AN IMPROVED EIFEL ALGORITHM FOR TCP OVER WIRELESS LINKS

YU LIANG

NATIONAL UNIVERSITY OF SINGAPORE

2003

AN IMPROVED EIFEL ALGORITHM FOR TCP OVER WIRELESS LINKS

YU LIANG
(B.Comp.(Hons.), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2003

Acknowledgements

This thesis would not have been possible without the help and support from a number of people. I am immensely grateful to my supervisor, Dr. Tan Sun Teck for his invaluable support and guidance throughout my thesis work. I also thank him for his patience and understanding when nothing good seemed to happen.

I am thankful to my friend Zhu Yingjie for his useful inputs and patient listening.

Last but not the least, my family and my love, Li Nan, have always been beside me with their love and support in times of need.

Dedicated to my family and my love, Li Nan

Contents

Acknowledgements **i**

Summary **xv**

1 Introduction **1**

1.1 Motivation of Research 1

1.1.1 Spurious Timeouts 2

1.1.2 Packet Loss 4

1.2 Objectives of Research 5

1.3 Contributions of Thesis 7

1.4 Organization of Thesis 8

2 Overview of Cellular Mobile Radio Systems **10**

2.1 GPRS: General Packet Radio Service 11

2.1.1 System Architecture 12

2.1.2 Radio Interface 13

2.1.3 Protocol Stack 16

2.1.4 Quality of Service 19

2.1.5 Mobility Management 20

2.2 UMTS: Universal Mobile Telecommunication System 21

2.2.1 System Architecture 22

2.2.2 Protocol Stack 26

2.2.3 Quality of Service 28

2.3 Summary 28

3 TCP and Its Behavior over Wireless Links 29

3.1 Basics of TCP 30

3.1.1 TCP Transmission and Acknowledgment 31

3.1.2 TCP Flow and Congestion Control 32

3.2 TCP over Wireless Links 39

3.2.1 Wireless Link Characteristics 39

3.2.2 Interactions between TCP and Link Layer Retransmission . . . 42

3.2.3 Proposed Solutions in TCP 44

3.3 Spurious Retransmission 45

3.3.1 Problem Formulation 45

3.3.2 Related Works on Spurious Retransmission 48

3.4 Multiple Packet Losses 57

3.5 Summary 58

4 Eifel-I: the Improved Eifel Algorithm 60

4.1 Limitation of the Timestamp Option 60

4.2 Selective Use of Timestamps in Eifel-I 65

4.2.1 Negotiating the Use of Eifel-I 68

4.3 Responses to Spurious Retransmissions 69

4.3.1 Adapting the TCP Retransmission Timer 70

4.4 Avoiding Multiple Fast Retransmits 75

4.4.1 The Multiple Fast Retransmits Problem 76

4.4.2 BugFix in TCP NewReno 77

4.4.3 The Eifel-I-based Solution 78

4.5 Summary 86

5 Implementations 88

5.1 The NS-2 Network Simulator 88

5.1.1 Overview of NS-2 88

5.1.2 A Link in NS-2 90
5.1.3 TCP Agents in NS-2 91
5.2 Implementation of Eifel-I and Others 92

6 Experiments by Simulation **95**
6.1 General Settings for Experiments 95
6.2 A Single Spurious Timeout 96
6.3 Scenarios and Discussions 98
6.3.1 Variable Delays and Losses due to Handovers 99
6.3.2 Variable Delays due to Link Layer Retransmissions 107
6.4 Summary of Results 120

7 Conclusion and Future Work **122**
7.1 Summary 122
7.2 Future Work 125

Bibliography **126**

Appendix A. Cellular Wireless Systems **130**
A.1 Cellular Wireless Fundamentals A-1
A.1.1 Multiple Access A-1
A.1.2 Error Protection in Radio Channels A-2
A.2 Some Details on GPRS A-5
A.2.1 Logical Packet Data Channels A-5
A.2.2 Multiframe Structure A-6
A.2.3 QoS Parameters A-6
A.2.4 Mobility Management Scenarios A-10

