill the pipe between the sender and the receiver. Can TCP really get the most out of a specific path? On the other hand, one might argue that such "classical metrics" based on throughput and delay, could hide some protocol's inefficiencies [3]. Throughput can be used to measure the rate at which data is flowing over the channel. Unfortunately enough, the effort of TCP to achieve its throughput is not taken into account. Considering an increasing number of battery-powered devices, an appropriate evaluation should consider energy efficiency or transmission effort.

Inside a TCP connection, the traffic flowing between the sender and the receiver will depend upon the way TCP has probed the network. Probing the network is done on the assumption that every packet loss can be seen as a congestion sign. The fundamental problem of TCP in a mixed environment is due to its inability to detect the nature of the errors, considering only their consequence. Namely, that the packet is dropped. As mentioned in the section II.E, two kinds of error patterns can occur inside a wireless segment: transient random errors and burst errors. As the error control algorithm of TCP is centred on the congestion loss, both will lead to the same TCP reaction: readjustment of the CWND, resending of the lost packets and finally a gradual increase of its reduced window size.

In the presence of a transient random error pattern, the TCP back-off strategy will not avoid any future retransmissions coming from a potential congested network – because the network is not about to be

congested. This behaviour will be at the cost of a serious decrease of the throughput. Besides, during the Slow-Start phase following the recovery procedure opportunities for error-free transmissions are wasted and communication time is extended, increasing the energy cost of the communication. Added to the adjustment of the congestion window, the timeout will be extended. The extension of this timeout reduces the ability of the network to quickly detect further errors occurring in the network. As a conclusion, in the presence of transient random errors, inappropriate low throughput, leading to bad energy management, is observed.

The presence of a burst error pattern will lead to a succession of Slow-Start phases. Mostly, if this behaviour seems to be well adapted from a throughput point of view, it will not be in favour of energy consumption. Indeed, only minor throughput improvements will be observed at the cost of high transmission expenditure.

Shortly, from the perspective of energy expenditure, in a context of heterogeneous wired/wireless networks, TCP seems to have an inherent tendency to back-off too much when it should not, and too little when it should [4].

Although the conclusions developed in [4] seem to be relevant. For our analysis we decided to focus on the "classical metric". Indeed, even if it appears bias in the way that this does not take the energy expenditure into account, this metric seems more convenient to interpret results.

VI.C Constant error rate environment

LL recovery

One of the major TCP drawbacks is its inability to handle the high error transmission rates occurring within the UMTS networks. One proposal tries to solve this problem by making higher layers unaware of transfer losses occurring on the UMTS link. In this case recovery mechanisms are useful.

Error recovery can be done on the ground of the generation and the transmission of redundant information computed from user data. Depending on how much redundant information is sent and how it is generated, the user can use it to reliably detect transmission errors, to correct up a maximum number of transmission errors or even both. The general approach is known as Error-Control-Coding (ECC). When enough ECC information is available to allow the receiver to correct transmission errors, the approach is known as Forwarding-Error-Correction (FEC). For hardware reasons, usually FEC mechanisms are implemented on the physical layer. Another possibility is provided by means of retransmission mechanisms known as Automatic-Repeat-Request (ARQ). In this situation, on ground of the detected errors by means of the ECC information, a repeat request is sent by the receiver to the sender.

Depending on the layer they have been implemented in, error mechanisms can operate on short paths like LL or on an end-to-end basis. On internet, the only way to ensure a reliable service to applications is to appeal to end-to-end recovery mechanisms [1]. In our analysis, the use of recovery mechanisms

has not been proposed in order to provide reliability to the application layer, but rather to offer an enhancement of the UMTS LL reliability to the network and the transport layer. The ARQ mechanism has to be implemented at the LL level. However, the duplication of the ARQ mechanism in two different layers i.e. transport and LL, of the protocol stack can lead to some inefficient interactions. Indeed, in our case, LL recovery leads to an increase of delay variations [7]. This can easily be explained by the different error patterns occurring inside the UMTS network. As an example, errors occurring in a short and random pattern will lead to few LL retransmissions and will therefore induce a delay spike. In the presence of a burst error pattern, the situation could be more disturbing if one LL packet is requested several times. Actually, interactions start occurring when information asked by the upper layer is delayed by the lower layer for retransmission purposes. Related details can be found in [7].

To highlight the impact of the LL on the **delay**, we have run several simulations with and without ARQ mechanism. As it depends on the BLER value, we have seen how significantly the delay can be affected in its deviation. *Figures VI.1, VI.2 and VI.3* depict those simulations. The context of the simulation is the following: downlink SF = 8 (leading to an available bandwidth of 480kbps), initial windows size = 1 packet, TCP packet size = 540 B, file size = 1 Mb, TCP version = Tahoe. BLER is either 1 or 10%.

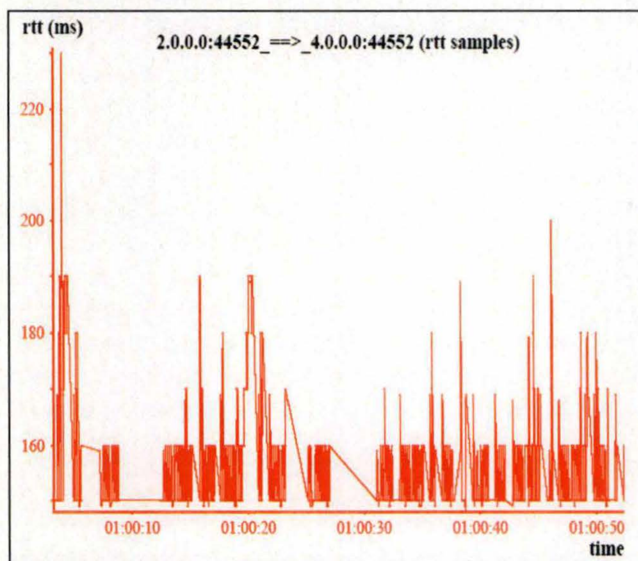


Figure VI.39: RTT variation without LL recovery, BLER = 1%

Figure VI.1 shows that the minimum RTT experienced throughout the simulation is about 140ms. It is basically the one that can be encountered when the queue located ahead of the bottleneck is empty. Concerning the RTT average, it can be estimated around 158ms and even in the worst cases, the RTT never goes beyond 240ms. The fact that, the RTT is kept rather low and that its variation is limited, can be explained by the number of transmission errors occurring over the wireless segment during the whole simulation. Indeed, each of them is taken as a sign of congestion. As a result, the size of the CWND is reduced below the BDP. In that context the queue preceding the bottleneck frequently runs out of packets, and therefore no extra time coming from a potentially short period spent inside the queue, needs to be added to the global sending time of the packet.

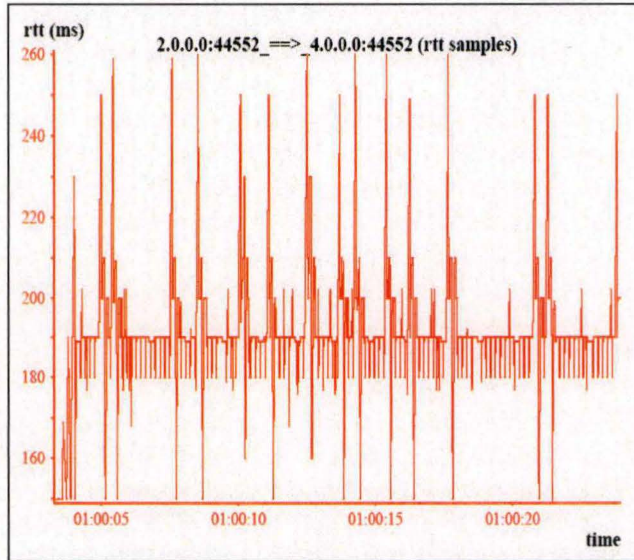


Figure VI.40: RTT variation with LL recovery, BLER = 1%

In *Figure VI.2*, the TCP flow experiences a full LL recovery along with an unlimited queue size prior to the bottleneck. Because of that configuration, transmission errors can never occur. Indeed, by LL recovery, TCP is made unaware of drops resulting from damage and as the buffers cannot be overflowed, drops due to congestion never occur. A RTT average occurring inside the network of around 192ms can be observed. That is 40ms more than the one observed without LL recovery in *Figure VI.1*. This can be explained by a higher queue level. Indeed, as no drops have been experienced, the CWND size has reached its maximum. Moreover, in case of retransmissions, peaks almost reach values around 250ms, i.e. 60ms on top of the RTT average. The average peak size can be explained by a LL timeout set up at 60ms meaning that if errors occurred inside the wireless segment, LL retransmissions are requested 60ms later. An average peak of 60ms above the RTT average shows that if LL segments are requested for retransmission, it occurs rarely more than once.

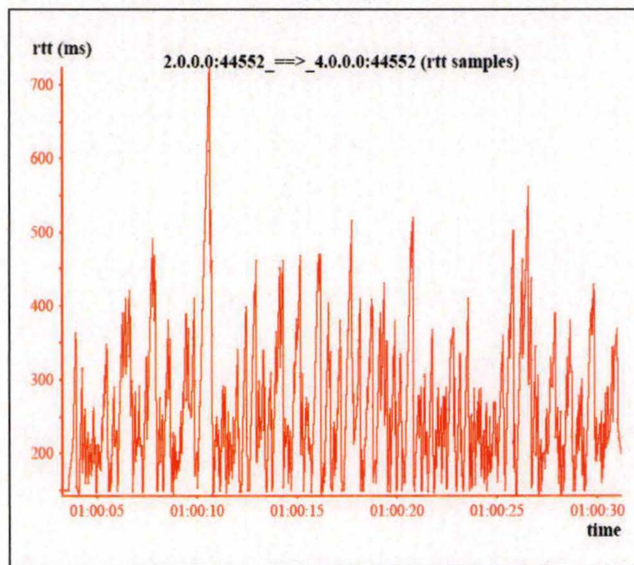


Figure VI.41: RTT variation with LL recovery, BLER: 10%

Figure VI.3 shows the highly variable nature of the RTT deviation meaning that quite often, LL packets are required for retransmission several times. Moreover, as the RTT never seems to go below 180ms, a higher queue-level occupancy than in the two previous scenarios can be expected.

All of this highlights how retransmissions can affect the **RTT average** and the **RTT deviation.** A major part of the **RTT average** can be explained by the average time each packet spends in the buffer, waiting to be transmitted. On the other hand, the **RTT deviation** can be explained by the use of an ARQ mechanism. The fact it has reached such a high level can result form the reliability enhancement mechanism. Our module provides a complete LL recovery resulting in the fact that no drops coming from damage can occur. Moreover, the LL provides an in-order delivery service. This means that the recovery mechanism implemented, does not change the order of the packets throughout the simulation. Thus, packets can be delayed because:

- Many packets are already waiting inside the queue.
- Several segments of the packet are corrupted and need to be retransmitted.
- Another packet with a lower packet number is being retransmitted. With as result that every packet following this one, still has to wait for its retransmission even if it already has successfully been transmitted.

But, whether a general average RTT can only be disturbing for interactive data flows, a highly variable RTT can be more disturbing in terms of TCP performance. In fact, as mentioned previously, a RTT deviation can lead to interaction problems between the ARQ mechanisms (from the LL and TCP) and poor performances will be the result. In view of that, this leads us into 2 questions:

- Which is the reliability level needed by TCP from a LL point of view?
- Which is the minimum buffer size allowing TCP to reach the BDP of the pipe?