List of Figures

2.1 GPRS system architecture 12

2.2 GPRS radio interface 13

2.3 GPRS protocol stack 17

2.4 UMTS network architecture 22

2.5 UMTS Terrestrial Radio Access Network 24

2.6 Functions of UTRAN elements 25

2.7 Functions of UMTS UE 26

2.8 UMTS user plane protocol architecture 27

3.1 Visualization of TCP sliding window 33

3.2 Visualization of slow start and congestion avoidance 36

3.3 A spurious timeout 46

3.3(a) Time sequence 46

3.3(b) Congestion control state 46

3.4 A spurious fast retransmit 48

3.4(a) Time sequence 48

3.4(b) Congestion control state 48

3.5 A spurious timeout using TCP SACK with DSACK 50

3.5(a) Time sequence 50

3.5(b) Congestion control state 50

3.6 A spurious timeout with Eifel 52

3.6(a) Time sequence 52

3.6(b) Congestion control state 52

3.7 A spurious timeout with F-RTO 54
3.7(a) Time sequence 54
3.7(b) Congestion control state 54

4.1 Packet-framing in GPRS protocol stack 62
4.2 RTO Dynamics when a delay spike occurs – Eifel 71
4.2(a) During slow start 71
4.2(b) During congestion avoidance 71
4.3 RTO dynamics when a delay spike occurs – Eifel-I 73
4.3(a) During slow start 73
4.3(b) During congestion avoidance 73
4.4 In the presence of delays – Eifel 74
4.5 In the presence of delays – Eifel-I 74
4.6 TCP Reno with unnecessary multiple fast retransmits 77
4.6(a) Time sequence 77
4.6(b) Congestion control state 77
4.7 TCP NewReno with a lost retransmitted-packet 78
4.7(a) With NewReno's bugfix 78
4.7(b) With Eifel-I-based solution 78
4.8 The Eifel-I-based approach's recovery upon a retransmit loss early in the window 81
4.8(a) The initial version 81
4.8(b) The improved version 81
4.9 Comparison of different approaches for avoiding multiple fast retransmits 84
4.9(a) The "duplicate" scenario 84
4.9(b) The "loss" scenario 84
4.10 Comparison of different approaches for avoiding multiple fast retransmits, – delayed acknowledgment is enabled 85
4.10(a) The "duplicate" scenario 85
4.10(b) The "loss" scenario 85

5.1 A simplified user's view of NS-2 89
5.2 The correspondence between OTcl and C++ 90
5.3 The structure of a link in NS-2 91

6.1 Simulation topology 96
6.2 A spurious timeout 97
6.2(a) DSACK: time sequence 97
6.2(b) DSACK: congestion control state 97
6.2(c) Eifel: time sequence 97
6.2(d) Eifel: congestion control state 97
6.2(e) F-RTO: time sequence 97
6.2(f) F-RTO: congestion control state 97
6.2(g) Eifel-I: time sequence 97
6.2(h) Eifel-I: congestion control state 97
6.3 TCP Reno and Newreno during a handover with different buffer sizes . 100
6.3(a) Time for Reno with bugfix 100
6.3(b) Packets for Reno with bugfix 100
6.3(c) Time for Reno without bugfix 100
6.3(d) Packets for Reno without bugfix 100
6.3(e) Time for Newreno with bugfix 100
6.3(f) Packets for Newreno with bugfix 100
6.3(g) Time for Newreno without bugfix 100
6.3(h) Packets for Newreno without bugfix 100
6.4 TCP Sack during a handover with different buffer sizes 102
6.4(a) Time for Sack with bugfix 102
6.4(b) Packets for Sack with bugfix 102
6.4(c) Time for Sack without bugfix 102
6.4(d) Packets for Sack without bugfix 102
6.5 A spurious timeout on a congested link - with bugfix 103
6.5(a) A whole view 103

6.5(b) A clearer view of the lost packet 103
6.6 TCP Reno and Newreno during a handover with different buffer sizes – two connections 105
6.6(a) Time for Reno with bugfix 105
6.6(b) Packets for Reno with bugfix 105
6.6(c) Time for Reno without bugfix 105
6.6(d) Packets for Reno without bugfix 105
6.6(e) Time for Newreno with bugfix 105
6.6(f) Packets for Newreno with bugfix 105
6.6(g) Time for Newreno without bugfix 105
6.6(h) Packets for Newreno without bugfix 105
6.7 TCP Sack during a handover with different buffer sizes – two connections 106
6.7(a) Time for Sack with bugfix 106
6.7(b) Packets for Sack with bugfix 106
6.7(c) Time for Sack without bugfix 106
6.7(d) Packets for Sack without bugfix 106
6.8 TCP Reno and Newreno during a handover with different buffer sizes – four connections 108
6.8(a) Time for Reno with bugfix 108
6.8(b) Packets for Reno with bugfix 108
6.8(c) Time for Reno without bugfix 108
6.8(d) Packets for Reno without bugfix 108
6.8(e) Time for Newreno with bugfix 108
6.8(f) Packets for Newreno with bugfix 108
6.8(g) Time for Newreno without bugfix 108
6.8(h) Packets for Newreno without bugfix 108
6.9 TCP Sack during a handover with different buffer sizes – four connections 109
6.9(a) Time for Sack with bugfix 109
6.9(b) Packets for Sack with bugfix 109

6.9(c) Time for Sack without bugfix 109
6.9(d) Packets for Sack without bugfix 109
6.10 TCP Reno and Newreno in GPRS with LLR 111
6.10(a) Time for Reno with bugfix 111
6.10(b) Packets for Reno with bugfix 111
6.10(c) Time for Reno without bugfix 111
6.10(d) Packets for Reno without bugfix 111
6.10(e) Time for Newreno with bugfix 111
6.10(f) Packets for Newreno with bugfix 111
6.10(g) Time for Newreno without bugfix 111
6.10(h) Packets for Newreno without bugfix 111
6.11 TCP Sack in GPRS with LLR 112
6.11(a) Time for Sack with bugfix 112
6.11(b) Packets for Sack with bugfix 112
6.11(c) Time for Sack without bugfix 112
6.11(d) Packets for Sack without bugfix 112
6.12 TCP Reno and Newreno in UMTS with LLR 114
6.12(a) Time for Reno with bugfix 114
6.12(b) Packets for Reno with bugfix 114
6.12(c) Time for Reno without bugfix 114
6.12(d) Packets for Reno without bugfix 114
6.12(e) Time for Newreno with bugfix 114
6.12(f) Packets for Newreno with bugfix 114
6.12(g) Time for Newreno without bugfix 114
6.12(h) Packets for Newreno without bugfix 114
6.13 TCP Sack in UMTS with LLR 115
6.13(a) Time for Sack with bugfix 115
6.13(b) Packets for Sack with bugfix 115
6.13(c) Time for Sack without bugfix 115

6.13(d) Packets for Sack without bugfix 115
6.14 TCP Reno and Newreno in UMTS with LLR – two connections 116
6.14(a) Time for Reno with bugfix 116
6.14(b) Packets for Reno with bugfix 116
6.14(c) Time for Reno without bugfix 116
6.14(d) Packets for Reno without bugfix 116
6.14(e) Time for Newreno with bugfix 116
6.14(f) Packets for Newreno with bugfix 116
6.14(g) Time for Newreno without bugfix 116
6.14(h) Packets for Newreno without bugfix 116
6.15 TCP Sack in UMTS with LLR – two connections 117
6.15(a) Time for Sack with bugfix 117
6.15(b) Packets for Sack with bugfix 117
6.15(c) Time for Sack without bugfix 117
6.15(d) Packets for Sack without bugfix 117
6.16 TCP Reno and Newreno in UMTS with LLR – four connections 118
6.16(a) Time for Reno with bugfix 118
6.16(b) Packets for Reno with bugfix 118
6.16(c) Time for Reno without bugfix 118
6.16(d) Packets for Reno without bugfix 118
6.16(e) Time for Newreno with bugfix 118
6.16(f) Packets for Newreno with bugfix 118
6.16(g) Time for Newreno without bugfix 118
6.16(h) Packets for Newreno without bugfix 118
6.17 TCP Sack in UMTS with LLR – four connections 119
6.17(a) Time for Sack with bugfix 119
6.17(b) Packets for Sack with bugfix 119
6.17(c) Time for Sack without bugfix 119
6.17(d) Packets for Sack without bugfix 119

A-1 Multiframe structure with 52 TDMA frames A-6

A-2 Cell change – new cell in the same routing area A-12

A-3 Cell change – new cell in another RA handled by the same SGSN . . . A-13

A-4 Cell change – new cell in another RA handled by another SGSN A-14

List of Tables

2.1 Coding schemes of GPRS 14

2.2 GPRS logical channels (UL: uplink; DL: downlink) 15

4.1 RLC data block size for the four GPRS coding schemes. No MAC header is included here. 63

4.2 Comparison of Eifel-I and other approaches 67

A-1 Precedence levels A-7

A-2 GPRS delay classes A-7

A-3 GPRS reliability classes in terms of residual error rates A-8

A-4 GPRS reliability classes with the corresponding protocol mode combinations A-9

A-5 GPRS peak throughput classes A-9

A-6 GPRS mean throughput classes A-10

List of Abbreviations

ACK	Acknowledgment
BDP	Bandwidth-Delay Product
BS	Base Station
cwnd	Congestion Window
DSACK	Duplicate Selective Acknowledgment
DUPACK	Duplicate Acknowledgment
dupthresh	Duplicate Acknowledgment Threshold
KB	kilobytes
kb	kilobits
kbps	kilobits per second
MS	Mobile Station
MSL	Maximum Segment Lifetime
MTU	Maximum Transmission Unit
PAWS	Protect Against Wrapped Sequence Numbers
RFC	Request For Comments
RTO	Retransmission Timeout
RTT	Round-Trip Time
RTTM	Round-Trip Time Measurement
RTTVAR	Round-Trip Time Variation
SACK	Selective Acknowledgment
SRTT	Smoothed Round-Trip Time
ssthresh	Slow Start Threshold
TCP	Transmission Control Protocol

Summary

Transmission Control Protocol (TCP) is probably the most widely used and mature transport protocol today for Internet access. However, TCP was originally designed for wired networks, so some assumptions based on the properties of wired networks not hold for the currently widely-deployed wireless networks any more. In fact, many problems have arisen in recent years for TCP over wireless links. Some of the main problems include spurious timeouts, congestion losses, etc. In the thesis, we propose a new approach, Eifel-I, for enhancing TCP's robustness in the presence of these problems. Our main focus is on dealing with spurious timeouts. In conjunction with Eifel-I, we also suggest some enhancements to the TCP retransmission timer and to non-SACK TCPs' ability in handling multiple packet losses. Experiment results show that in situations like wireless networks where packet losses and variable delays frequently occur or co-occur, Eifel-I can deliver consistent performance improvement because it is capable of efficiently coping with both variable delays and packet losses. In all the scenarios we have experimented in, Eifel-I is always better than or at least the same as the other related approaches. In certain cases, it can achieve up to 40% improvement over the original TCP, and more than 20% improvement over approaches like DSACK, Eifel and F-RTO.