To emphasize the impact of LL recovery mechanisms on several TCP versions (Tahoe, Reno, New Reno and SACK), we run numerous simulations in different contexts. Now we aim at evaluating the **throughput** point of view. More precisely the BLER values concerned are 1 and 10%. Regarding the protocol stack, different TCP packet sizes (540 and 1,540 B) and initial window sizes (1, 2 and 4 packets) have been tested. As to the bandwidth, downlink spreading factors (SF) of 128, 64, 32 and 8, leading to an available bandwidth of 30, 60, 120 and 480 kbps have also been tested. It is also worth mentioning that, the queue size problem has been set aside by avoiding any upper boundaries. Beside the performance enhancement, the main conclusion that can be drawn is, that as long as LL recovery was provided, every TCP version tested offered exactly the same results.

	BLER: 0%	**BLER: 1%**	**BLER: 1% LL recovery**	**BLER: 10% LL recovery**
Throughput	**51 344 Bps**	**2132 Bps**	**50 478 Bps**	**37 409 Bps**
RTT average	**188.8 ms**	**158.1 ms**	**192.2 ms**	**259.5 ms**
RTT deviation	**4.5 ms**	**10.9 ms**	**13.7 ms**	**83.3 ms**

Table VI.7: Impact of ARQ on throughput and RTT statistics

Table VI.1 shows how LL recovery can be of use, from throughput point of view. The simulation took place in the following context: downlink SF: 8, initial windows size: 1, packet size: 540B, TCP version: Tahoe. We notice that even in context of high BLER (10 %), recovery mechanisms still allow the

average throughput to remain acceptable. For example, at a 10% BLER, the experienced throughput reaches 72% of the throughput measured in an error free context. The results of the other TCP versions were identical.

In order to compare the different congestion algorithms, we traced the instantaneous throughput as to the different TCP versions (Tahoe, Reno, New-Reno and SACK). All these four simulations were run in exactly the same context i.e. having the same error distribution. The other settings of the simulation environment are: BLER = 10%, file size = 1MB, TCP packet size = 540B, initial window size = 1 packet and SF = 8.

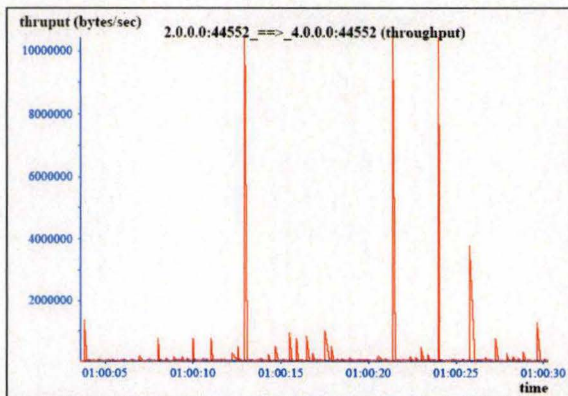


Figure VI.42: Throughput of TCP Tahoe

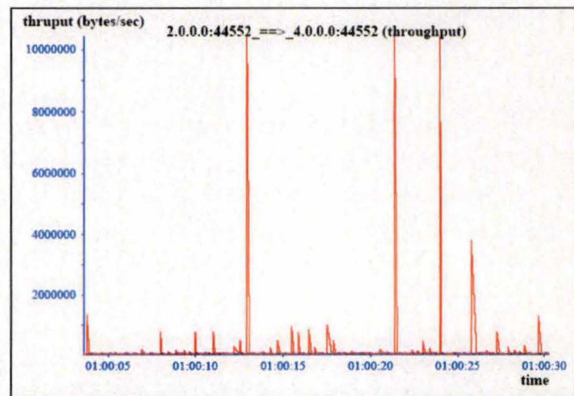


Figure VI.43: Throughput of TCP Reno

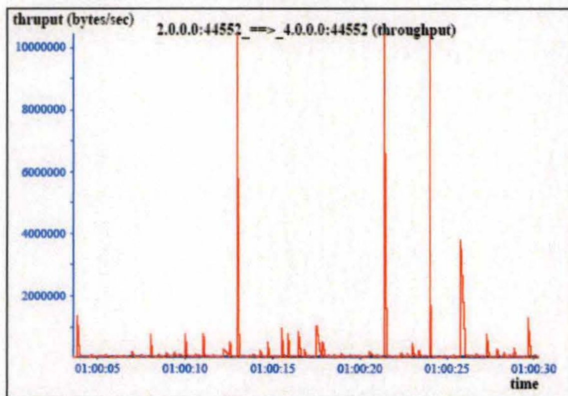


Figure VI.44: Throughput of TCP New Reno

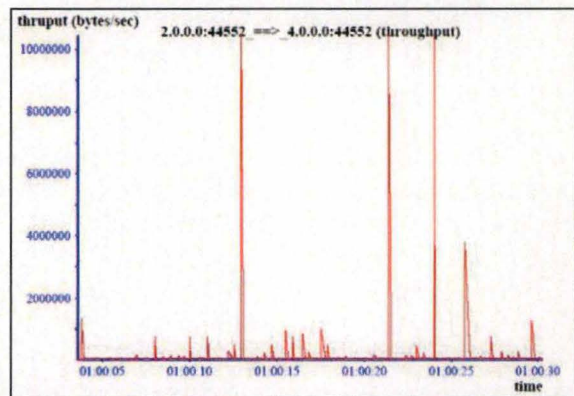


Figure VI.45: Throughput of TCP Sack

As you can see, *figures VI.4-7* are exactly alike. In order to find an explanation, let's make an assumption: *The only thing that makes those versions (Tahoe, Reno, New Reno and SACK) different is from the congestion algorithm point of view. So if the instant throughput remains identical (from one version to another) no matter what happens, it can only be explained by the fact that the improvements between versions (selective acknowledgment, fast recovery ...) aren't used at all in this context of simulation.* Moreover, the context is rather simple: given the fact that buffers can't be overflowed and LL is fully reliable, no drop (due to congestion or damage) can ever occur. So about congestion control, either:

i. The CWND keeps growing until it reaches its upper boundary, or,
ii. Retransmissions could happen because of spurious timeouts.

Finally, some conclusions:

i. The use of LL recovery mechanisms has a great impact on TCP's performance, especially from throughput point of view.
ii. Requiring LL retransmissions as long as the packet is not well received may be harmful in some situations. Indeed, after a while, interactions between ARQ mechanisms at LL and transport layer will start to occur. Hence, care should be taken to set a threshold to the maximum number of LL retransmissions.
iii. Providing at in-order delivery service may also be damageable. It can happen, mostly when the simulations are done in a context of high BLER, that several TCP packets perfectly received, are still waiting inside the RLC layer for the correct retransmission of one RLC segment belonging to a previous TCP packet. The complete receipt the TCP packet with the lowest sequence number, present in the RLC buffer will thus generate the sending of a TCP packet burst. This increases the probability of seeing a part of those packets dropped somewhere else on the Internet because of congestion.
iv. Finally, it may be important to set a buffer boundary regarding to the bandwidth capacity of the path. So far, we have met situations where the BDP was so high that TCP congestion algorithms became rather inefficient.

Queue Level

Queue size is a complex problem. TCP's congestion control algorithm tries to adapt constantly its CWND to the alleged capacity of the network. Increasing the network's queues means increasing the network's capacity and by the same way, the BDP of each path crossing the network. We will highlight the fact that TCP, in order to be fully efficient, requires for a certain amount of buffer capacity. In context of a fluctuating bandwidth, higher buffer space may prevent the path from going empty. Indeed, we will see that buffers have the ability to smooth network fluctuations. On the other hand, too many available network resources may affect the experienced RTT. When LL is fully reliable a large enough queue size, preventing any buffer overflow, ensures that packet drops (caused by congestion or damage) will never occur. This may seem very appealing from throughput point of view. In fact it simply prevents the TCP congestion algorithm from doing its job. This leads the CWND to an absolutely unadapted size with regard to the connection's needs. As a result, the RTT average will get abnormally inflated. We will finally highlight that a trade-off is required between security, i.e. the amount of outstanding data, and connection efficiency (especially in terms of delay). This trade-off may depend on the type of data flowing over the connection.

Regarding the queue size analyses, we have based our conclusions on an average of five simulations. Seven queue sizes have been considered: 2, 4, 6, 8, 10, 15, and 50 packets. Theses queues were ahead of a UMTS LL, which provided different bandwidths: 60, 120 and 240 kbps. Throughout all those simulations, the considered BLER values were 0 and 10 %. For every situation, the UMTS LL ability was enhanced by an ARQ mechanism. TCP packet sizes of 500B and 1,500B were chosen. Finally, the TCP version was a Tahoe version and the file transferred over the UMTS segment had a size of 1MB.

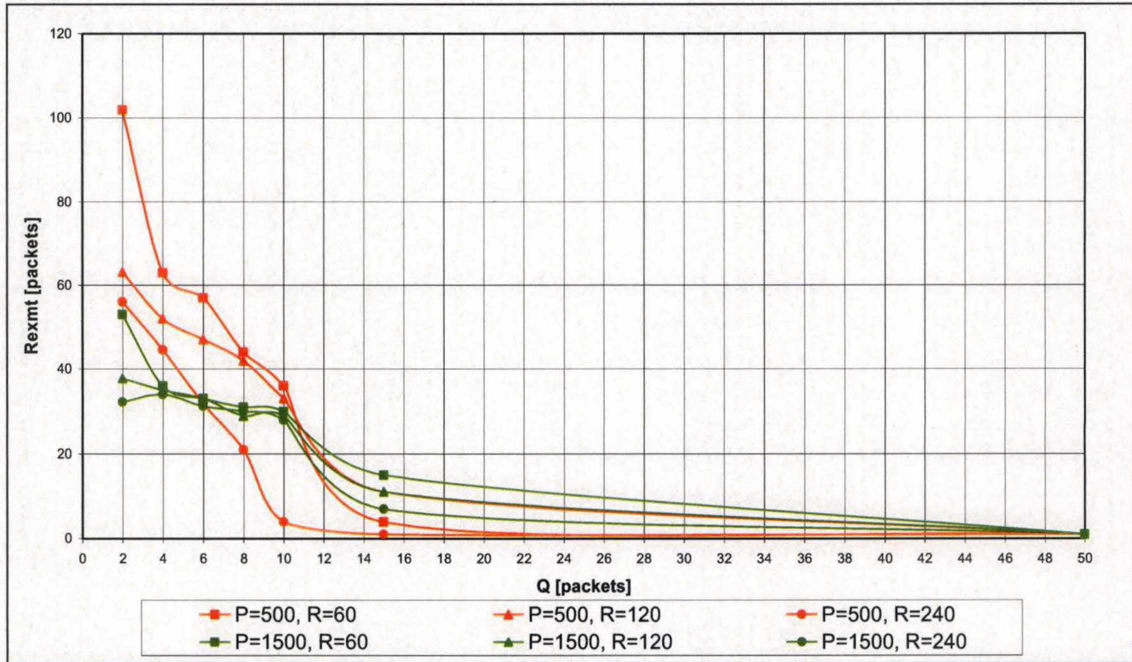


Figure VI.46: Number of retransmitted packets as a function of the queue size, BLER=0%