Keywords: Transmission Control Protocol, Wireless, TCP Timestamp, Retransmission, Timeout, Packet Loss

Chapter 1

Introduction

Transmission Control Protocol (TCP) [41] has been in use for more than two decades since its standardization, and it still remains the most widely used transport protocol today for Internet applications such as the World Wide Web (WWW), file transfer, email, etc. Its congestion control algorithms [4] are essential for the stability of the Internet, and they have a strong effect on TCP performance. During the past years, a great deal of work has been devoted to the research and development of TCP, to enable it to cope with new challenging circumstances that were not anticipated when it was initially designed. One of the main challenges in recent years is the increasing deployment of wireless networks or wireless Internet access.

1.1 Motivation of Research

TCP algorithms were mostly developed empirically and were based on assumptions that hold in wired networks but not necessarily in wireless ones. As we all know, TCP has been tuned well for traditional networks made up of wired links and stationary hosts. However, it does not work well with the current cellular wireless networks such as GPRS, UMTS, etc., which are becoming more and more popular. In fact, many problems have arisen in recent years for TCP over wireless links.

1.1.1 Spurious Timeouts

One of the main causes for TCP's bad performance over wireless links is the large delay variations that frequently occur over wireless links, which can trigger problems like spurious timeouts [32]. A delay spike is a sudden increase in the latency of the communication path. 2.5G/3G wireless links are likely to experience delay spikes exceeding the typical RTT by several times due to reasons like link layer retransmission, handover, resource allocation, bandwidth oscillation, etc. Delay variation occurs quite often because of these reasons, so it has led the spurious timeout problem to be a more serious concern which needs to be handled properly. Another related problem is spurious fast retransmits. It is mainly caused by packet reorderings due to link layer retransmission. However, as packet reordering is currently disabled in 2.5G/3G wireless systems, spurious fast retransmit is not a main concern at the moment.

Spurious timeouts can cause suboptimal TCP performance by falsely triggering the go-back-N timeout retransmission and unnecessarily reducing the TCP transmission speed, so some enhancement is needed for TCP to alleviate the sacrificed performance. Generally speaking, the possible solutions for this can roughly be divided into two categories. One alternative is to avoid the spurious retransmission in the first place. This can be achieved by changing the algorithm used for the RTO calculation. Different constants and granularities applied to the standard TCP retransmission timer [39] have been studied [3]. A totally new set of algorithms for adapting the retransmission timer has also been suggested, as in [34]. However in our opinion, such kind of algorithms may not work well for the various network environments.

Another way to mitigate the performance penalty is to avoid the problems caused by spurious timeouts by changing the TCP sender's behavior thereafter. A number of algorithms have been proposed in this category during the past few years:

The Eifel algorithm [32] suggests that the TCP sender includes extra information in every packet sent and the receiver echoes it back in the corresponding ACK. With the information in the ACK, the sender can eliminate the retransmission ambiguity and detect spurious retransmissions. It can be used for solving both the spurious timeout

and the spurious fast retransmit problem. A key feature of this algorithm is that it is able to detect, upon the first acceptable ACK that arrives during loss recovery, whether a retransmission is spurious. It is crucial to be able to avoid the go-back-N retransmission. Currently, the Eifel algorithm uses the TCP Timestamp option [29] as the piece of extra information for distinguishing original transmits from retransmits, in order to disambiguate unnecessary retransmissions from real loss events.

Unlike Eifel, the Duplicate SACK (DSACK) [18] based enhancement [7] [8] [51] relies on the TCP receiver to indicate whether it receives a packet that has arrived earlier. The receiver can pass this information to the sender through the first SACK block, i.e., the DSACK block in the TCP header. This alternative has its benefits over the Eifel algorithm because the SACK option [35] is more widely deployed than the Timestamp option, and the SACK blocks are appended to TCP headers only when necessary. However, when a spurious retransmission occurs, the first ACK carrying the DSACK block only arrives at the sender after loss recovery has already terminated. Thus, this DSACK-based approach cannot avoid the unnecessary go-back-N retransmission.

Forward RTO Recovery (F-RTO) [45] is a new algorithm for a TCP sender to only recover after a spurious timeout. Unlike the two algorithms presented above, it does not require the use of any TCP options or additional bits in the TCP header. It uses a set of heuristic rules for detecting spurious timeouts.