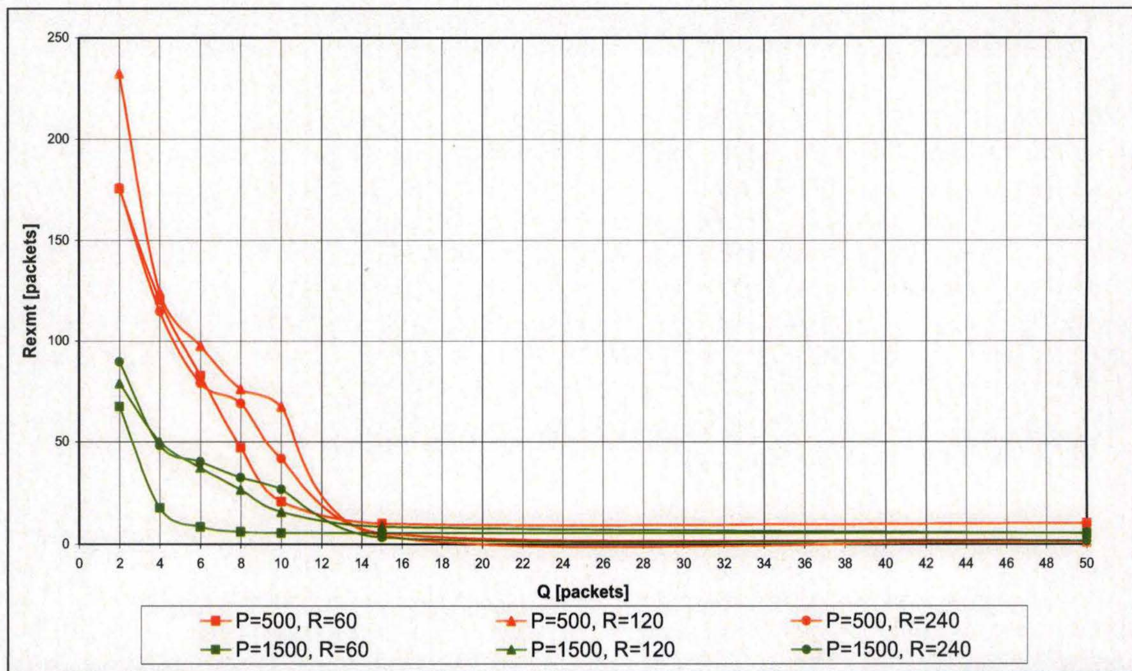


Figure VI.47: Number of retransmitted packets as a function of the queue size, BLER=10%

Figures VI.8 and VI.9 depict the variation of the number of retransmitted packets according to the queue size. Given the fact that those simulations were done with LL enhancements, every error could only have occurred because of buffer overflow. So, it is obvious why the queue size can have an impact on the number of retransmitted packets. In fact, from a certain point, TCP's retransmissions can nearly

be avoided. Of course, those graphs don't yet give enough information to state a relation between queue size and throughput. Even if LL was assured to be fully reliable, simulations with BLER value of 10% lead to more retransmissions than with a BLER of 0%. This might be because with a BLER of 10%, the BDP is more fluctuating. With an ARQ mechanism set up at LL level, the available bandwidth at time t_{i+1} depends not only on the provided bandwidth but also on the amount of data damaged at time t_i. Whether or not the provided bandwidth remained stable (for those simulations), the retransmitted data is more randomised. It can be assumed that TCP has more difficulties in reaching the network's equilibrium. In that context, bigger buffers may further smoothen this oscillation feature. Along with the growing queue size, an increase of the RTT average can also be expected. Indeed, when the rate at which the data is transferred is kept at the same level (whatever buffer capacity), the average time spent by each packet inside the queue has to be longer. It is worth mentioning that even if a bigger queue size tends to reduce the number of retransmissions, as shown in *Figures VI.8 and VI.9*, it does not yet give any information about the average queue size used during the simulation. This is precisely what we need to evaluate the RTT.

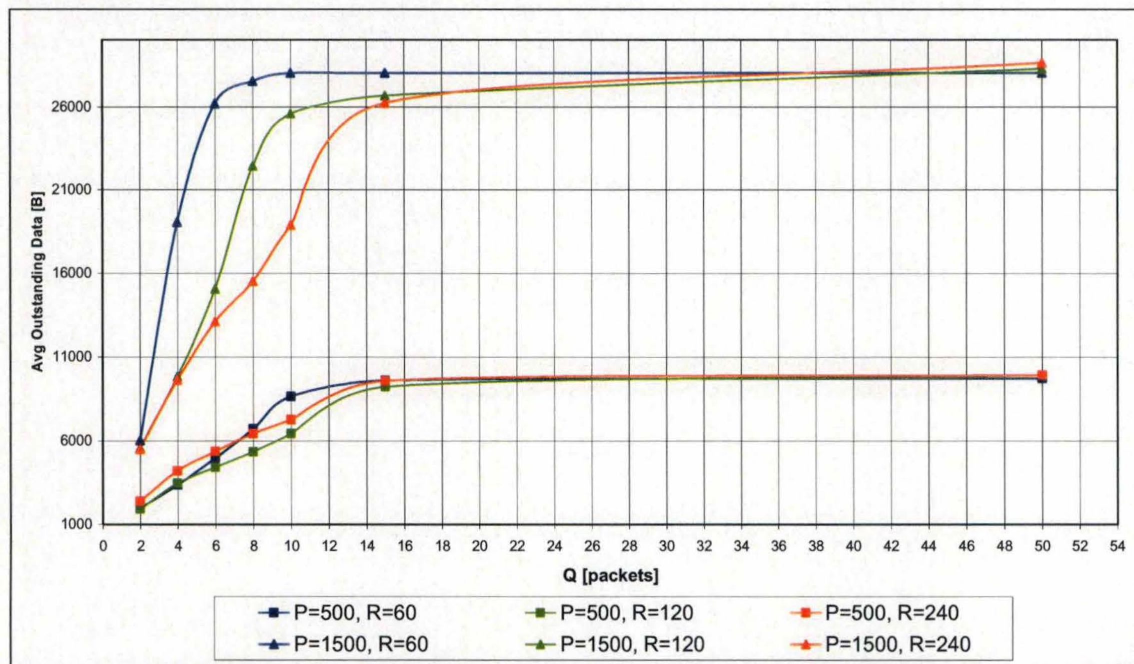


Figure VI.48: Average outstanding data as a function of the queue size, BLER=10%

Figure VI.10 shows the average outstanding data according to the queue size. It represents the amount of data already sent but not yet acknowledged. This kind of information can be of use to estimate the average size of the CWND. We hereby note that generally speaking, the average amount of outstanding data is much higher when TCP packets of 1,500B are used rather than 500B ones. In order to explain, we must keep in mind that the size of the queues is expressed in terms of packets. Using 1,500B packets enlarges further the BDP than the use of 500B packets. Moreover, it can be observed that, whatever the bandwidth is, this gap keeps growing rather exponentially up to the size of 15 packets to finally become more stable further on. This comes from the fact that, like every window based protocol, TCP limits its maximum outstanding data by setting a window boundary. This boundary represents the maximum amount of data that can be kept waiting inside the sender's buffer. Under NS2, this boundary is set by default at 20 packets. Without this boundary, the CWND size would have kept growing all along the

simulation.

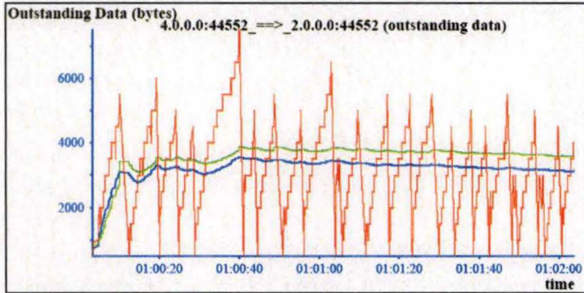


Figure VI.49: Evolution of the outstanding data, q4

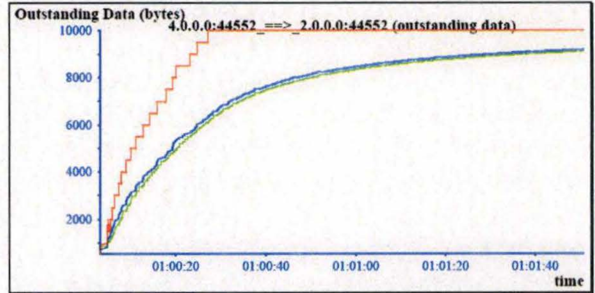


Figure VI.50: Evolution of the outstanding data, q50

Figures VI.11 and VI.12 depict the evolution of the outstanding data throughout the simulation. In both cases (left and right figures) the transmission rate, the packet size and the BLER value are set at 60kbps, 500B and 10%. On the left side, the queue is set at 4 packets and on the right side, the queue is set at 50 packets. The Red line represents instantaneous outstanding data; the Blue line tracks the average outstanding data up to that point and the Green line tracks the weighted average of outstanding data up to that point. On the right, it can be seen that the TCP congestion control algorithm is not triggered at all. The CWND just keeps growing till its upper boundary. In this scenario, it can be assumed that the delay is dominated by the waiting time in the queue rather than by the transmission time. This underlines a TCP characteristic: by always trying to fill in the network capacity, TCP can send much more data than useful to fulfil its best throughput.

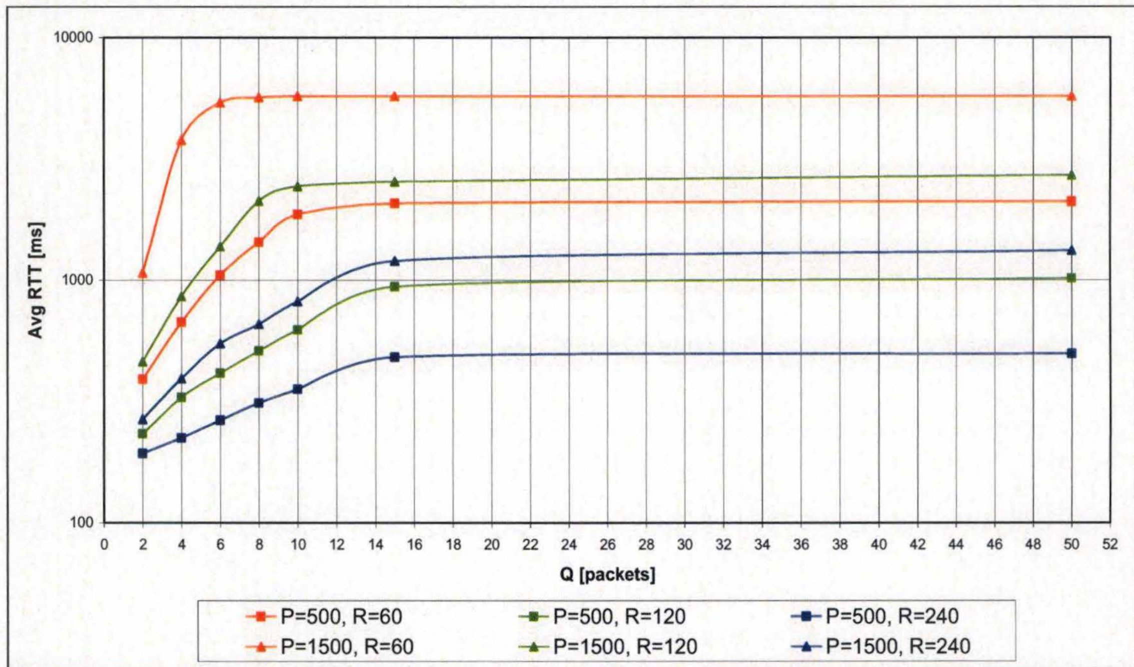


Figure VI.51: Average RTT as a function of the queue size, BLER=10%

Figure VI.13 shows the RTT average according to the queue size with a BLER value of 10%. Those

figures come strengthen our previous assumption. Indeed, it is obvious, to some extend, in every situation the average RTT keeps increasing along with the queue size. In the worst situation (TCP packet size of 1,500B and an available bandwidth of 60kbps) the RTT average goes up to 5,700ms. As a conclusion this graph really shows how a too large queue size can have a significant impact on the RTT.

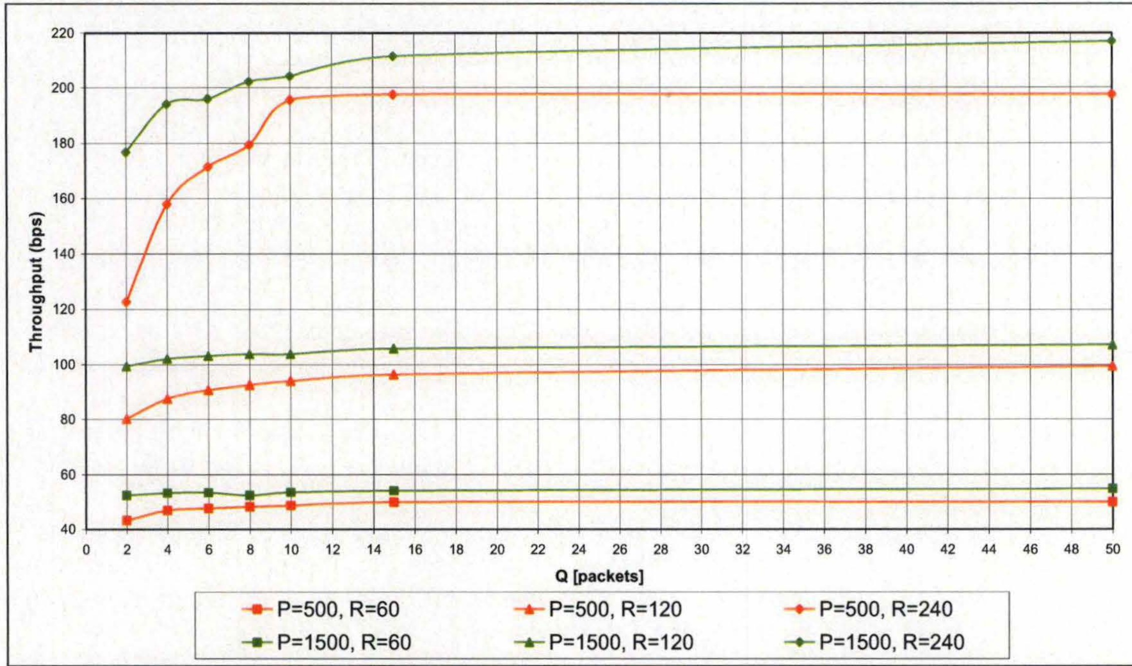


Figure VI.52: Throughput as a function of the queue size, BLER=0%

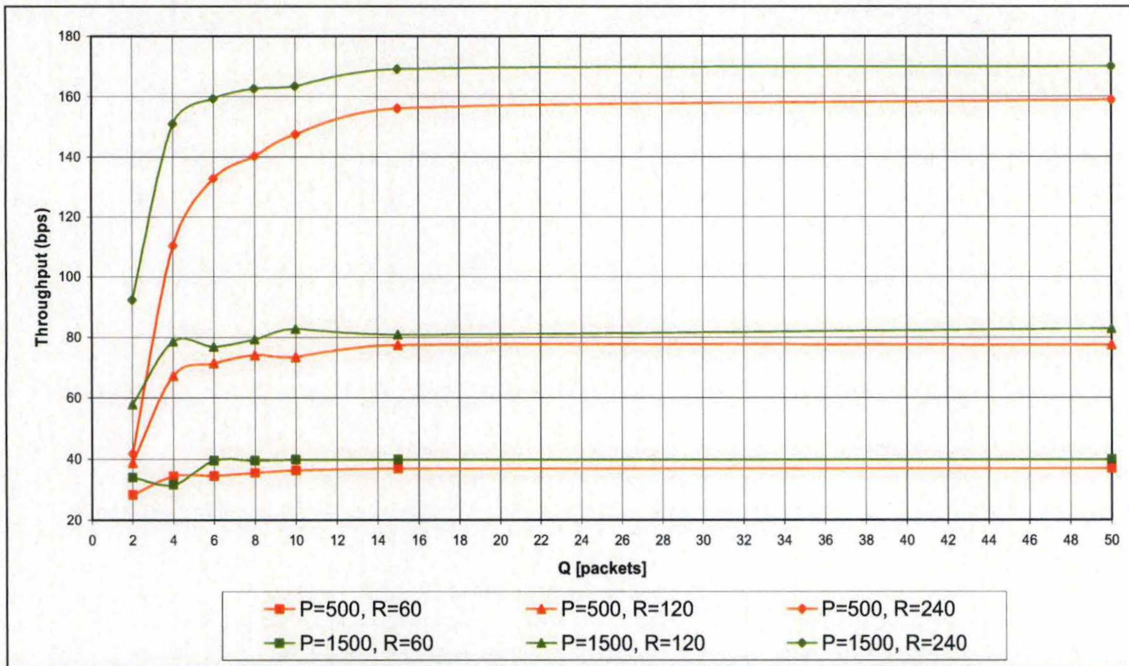


Figure VI.53: Throughput as a function of the queue size, BLER=10%

Figure VI.14-15 depicts the impact of the queue size on the throughput. It may be seen that simulations done with a BLER value of 10% need slightly more buffer's capacities in order to reach the optimal throughput than the ones done without any transmission errors. We can assume that in context of LL retransmission, the resulting BDP got enlarged. Basically, that results from the fact that the retransmission of a short packet's segment (in fact, RLC divides packets into small segments) can delay several TCP packets (if LL enhancement mechanism provided an in-order delivery service). Therefore, we can assume that retransmissions have far more impact on enlarging the delay than reducing the bandwidth. As an example, we set up our module to split TCP packets into 60B RLC segments. Moreover by default our ARQ times-out after 60ms, meaning that even the retransmission of one RLC segment, can delay a packet by 60ms. This shows why the BDP got increased when retransmission occurred on LL level. And finally, we must keep in mind that in order to reach a CWND of W packet, a minimum network capacity of W / 2 is required. All of this highlights how ARQ mechanism can be an acute problem.

Another remark can be made on the fact that, as a rule, the throughput remains higher with packet sizes of 1,500B than with packet sizes of 500B. That can be explained by the fact that large packets enable the CWND to adapt in a quicker way to the available BDP. A second part of the explanation has to be found in the amount of data used by the IP headers, which are proportionally smaller with packets of 1,500B. The part of "useful information" is thus increased and by the same way, the overall throughput.

Finally, we can conclude by the fact that care should be taken to allocate the right buffer size to the right flow. For example, in case of a BLER value set at 10% and with bandwidth of 120bps and packet size of 500B, it does not seem so interesting to allocate a buffer larger than 4 packets. The resulting bandwidth is nearly the best that we could achieve (67.2 instead of 77.4kbps) and the corresponding average RTT is kept much lower (325ms instead of 1,015ms).

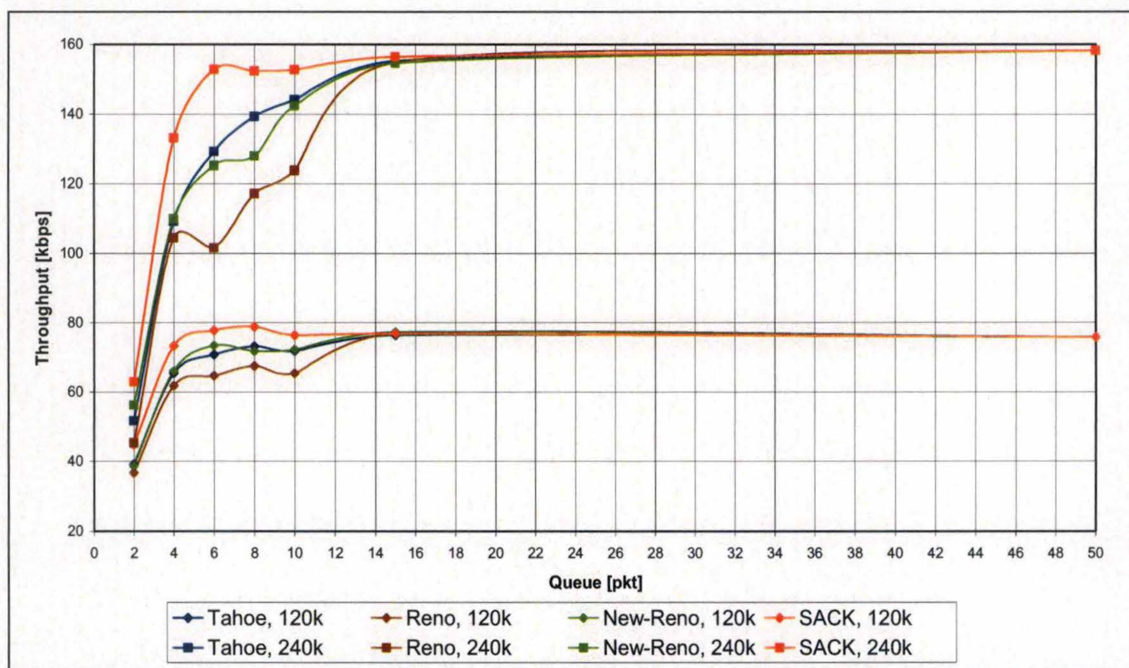


Figure VI.54: Throughput got from several TCP versions as a function of the queue size, BLER=10%

Figure VI.16 depicts the differences between TCP versions in terms of throughput. The simulation was done under a BLER value of 10%. Without any surprise, it is obvious that SACK TCP is the best candidate. It also appears that Tahoe TCP may represent an interesting option.

As a conclusion, we may say that it is essential to combine the right buffer size with the right flow. Indeed, each flow has its own requirements. Some of them can be more interactive (like shell or chat programs) and other can represent more a bulk traffic tendency (downloading or http). Accordingly, one might choose to give the priority, for one specific flow, to the throughput and for some other cases, to give the priority to the delay.

Bandwidth [kbps]	**Packet Size [B]**	**Queue Size [Packets]**	**Throughput [kbps]**	**RTT avg [ms]**
240	1,500	4	194.208	297.84
240	500	6	171.434	231.94
120	1,500	2	99.336	357.28
120	500	4	87.4944	251.44
60	1,500	2	52.3568	830.98
60	500	2	43.1904	278.84

Table VI.8: Throughput and Average RTT as a function of scenarios and queue sizes

Table VI.2 gives an example of two orientations concerning different bandwidths that might be provided by a UMTS network. One of them is more focused on the throughput but keeps concerned by the RTT. Indeed, as shown above, choosing the best solution for one criterion systematically leads the other to perform weakly.