These algorithms are different from each other in how they detect a spurious retransmission, but they may share the same response algorithm for undoing the changes of congestion control parameters made after a spurious retransmission. In addition, some also try to avoid future spurious retransmissions by adapting either the retransmission timer for a spurious timeout or *dupthresh* for a spurious fast retransmit. These adaptation algorithms in fact pick the idea of avoiding spurious retransmissions in the first place. But the adaptation can only be done after some (at least one) spurious retransmissions have occurred. The DSACK-based algorithm and the Eifel algorithm may use the same approach to adapt *dupthresh*. Because the Eifel algorithm currently uses the TCP Timestamp option in its implementation, it has the advantage of sampling every ac-

knowledged packet for RTT measurement, including retransmitted packets. F-RTO TCP and DSACK TCP cannot use retransmits in the RTT sampling and neither can common TCP implementations as they are prohibited from doing by Karn's algorithm [30]. Collecting more RTT samples may enable the TCP sender to come out with a better RTO estimation for adapting network changes and avoiding future spurious retransmissions. We will discuss the topic of retransmission timer adaptation in more detail in Chapter 4.

1.1.2 Packet Loss

Another main impairment to TCP is packet losses over wireless links. Due to the intrinsic properties of radio interface, wireless links were originally characterised as a transmission media with high non-congestion loss. As TCP congestion control algorithms (refer to Section 3.1.2) infer packet losses as indicating network congestion, such non-congestion losses can incorrectly trigger TCP congestion control and lead to transmission rate reduction in TCP-based applications. However, current 2.5G/3G wireless systems are heavily protected by link layer retransmission, so packet losses due to error or corruption are now very low over these wireless links. Other than corruption-based losses, packet losses in current wireless networks are mostly due to congestion at the bottleneck wireless nodes during handovers or mobility management. A number of algorithms have also been proposed in this category, such as Indirect-TCP [9], the Snoop protocol [6], Cumulative Explicit Transport Error Notification (CETEN) [16], etc. However, these algorithms are mainly aimed at corruption-based losses. As link layer enhancements for reducing wireless link losses including ARQ and FEC are already part of the 2.5G/3G wireless systems, these schemes for TCP only provide overlapped functions that can introduce little performance improvements. For example, Snoop TCP is reported [47] to not work well over GPRS because high delay over the GPRS radio interface can trigger duplicate retransmissions in the Snoop agent.

1.2 Objectives of Research

The spurious timeout problem has become a big concern for TCP in recent years, mainly because the wide deployment of wireless links introduces frequently-occurring delay spikes. Besides a reduction in the TCP sender's transmission rate, a spurious timeout also results in the unnecessary retransmission of the last window of packets. As pointed out in [22], the amount of data sent over wireless links should be minimized because battery power consumption and radio resource usage are often as important as download time for mobile users and operators. So in a wireless environment, it may be more desirable to avoid unnecessary go-back-N retransmission rather than undo unnecessary sending rate reduction. In this sense, spurious timeouts are much more troublesome than spurious fast retransmits. Avoiding unnecessary transmission rate reduction becomes more important as the capacity (bandwidth-delay product - BDP) of wireless links increases, such as in the UMTS network.

As packet loss over wireless links are mainly due to congestion, it adheres to the basic assumption of TCP. So current TCP implementation should be able to cope with it to some extent. Although some explicit mechanism may still be needed for mitigating the impairment due to multiple congestion losses, congestion is less serious than spurious timeouts. So our main focus in this thesis will be on the spurious timeout problem. In the meantime, we will also try to solve the problems caused by congestion losses over wireless links.

Before devising our own solution, it is good practice to analyze the pros and cons of the existing algorithms:

- Although the DSACK-based algorithm is based on the well deployed TCP SACK option [35], it has a major drawback: it can only come into use after the unnecessary go-back-N retransmission has been done. As avoiding unnecessary transmission is even more important than recovering sending rate reduction in wireless environments, it is essential for the new approach to be able to avoid the go-back-N retransmission.

- F-RTO works only in detecting spurious timeouts. It is efficient because it can avoid the go-back-N retransmission. But it is only a heuristic approach that can be confused by network pathologies like reordering or duplication of key packets, and so it may not always be effective. Although the behavior of packet reordering is currently prohibited, it may be re-enabled later to allow for better performance of real-time applications. So it is still crucial for the new approach to have the ability to handle spurious fast retransmits.

- Compared with the above two algorithms, the basic Eifel algorithm is both effective and efficient in detecting spurious retransmissions (including spurious timeouts and spurious fast retransmits). Results in [22] show its ability in improving TCP performance over a GPRS link, but its current implementation introduces a 12-byte timestamp for each packet. The timestamp overhead is a considerably high cost for low-speed wireless links. It also prevents the use of other TCP options and the use of current TCP/IP header compression schemes [28] [14], which are very useful for slow links. However, the timestamping ability can lead to a more up-to-the-minute RTT timing, which may benefit RTO estimation.

In conclusion, although the existing algorithms can effectively detect spurious retransmits and do some recovery, each of them also suffers from some weaknesses. In order to achieve optimal TCP performance, especially over wireless links, we want to devise a new and more superior approach to fixing the spurious retransmission problem. Ideally, this approach would keep the advantages of the existing approaches while eliminating their problems. In developing our approach, we keep the basic idea of the Eifel algorithm but work out a better way to realise it. The following is some considerations for our proposal:

- First of all, Eifel currently suffers a lot from the use of timestamps as extra information. Hence, we need to find a piece of extra information that would introduce as little overhead as possible.

- Second, the new approach should retain the strengths of the current approach, such

as its early detection and its robustness against ACK losses.

- Third, the new approach should enable the use of current TCP/IP header compression schemes that have been proved to be useful over low-speed links.

- Fourth, [3] pointed out that the current standard TCP retransmission timer defined in RFC2988 [39] adapts fairly slow to changes in network conditions. This is because retransmits are not allowed by Karn's algorithm [30] to be used in RTT sampling. With the use of timestamps, the current Eifel approach solves this slow adaptation problem, and provides the possibility for a better RTO estimator to avoid future spurious timeouts. Our new approach should also try to retain this property.

- Fifith, if possible, our approach should cover the packet loss problem as well.

1.3 Contributions of Thesis

With the above considerations in mind, we propose Eifel-I, which introduces the selective use of timestamps in implementing the Eifel algorithm. The new approach uses timestamps only for retransmits and their corresponding ACKs. This "use-on-demand" idea comes from the usage pattern of SACK blocks in the TCP SACK option [35]. Since the retransmits only form a relatively small part of total transmitted packets, the 12-byte timestamp overhead can generally be avoided most of the time and the compression schemes can now be used. Moreover, by retaining the timestamps in retransmits and their ACKs, we also keep the TCP sender's ability to sample retransmits for RTT measure. The following is a list of our contributions:

- We propose a new approach for solving spurious retransmissions by improving on the existing Eifel algorithm [32]. Our approach retains the advantages of the existing approach while avoids its overheads and problems which are caused by the persistent use of timestamps.

- In conjunction with the new detection approach, we also develope a simple yet effective enhancement to the current TCP retransmission timer. According to our simulation results, after incurring the first spurious timeout, the enhanced retransmission timer can avoid most subsequent spurious timeouts. In fact, those future spurious timeouts never happen at all. The advantage of our enhancement comes from its fast and stable adaptation to changing delays in the network.

- With Eifel-I, we also work out a new method to greatly improve the ability of non-Sack TCPs (e.g., TCP Reno, NewReno, etc.) to recover from multiple packet losses. It enables the TCP sender to avoid unnecessary fast retransmits if the DUPACKs are triggered by duplicate packets, and to efficiently recover lost packets through fast retransmit and fast recovery instead of waiting for a timeout.

- We evaluate Eifel-I with the original TCP and other approaches such as DSACK, Eifel, F-RTO by using simulation. We provide extensive experiment results and detailed discussions of Eifel-I's improvements in various circumstances. From the results, we find that in situations like wireless networks where packet losses and variable delays frequently occur or co-occur, Eifel-I can deliver consistent performance improvement because it is capable of efficiently coping with both variable delays and packet losses. In all the scenarios we have experimented (regardless of the TCP flavor used, or the number of concurrent connections, etc.), Eifel-I is consistently better than or on par with the other approaches. In certain cases, it can achieve up to 40% improvement over the original TCP, and more than 20% improvement over the approaches like DSACK, Eifel and F-RTO.

1.4 Organization of Thesis

The rest of the thesis is organized as follows. In Chapter 2, we provide an overview of wireless networks, including GPRS and 3G UMTS. In Chapter 3, we first briefly look at the basic concepts and algorithms of TCP, list the distinct wireless link characteristics and discuss their possible impairments on TCP performance, and then look in more

detail at the existing algorithms aimed at avoiding those impairments and improving TCP performance. Our main focus is on spurious timeouts and congestion losses. We discuss in detail our proposal for solving spurious timeouts and the other related enhancements for TCP in Chapter 4. We then briefly introduce the NS-2 network simulator [37] which we use as base simulator in our experiments, and describe our work on implementing Eifel-I and other approaches in the simulator in Chapter 5. We present and analyze our simulation results in Chapter 6. We conclude and outline our future work in Chapter 7.

Chapter 2

Overview of Cellular Mobile Radio Systems

During the first half of the last century, the transmission of human voice through the telephone was the dominant means of communication next to telegraphy. However, radio-supported mobile communication has since grown in importance in the last few decades, and particularly in the last few years due to the rapidly-increasing demand for wireless Internet access and technical advances in transmission and switching technology. In contrast to wireline networks, mobile radio networks allow geographically unrestricted communication to take place anywhere, especially where it is not economical or possible to install cabling.