VI.D Bandwidth fluctuation

In UMTS networks, TCP should frequently experience bandwidth oscillations (see section II.E). But the question is: what about the ability of TCP to adapt itself to those oscillations? More precisely we wanted to check out whether a sudden bandwidth decrease may lead either to some spurious timeouts [2] (and so unnecessary retransmission) or more simply drops coming from congestion (in fact, modifying the bandwidth also affect the BDP). On the other hand, we also wanted to verify whether TCP could efficiently face a bandwidth increase. Or with other words, providing a window increasing of one packet per RTT is it enough in order to adapt itself to every bandwidth fluctuation? In both cases, we will see that everything is a matter of queue size. In context of network fluctuation, queue size is the feature may allow to smooth oscillations.

To check out these assumptions we had to make a new set of modifications on our modules. Those modifications enable changing the specifications, including the sending rate, of each flow at any time.

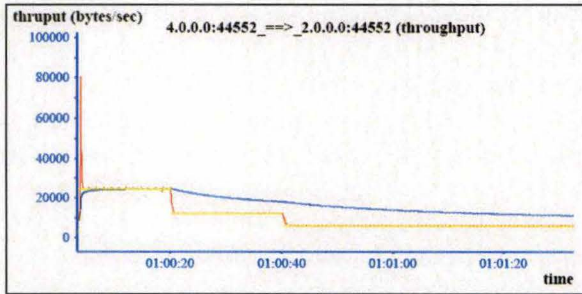


Figure VI.55: Situation I – Bandwidth decreasing

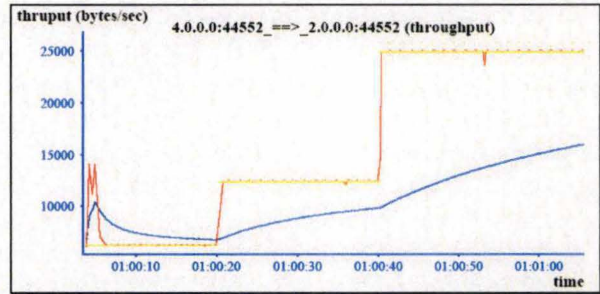


Figure VI.56: Situation II – Bandwidth increasing

Figures VI.17-18 show the 2 first situations that we were looking at for analysing bandwidth oscillation. The Yellow dots represent the instantaneous throughput defined as the size of the segment sent divided by the time gone by since the last segment was sent. The Blue line tracks the average throughput of the connection up to that point in the life time of the connection. And finally, the Red line tracks the throughput seen from the last few samples, calculated as the average of 10 previous Yellow dots. Situation one starts with an available bandwidth of 240kbps to fall at 60kbps after passing by 120kbps. For situation two, we did exactly the opposite. We started with a bandwidth of 60bps to progressively increase it to 120kbps and finally 240kbps. In the both cases, the "bandwidth jumps" were separated from each other by time period of 20s. We found out that TCP could quite well cope with those situations. Afterwards we decided to try out some more extensive situations. We tested TCP on network oscillations from 480kbps to 240kbps, 120kbps, 60kbps and 30kbps. Once more, we also did the opposite (from 30kbps, 60kbps, 120kbps, and 240kbps to 480kbps).

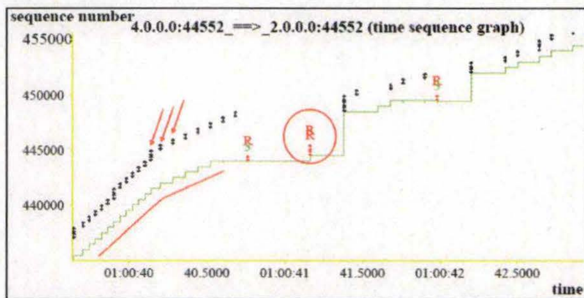


Figure VI.57: Scenario 1, q2

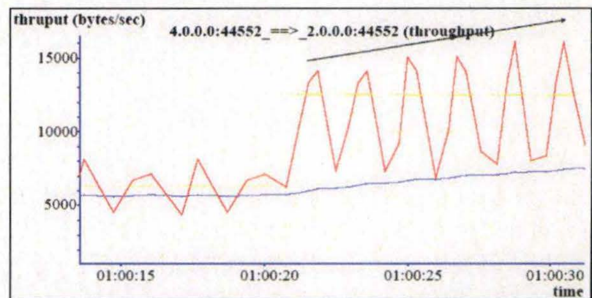


Figure VI.58: Scenario 2, q2

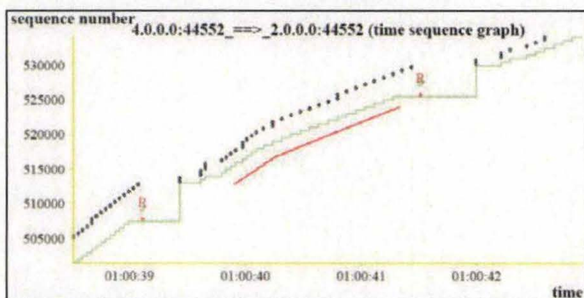


Figure VI.59: Scenario 1, q4

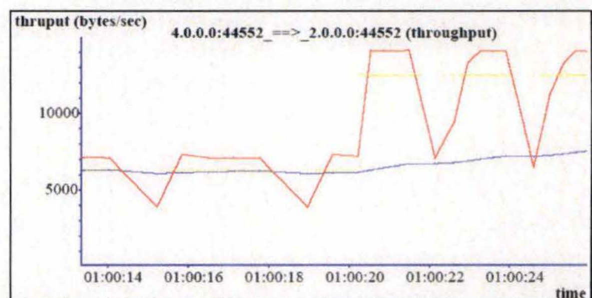


Figure VI.60: Scenario 2, q4

Figure VI.19 and VI.21 are "time sequence graphs". The Y-axis represents sequence number space and the X-axis represents time, and the slope of these curves gives the throughput over time. The Green Line keeps track of the ACK values received from the other endpoint. The Yellow Line tracks the

receive-window advertised from the other endpoint. (It is drawn at the sequence number value corresponding to the sum of the acknowledgment number and the receive window advertised from the last ACK packet received.) The Black Arrows represent the segments sent and finally, retransmitted segments are signalled by Red Arrows (R). Those 2 figures show the impact of the second "bandwidth jump" (from 120kbps to 60kbps) on retransmissions. Along those simulations, the TCP packet size and the BLER value were set at 500B and 0%. Regarding to the queue size, *figure VI.19* depicts a simulation where queue size is set at 2 and *figure IV.21* where queue size is set at 4 packets. It shows that with the lower buffer capacity, the bandwidth adjustment triggers some extra retransmissions. But on the other hand, with a queue size of 4 packets, the transition occurs smoothly.

Retransmissions triggered by a bandwidth downward adjustment start occurring for two reasons. First, the sender is in congestion avoidance but the downward adjustment causes now the CWND to be higher than the BDP. As a result, the amount of data corresponding to the ΔBDP, sent right after the adjustment comes overflowing the network. Secondly, a situation may occur when the sender experiences a downward oscillation while the Slow-Start mode. This phase usually is triggered by 3 duplicated ACKs or a timeout and mostly goes ahead until the *sshthresh* is reached. Normally, this threshold is set in a way that packet drops should happen during this phase (half of the previous CWND). But once again, bandwidth oscillation could make things different in the way that even the *sshthresh* can be higher than the modified BDP.

Figure VI.19 highlights the first causes of drop. The sender experiences the bandwidth oscillation when Congestion-Avoidance. Tree packets are drops. The first packet overflowing the buffer was sent because the sender just increased its CWND by one packet. So it might be that the CWND was already higher than the BDP (before its modification). The two following drops clearly indicate that ΔBDP is close to the amount of data contained inside the two 500B TCP packets. Having more buffers may smooth the situation by decreasing the probability of having a buffer capacity already full before the oscillation occurs. By that the BDP adjustment can be directly compensated by more buffer occupation. This is exactly the situation highlighted in *Figure VI.21.*

Figures VI.20 and 22 depict TCP facing the first "bandwidth jump" (from 60kbps to 120kbps) of scenario two. Again, we may observe that the registered oscillation shows that the connection performs better with a queue size of two packets than with a queue size of four packets. That comes from the number of packets waiting for transmission inside the buffer. Those packets represent "extra outstanding data" supplying to the one needed to fill the transmission medium. After BDP enlargement, those data are directly used to avoid network idle time. In other words, the bandwidth oscillation lowers for a short-term period the queue occupancy rather than the use of the transport mean.

So far, our conclusion is that TCP can deal quite well with bandwidth oscillation close to the one happening in situation one and two. Even if it is true that with queue sizes of two, the sender experienced some difficulties facing the oscillations. A queue size of four packets, on the other hand, allows to bypass the situation without any problem. One might argue that the previous paragraph advices using a queue size of 2 packets when the provided bandwidth is precisely 60kbps and as it has been shown, this size of queue might be inadequate to face oscillations. One solution could simply be to anticipate the oscillation by making an adequate queue adaptation. One might also point out that those scenarios rendered only short oscillations. To verify whether the oscillation problem is only a matter of queue size, we checked out TCP in tougher situations.

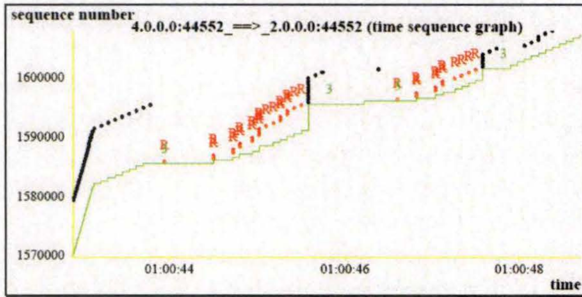


Figure VI.61: Scenario down:460k-57k, q=6

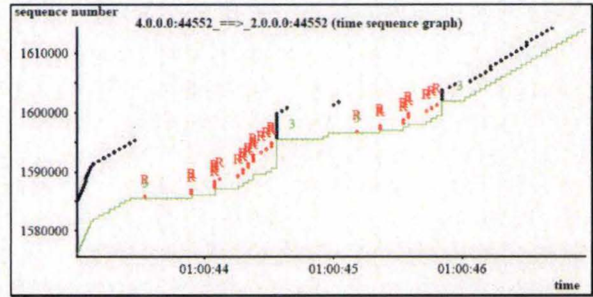


Figure VI.62: Scenario down:460k-112k, q=6

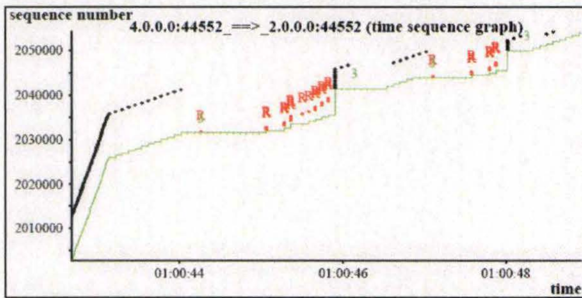


Figure VI.63: Scenario down:460k-57k, q=10

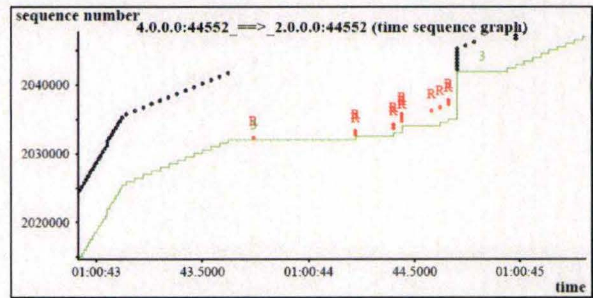


Figure VI.64: Scenario down:460k-112k, q=10

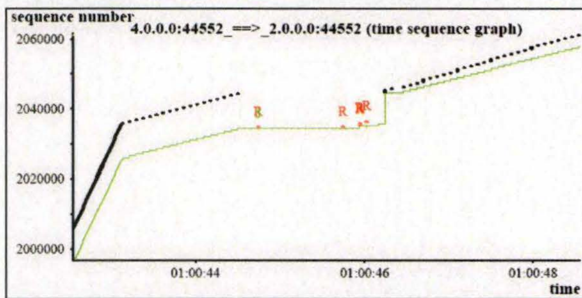


Figure VI.65: Scenario down:460k-57k, q=15

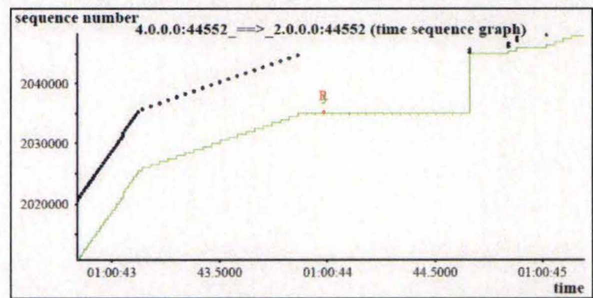


Figure VI.66: Scenario down:460k-112k, q=15

Figures VI.23-VI.28 depict the "time sequence graphs" of two situations where TCP is facing a "bandwidth jump" from 480kbps to 120kbps and from 480kbps to 60kbps. Those simulations were done with a BLER value set at 0% and TCP packets size of 500B. Generally speaking, we observe that, irrespective the queue size, TCP performs better with the bandwidth oscillation leading to a downward adjustment of 120kbps than the one leading to 60kbps. This is simply because the ΔBDP is lower in the first case than the other. Moreover, it can be seen that for both scenarios, TCP faces the situation with better results when the queue size is higher. For example, *figure IV.28* depicts a situation where retransmissions are nearly avoided. That emphasis our conclusion that a dynamic queue size handling may be a solution to cope with bandwidth oscillations in UMTS networks.

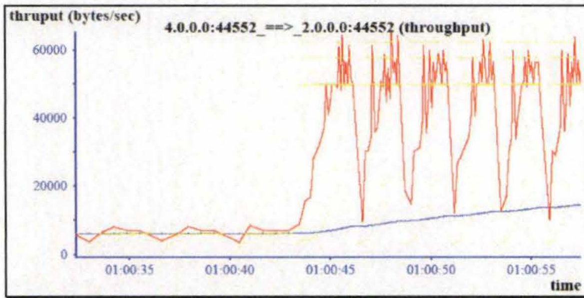


Figure VI.67: Scenario up:57k-460k, q=4

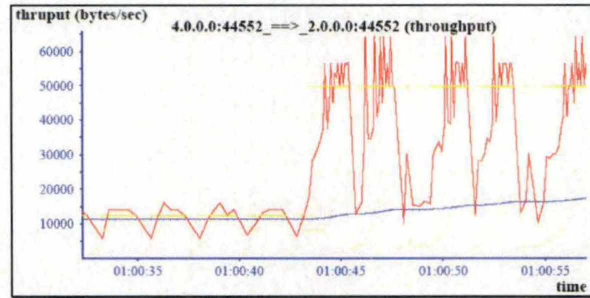


Figure VI.68: Scenario up:112k-460k, q=4

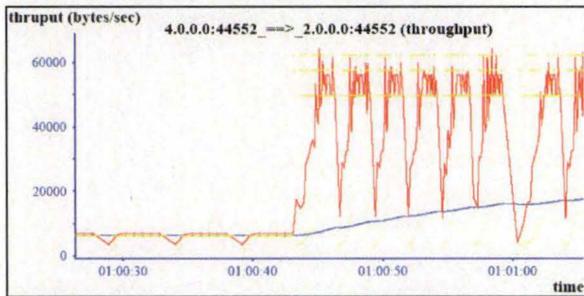


Figure VI.69: Scenario up:57k-460k, q=6

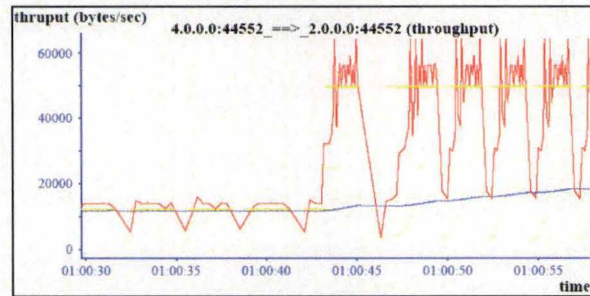


Figure VI.70: Scenario up:112k-460k, q=6

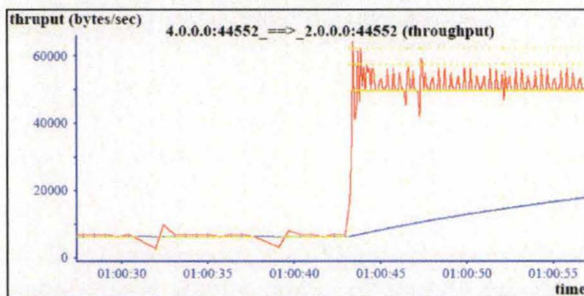


Figure VI.71: Scenario up:57k-460k, q=8

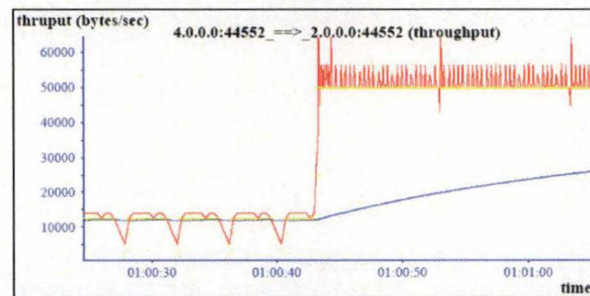


Figure VI.72: Scenario up:112k-460k, q=8

Figures VI.29-VI.34 depict the "throughput graphs" of TCP facing an upward adjustment of the bandwidth. The first scenario provides an oscillation from 60kbps to 480kbps and the second one, from 120kbps to 480kbps. Again the BLER value is set at 0% and the TCP packet size is at 500B. Regarding the two first figures (VI.29 and VI.30), where both queues are set at four packets, we can see that after the oscillation occurred, TCP has some major difficulties to use all network resources. According to us, there the buffer capacity was simply too low to back such a bandwidth. This underlines the fact that every "bandwidth-jump" should at least be followed by an adaptation of the queues capacity. *In figure VI.31 and VI.32*, a new element appears: in both cases the throughput reaches quite quickly the new network capacities. But shortly after the jump, again TCP experiences some difficulties to fill the network. It could be that the amount of data waiting inside the queue wasn't enough for backing the sender until its window reached the right level. This problem disappears at *figures VI.33 and VI.34*.

As a conclusion, regarding TCP facing a downward bandwidth adjustment, we can bring up the fact that never in the experimented situations, bandwidth oscillation has led to spurious timeouts. According to this, our conclusion is that optimal performance can be reached by providing a dynamic buffer management. On the other hand, with respect to TCP dealing with an upward adjustment of the bandwidth, things are less clear. In that context, in order to reach optimal performance, TCP needs

minimum buffer occupancy with intend to compensate the too slow CWND growth. Of course, this needs some time before the oscillation.

VI.E Error rate fluctuation

We described earlier the features that characterise a wireless network like the UMTS network (see chapter II). Simply due to its air interface, many errors caused by interferences occur during data transmissions. Up to now, we kept this Block Error Rate constant throughout the simulations. However, the BLER increases when the Signal to Noise Ratio drops [7]; this obviously means that when the interference level increases, the BLER increases also. The mobile equipments or UE's, have to cope with three main types of interference:

- the Intra-cell interference, caused by other connections inside the same cell;
- the Intra-cell interference, caused by connection from other neighbour cells;
- the background noise.

In order to simulate a realistic UMTS network, where mobility is one of the main characteristics, these factors fluctuate during the lifetime of a connection, and thus influence the BLER. For example, when users enter the cell, not only the bandwidth may change: but also the interference level will increase. Therefore, we will have to cope with a higher-than-expected BLER. Another factor is directly linked with mobility offered to the users of a UMTS network, is to be found in the layout of the topography. A connection could suffer from a sudden increase of the BLER simply due to the movements of the user. Someone moving through a tunnel or leaving momentarily the area covered by the network, will suffer momentarily from a very high BLER, possibly even up to 100%.

In order to study these fluctuations of the BLER, some modifications had to be applied to the UMTS module, enabling us to alter the BLER value during the entire lifetime of a connection. The following scenarios will focus on the sudden increase of BLER, caused by the surrounding topography. While transferring via FTP a 1MB file, over the UMTS connection, we introduced a 100% BLER, once at two stages of the transmission, in order to observe the way TCP tries to adapt itself to the "black-outs" and once during a 50 seconds period permitting us thereby to see the effect of the exponential back-off on the TCP's reaction time.

Figure VI.35 depicts the sudden increase of the BLER value at t = 5 seconds, for 15 seconds. Another one occurred at t = 50 seconds and lasted 10 seconds. The bandwidth was at 120 kbps. Between the 100% error phases, we assumed an error-free connection. In order to read the time on the X axis, the four last digits have to be considered. The last two express the seconds; the following express the minutes. The digits "01" preceding the time have to be discarded.

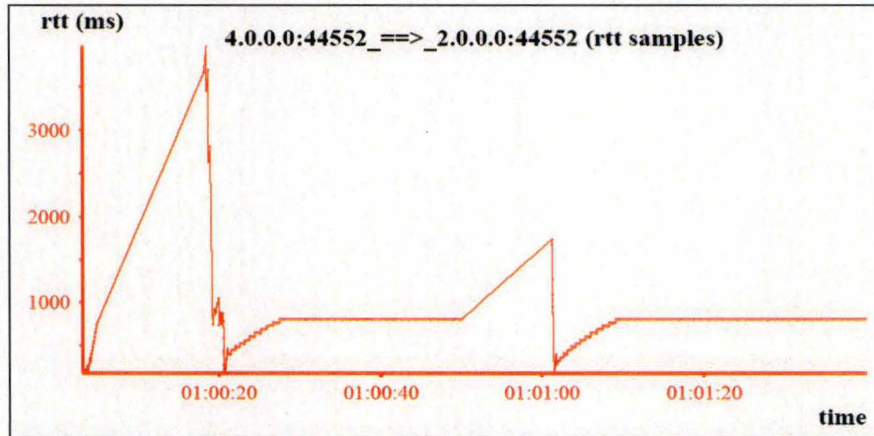


Figure VI.73: RTT variation.

This RTT curve has been drawn using results provided by the TCP Tahoe version with a TCP packet size of 1,500B and an unlimited buffer. It is impossible not to notice the expected sudden increase of the RTT during the lifetime of the connection. The RTT starts a steep linear increase at around t = 5 s, until the drop just before t = 20s, the time around which an error free connection is re-established. The interesting feature that can be noticed on this graph is the time it takes TCP to start using the re-established connection at its full potential. Further we will see more explicit graphs of this phenomenon.