There are a number of wireless links deployed for different purposes [49]:

- *Public cellular mobile radio systems* extend the telephone service of wireline networks to mobile users.
- *Wireless Local Area Networks* (WLANs) take into account the growing demand to avoid the cabling of workstation computers. Compared with cellular systems, WLANs work within a limited distance.
- *Satellite radio systems* provide global communication and accessibility, which are mostly made possible with a fixed GEO satellite.

Among them, *public cellular mobile radio systems* have become most commonly used. They are usually known as 2G/3G wireless networks. In this thesis, we will focus on the widely-deployed cellular wireless links.

Second-generation (2G) wireless networks, especially the Global System for Mobile Communication (GSM), were once a step up in technology evolution and have gained spectacular growth in the last few decades. Currently, the extension of GSM – General Packet Radio Service (GPRS) – is widely deployed in the market. However, 2G wireless networks have primarily been designed for voice communication, and data services are essentially an add-on to these networks. Driven by the increasingly pervasive Internet access and the widespread use of mobile technologies, the next generation (3G) wireless networks have requirements for both radio access networks and core networks, including higher data rates, enhanced support for packet data, etc. The Universal Mobile Telecommuncation System (UMTS) [26] [38] is the main cellular wireless architecture developed to meet the 3G requirements.

As background to the discussion about TCP over Wireless later in the thesis, here we provide a detailed overview on the currently-domaint GPRS network, focusing on the parts essential to this thesis. We will also briefly look at future UMTS technology.

2.1 GPRS: General Packet Radio Service

In the evolution of GSM towards 3G systems, the integration of GPRS has been an important milestone. It is a new bearer service for GSM that greatly improves and simplifies wireless access to packet data networks, e.g., to the Internet. It applies a packet radio principle to transfer user data packets in a more efficient way between mobile stations (MSs) and external packet data networks (PDNs).

GPRS has been standardized by the European Telecommunications Standards Institute (ETSI) as part of the GSM Phase 2+ development. It represents the first implementation of packet switching within GSM, which is essentially a circuit-switched technology.

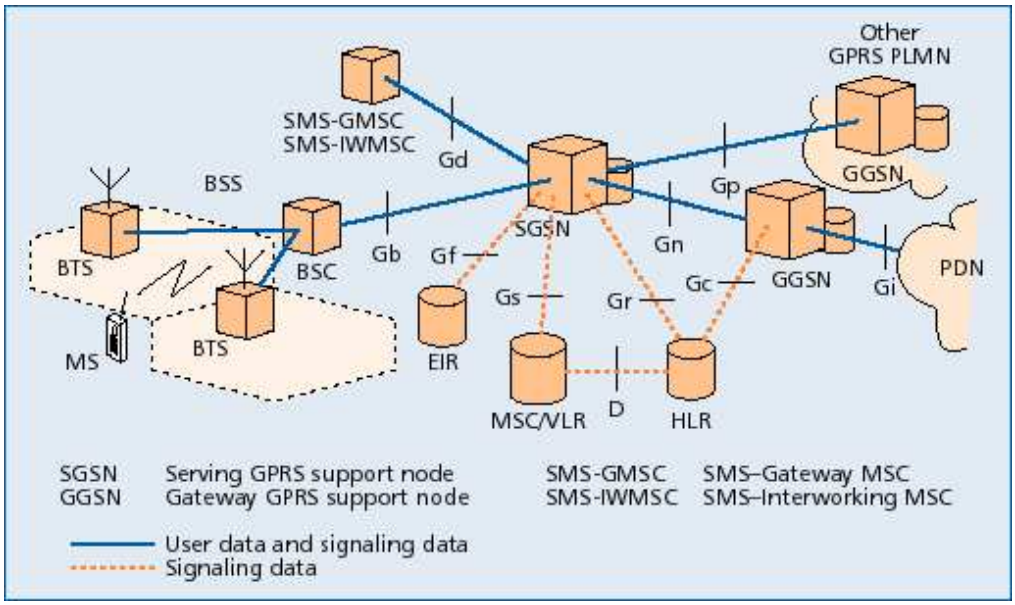


Figure 2.1: GPRS system architecture

2.1.1 System Architecture

The GPRS system architecture is illustrated in Fig. 2.1 [11]. A mobile station is denoted as MS. A cell is formed by the radio area coverage of a base transceiver station (BTS). Several BTSs together are controlled by one base station controller (BSC). The BTS and BSC together form the base station subsystem (BSS). Compared with a GSM network, there's no change in the BSS of a GPRS network.

A traditional GSM network contains the mobile switching center (MSC) for traffic routing, as well as databases like the home/visited location register (HLR/VLR), the authentication center (AUC), and the equipment identity register (EIR) for call control and network management. However, it does not provide sufficient functionality to realize a packet data service. To enable packet switching over the existing GSM network, two new elements are introduced into the GPRS core network:

- **Gateway GPRS Support Node (GGSN)** serves as the interface towards external PDN or other *Public Land Mobile Network* (PLMN). Here, packet switching functions are fulfilled, e.g., the evaluation of *Packet Data Network* (PDP) addresses and routing to MSs via the SGSN.