On *Figure VI.36*, plotted from the same trace-file, the situation becomes clearer.

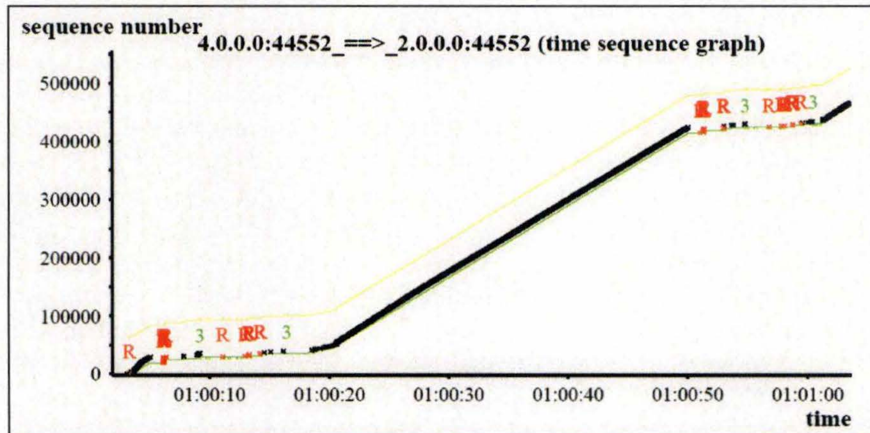


Figure VI.74: Time sequence graph.

During the period going from t = 20 seconds to t = 50 seconds, we notice that no retransmissions occurred. This period corresponds to error free transmission, where TCP, once reached the limit of bandwidth available, constantly sends the same amount of data. In that case, packets can only be dropped when a buffer is full, but with a big enough queue size, every packet can be stored and will reach the receiving end before it even can time out.

But let's go further into detail on the first part of this "time sequence graph".

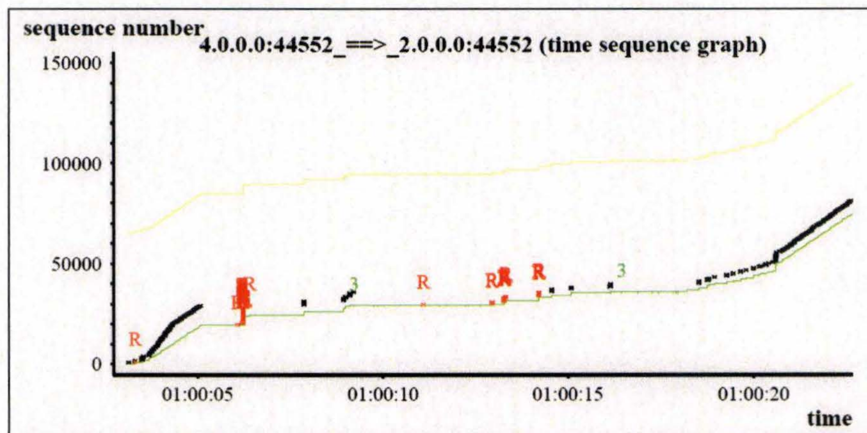


Figure VI.75: Time sequence graph zoomed in [0 s, 22 s].

Here we notice the slow start phase of TCP (from t = 0 s to t = 3 s), with its exponential increase followed by a linear one once the threshold (set by the first drop) has been reached. The moment the Green line flats-out, the sender does not receive any new ACKs, the timer expires, and the sender starts resending the whole window of segments. However, at the supposed 100% BLER during that 15-second period, slight increases of the Green line can be observed, which means that some packets still manage to go through. This artefact originates from an unexplained anomaly of the UMTS module on our NS2 simulator. Even with a 100% BLER, some data still succeeds to pass through, just before the connection closes. That's why we must not forget that these graphs are used to study general tendencies and not only focus on a detailed packet level study of a connection lifetime.

A more interesting approach is perhaps to see how the duration of the "black-out" is interfering with TCP's normal behaviour. As explained in chapter IV, without the timestamp feature, TCP discards retransmitted packets for the RTT calculation. This RTT, used to calculate the RTO will increase during the first stages of the connection. Once we start to experience some time-outs, TCP will back-off, by doubling the RTO value, each time it expires. TCP will proceed that way until it reaches, the usually 64s upper boundary for the RTO. But how should we interpret this behaviour? It is true that backing off during a black-out, is not bad. It is no use trying to transmit a packet constantly, when it will obviously get dropped. The downside of this way of proceeding is that once the RTO reaches important value, near or even equal to its maximum value, it will take TCP such a long time become aware of the re-established connection. The worst case scenario that is when, due to a long black-out, the RTO reaches its limit, a time-out occurs. The RTO is kept constant, but then the connection "comes" back. It that case, we might nearly wait the whole 64 seconds before restarting to transmit.

Lets see in the following graph how TCP Tahoe will react to a 50s period of 100% BLER.

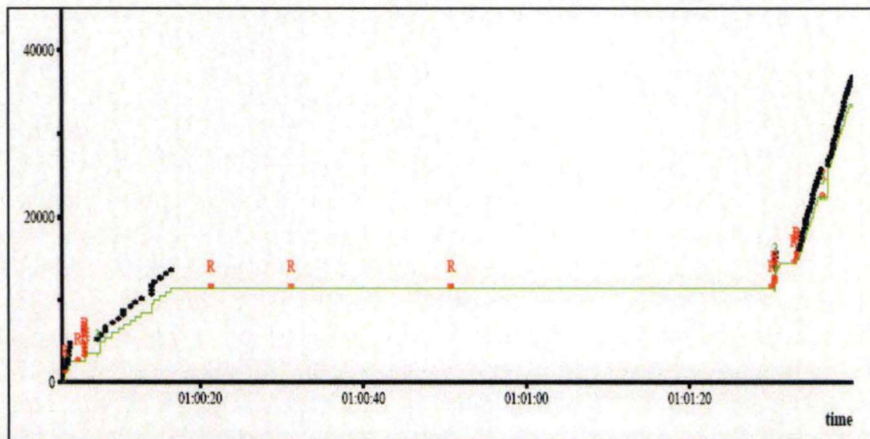


Figure VI.76 : Time Sequence Graph of TCP Tahoe during a 50s black-out.

During this simulation, the queue sizes are kept at 2 packets, mainly for highlighting TCP's response time. With a too large buffer, packets could still be sent, but remaining in buffer, and clouding the moment where the cut-off occurs. After that time, still unaware of the connections state, Tahoe keeps sending packets to fill the pipe. Where the Green line starts to remain horizontal, the sender does not receive any ACKs for the segments waiting in its CWND, a time-out is experienced, TCP retransmits, and the RTO is doubled. At around 22s, we can observe a retransmission. It occurs again at around 32s, this time the RTO doubles again in size. Being at 5s after 18s of simulation, it doubles to 10s, at 22s of simulation, to reach 20 seconds at 32s since the start of transmission. At 52s, the RTO has already been inflated to 40s, meaning that the next retransmission attempt will occur around 1m32s. But the connection is reinstated at around 50 seconds, meaning that approximately 42s of precious connection time is wasted! This is nearly the worst case scenario, where TCP is unable to evaluate the correct state of the connection and thus waits unnecessarily before starting a next retransmission attempt. Another very important TCP feature not to forget is that TCP will only try for 13 times to retransmit a packet. After that, it supposes the connection lost and closes it completely. For obvious reasons, that is of course something we absolutely want to avoid.

The right thing to do is not easy to figure out. We have to consider the following two statements:

- Reducing the maximum RTO value will speedup TCP's reaction time to the connection re-establishment but could generate useless retransmissions, draining the battery, and reducing the time before TCP will close the connection.

- By keeping the same RTO value, we spare more the battery, but also slow down TCP's reaction time.

Is it possible to give one correct way of proceeding? No, probably not. Changing the RTO's maximum value, may increase TCP's performance on the UMTS network. But once beyond it, in the wired network, we still have to consider an equal share of available resources between all TCP connections. The ones with a smaller RTO would take an advantage on the others, and maybe overload still more an already congested network. By simply considering the wireless part, a smaller RTO could be interesting, especially because the average file size transmitted over the UMTS network is quite small, and the slow reaction time of TCP will be that much bigger in proportion of its transmission time. For simulation purposes, we used a quite large, 1MB file in order to have enough time to for a correct

analysis of the situation.

This exponential back-off algorithm TCP uses, is implemented in all the TCP versions. The only difference that could be noticed is the way TCP expands its CWND, in the starting phase of the connection. There where TCP Tahoe will time-out and start again with a CWND of one, versions like Reno or New Reno will use Fast-Retransmit and Fast-Recovery when possible. This will simply keep the date flowing during that short period. When compared to the whole transmission time, this could thus be neglected. Of course, it is important to mention that before and after the black-out, we had a nearly error free transmission. During simulations where the BLER was already at 10% before rising to 100%, the more sophisticated versions like Reno or still better New Reno and SACK, not only performed in a more efficient way on the number of packets sent. But, they are less likely to time-out as Tahoe for example, because their RTO will not increase as much. This difference can influence the overall performance when the black-out is not to long. Once exceeding 30 seconds, the RTO will be large enough to cancel the effects of the more advanced TCP versions.

Therefore we may say that:

- The initial BLER before the time-out will only slightly influence the over-all performance of TCP during the entire transmission time. It could simply set the RTO to a greater value, before starting to double it. This is even more true when the "out of reach" effect lasts for a longer time, like around 30s or more.

- The TCP version will not affect that much the TCP's performance. During a complete "cut-off", only a time-out will notify the sender that a packet got lost. All the more complex recovery algorithms are of no use in those cases.

So, an interesting remaining approach is the buffering capacity of the network. Also, we will take the TCP packet sizes into account. Is it possible to tune them in order to make this transport protocol more efficient when confronting those scenarios? How will this affect the throughput or RTT?

Once more, we will be "cut-off" twice. For the first time during 6 seconds, this 4 seconds after the connection establishment. The second time will take place from the 20th second up to t = 25s.

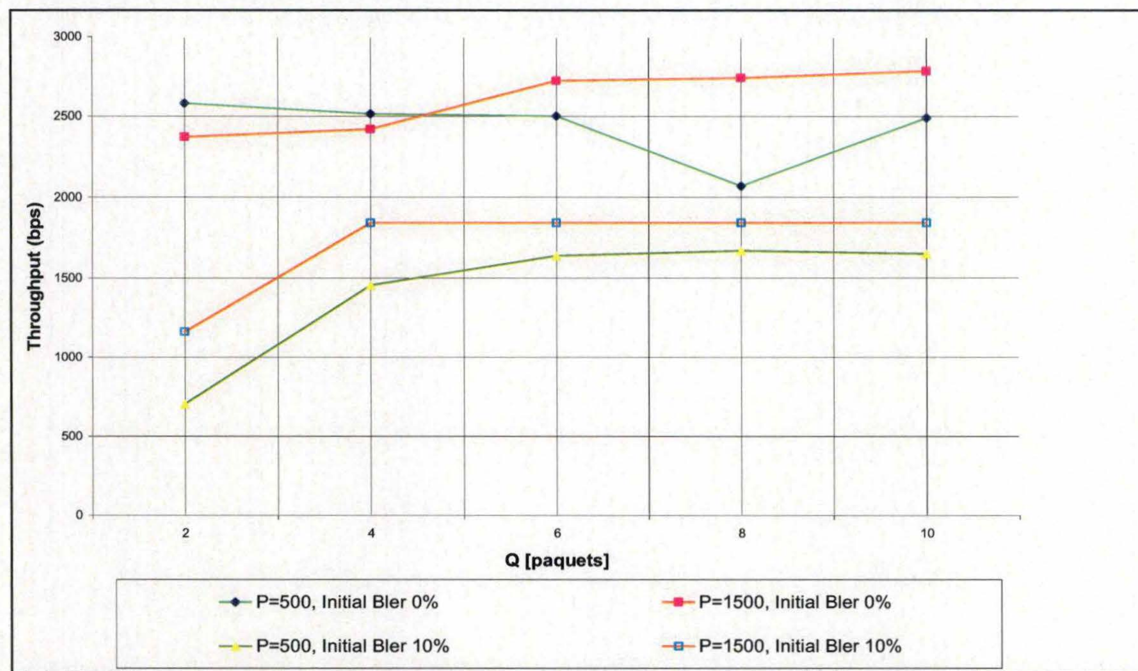


Figure VI.77: Throughput as a function to the queue size.

Again the throughput remains mostly higher with packet size of 1,500B. The same reason can be given as for the constant BLER, but it is interesting to notice that the throughput stabilises even more quickly around a queue size of 4 packets, than in the case of a constant error rate. This can teach us that the assumptions made about the value to set to the queue size for the constant bit rate, remain plausible if the error rate fluctuates. This seems quite obvious, since once the connection re-established; TCP will sooner reach the limit of the available resources of the network. Yet again, the queue size is expressed in packets; a comparison on effective queue sizes is thus not possible once more.

This is confirmed by the following graph showing the average RTT.

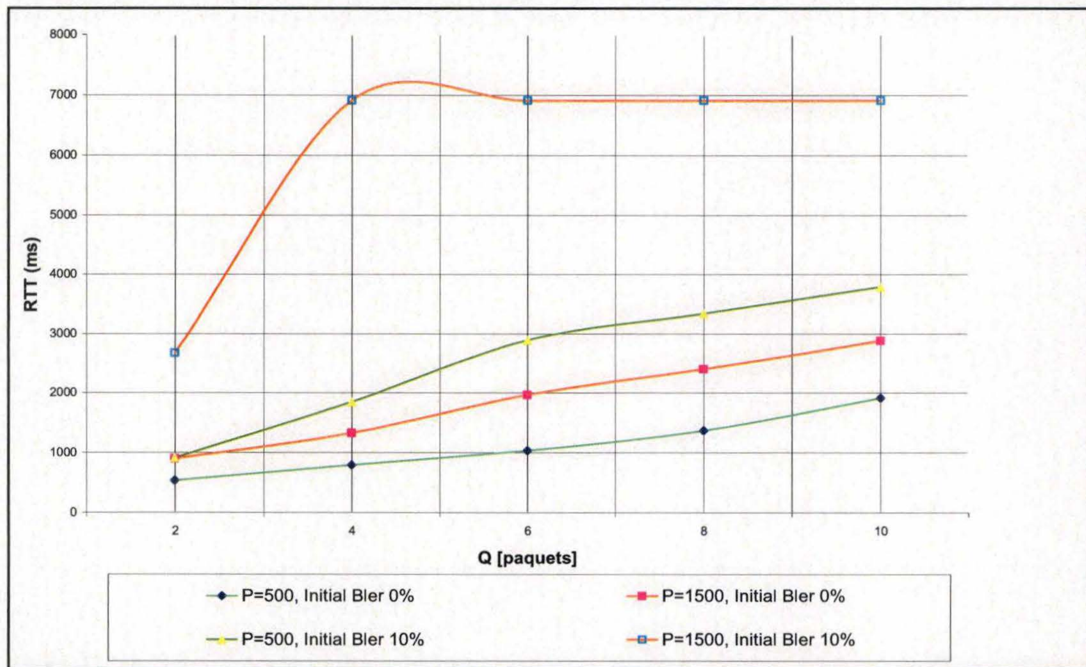


Figure VI.78: RTT average as a function of the queue size.

Like before, this leads us to the same conclusion: the average RTT keeps increasing along with the queue size but tends to stabilize slowly. Packets simply remain longer in the queues, making their stay in the network longer. It is again useless to have a to important buffer capacity in the network. Once the stay in the buffers exceeds the time-out value, this will simply trigger unnecessary retransmissions. Once the time the packets remaining in the buffers, exceeds the time-out value, unnecessary retransmissions will be triggered.

Consequently, on queue size basis, the results shows that the assumptions made for a constant error rate, are not far fetched when applied to a fluctuating BLER. This is the case for Tahoe TCP.

VI.F Proposed Solutions for TCP

Finally, it is worth mentioning, that some measures can be taken to get TCP more suited to wireless networks. The general idea of this part is to give a set of recommendations for configuration parameters of the protocol stacks that will be used to support TCP connections over the UMTS network.

As explained afore, TCP does not perform well when many transmission errors occur inside the network. LL recovery can be a partial solution to improve its behaviour in high error rate context. Unfortunately the general problem of TCP flowing over a UMTS segment cannot be reduced to this simple matter of transmission error rate. Even in error-free circumstances, TCP has still to probe, as efficiently as possible, the capacity of the path between the sender and the receiver. Again UMTS

networks make things more complicated for TCP: on a UMTS segment, it has to deal with bandwidth oscillation and LFN characteristics. As a result on TCP, LTN will slow down its probing ability. Indeed, a TCP sender adapts its use of the bandwidth, basing itself on the feedback from the receiver. The high latency-delay characteristic of LTN implies that TCP adaptation is correspondingly slower than on networks with shorter delays.

The first recommendation concerns the maximum size of the sending and receiving windows. This one should be based on the BDP of the end-to-end path. Without this, it will be never possible for the sender to fill in the pipe between the sender and the receiver.

The second recommendation concerns the Slow-Start phase. In presence of high RTT, typically the one occurring in UMTS network, the Slow-Start stage may represent an important part of the connection time, especially for small files. The sender can avoid this by increasing the initial window size up to 4 segments [5]. It has been showed that it is safe on congestion collapse point of view, to deploy this mechanism [5].

Table VI.3 depicts the throughput evolution in regards to the initial window size and the bandwidth. Moreover, BLER: 1%, packet size: 540B, file size: 10KB. It may be seen, especially with high bandwidth i.e. low SF, how beneficial larger initial window size can be.

SF	Initial Windows Size [Packets]	Throughput [Bps]
8	1	8,441
8	2	16,084
8	4	18,156
32	1	8,679
32	2	9,536
32	4	10,855
64	1	5,097
64	2	5,610
64	4	5,746

Table VI.9: Throughput as a function of the SF and the initial window size.

As mentioned, changing the maximum RTO value could have some beneficial results as long as the packets are travelling on the UMTS network, but once beyond it, it is important to consider the concurrent flows over the wired network. Therefore, a TCP simply optimised for wireless networks alone is of no use in those heterogenic cases. Even if we could modify the RTO dynamically, it is impossible to foresee the time a black-out will last, and attempting to many retransmissions during an out of range moment, would unnecessarily drain the battery of our mobile handset.

Finally, the use of large segment sizes, basically equivalent to the TCP packet size, can increase the TCP ability to handle heavy bandwidth or BLER oscillations. Indeed, as the sending window is expressed in units of segment, a larger packet size allows TCP to increase the CWND [6]. This will lead to a shorter Sow-Start phase and a more aggressive Congestion-Avoidance stage. However, it is worth mentioning that in context of UMTS networks, large packet sizes are only convenient if error recovery

mechanisms are simultaneously provided.

VI.G Sources

[1] Saltzer J.H., Reed D.P., Clark D.D, **End-to-end Argument in System Design**, ACM Transactions in Computer Systems, 2(4):277-288, November 1984.

[2] Gurtov A., Ludwig R., **Responding to Spurious Timeouts in TCP**, Proceedings of IEEE INFOCOM '03, March 2003.

[3] Tsaoussidis V., Matta I., **Open Issues on TCP for Mobile Computing**, Journal of Wireless Communications and Mobile Computing, March 2002. Available at http://www.cs.bu.edu/techreports/pdf/2001-013-open-issues-tcp-wireless.pdf, Last visited: September 30th, 2003.

[4] Tsaoussidis V., Badr H., Ge X., Pentikousis K., **Energy / Throughput Tradeoffs of TCP Error Control Strategies**, Proceedings of the 5th IEEE Symposium on Computers and Communication, July 2000.

[5] Allman M., Floyd S., **Increasing TCP's Initial Window**, RFC 2414, September 1998.

[6] Inamura H., Montenegro G., Ludwig R., Gurtov A., Khafizov F., **TCP over second (2.5G) and third (3G) generation wireless networks**. RFC 3481, February 2003.

[7] Gurtov A., **Effect of Delays on TCP Performance**, Proceedings of IFIF Personal Wireless Communications, August 2001.

Chapter VII: Conclusion

In our thesis we investigated the ability of TCP to carry data over a UMTS network. First of all, we emphasise on the absolute necessity of providing LL recovery, in order to allow the utilisation of "traditional TCP versions" i.e. those that do not make a distinction between the origins of the packet drops. Indeed, among all the tested TCP versions, even though some of them implemented significant improvements, they were still unaware as to the packet drop's cause. Whether some TCP versions perform better than others, in context of several mistakes inside one CWND, drops due to damage still remained taken as hints of congestion and so led to unnecessary downward readjustment of the throughput. One might argue that increasing the LL reliability is the only approach developed in this document. Every TCP version attempting to give the sender the ability to deal with none congestion losses were effectively discarded. Our point of view about this is that according to the time needed for new versions to get spread around the world, it is more appropriated to make UMTS LL more TCP friendly than making TCP more suited for UMTS. This can be easily pointed out by the following example: even if several new TCP versions have emerged, Windows XP still uses TCP Tahoe.
Afterwards, we identified three different contexts in order to make extensive TCP simulations. Those contexts were not meant to cover the entire field of possibilities. They were merely referencing expectable situations that should commonly occur in an UMTS network. As an example: What is TCP's reaction when I am experiencing a network black-out by simply crossing a tunnel, or how will TCP behave when I undergo a bandwidth modification because of users entering or leaving the cell I'm in? These contexts of research were set up in order to evaluate on them TCP's behaviour and, if possible to help providing means for improvement. With respect to LL enhancements and the simulations made on those scenarios, we first were focused upon interactions between LL and TCP ARQ mechanisms (leading to spurious TCP timeouts). Our measurements showed that in fact this problem rarely occurred. With other words, our results revealed that whenever LL recovery is provided, TCP can perform quite efficiently. Afterwards, our observations led us to question whether the UMTS LL should take the TCP flow specifications in consideration. The outcome of our simulations permitted to conclude that beside an enhancement of the UMTS LL, the buffer storage capacity is the key element that makes UMTS LL more TCP friendly. According to our opinion, a dynamic buffer size management that takes in account the TCP's flow specifications, would sustain TCP in coping with the varying conditions, which typically characterises the UMTS networks. Of course, in our thesis we only brought up several elements that should be taken into account for an efficient buffer management.
Finally, we can propose as a future work, the development of a module providing an individual buffer management for each flow. This module should of course be able to dialogue directly with the RCC layer in order to evaluate the current UMTS network state and even anticipate some network reactions.