- **Serving GPRS Support Node (SGSN)** represents the GPRS switching center by

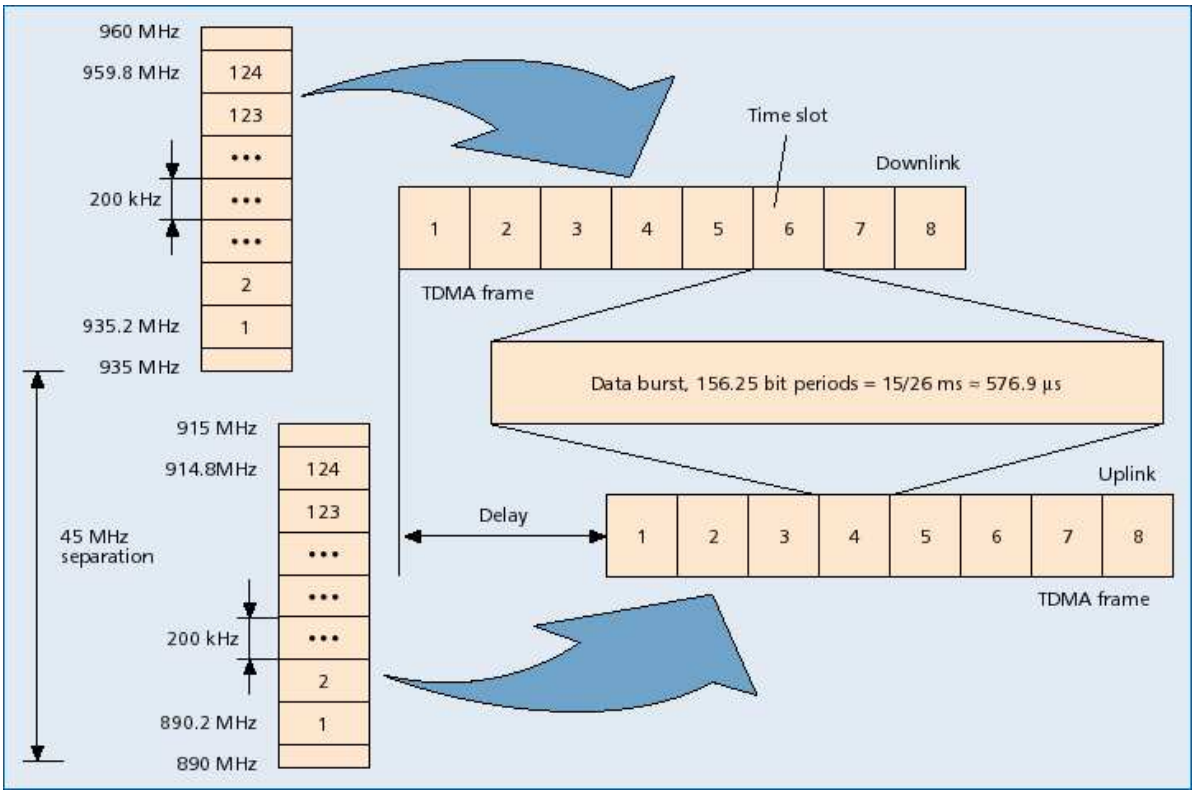


Figure 2.2: GPRS radio interface

analogy to the MSC. It is responsible for routing inside the radio network and for mobility and resource management. It also provides authentication and encryption for GPRS subscribers.

In general, there is a many-to-many relationship between SGSNs and GGSNs. All GSNs are connected via an IP-based GPRS backbone network. Within this backbone, they encapsulate PDN packets and transmit (or tunnel) them using the GPRS Tunneling Protocol (GTP).

2.1.2 Radio Interface

In managing radio resources, GPRS adopts the same combination of *Frequency Division Multiple Access* (FDMA) and *Time Division Multiple Access* (TDMA) as GSM. (Details on FDMA and TDMA may be found in Appendix A.1.1).

Coding scheme	Code rate	Uncoded payload (bit)	Coded bits (bit)	Punc-tured bits	Data rate (kbits/s) 1 TS	2 TS	4 TS	8 TS
CS-1	1/2	181	456	0	9.05	18.1	36.2	72.4
CS-2	2/3	268	588	132	13.4	26.8	53.6	107.2
CS-3	3/4	312	676	220	15.6	31.2	62.4	124.8
CS-4	1	428	456	0	21.4	42.8	84.6	171.2

Table 2.1: Coding schemes of GPRS

Channel Concept

As shown in Fig. 2.2, two frequency ranges 45 MHz apart are reserved for the uplink (890 – 915 MHz) transfer from the MS and the downlink (935 – 960 MHz) transfer from the BTS. Each 25 MHz width is divided into 124 single carrier channels of 200 kHz width. A certain number of these frequency channels is allocated to a BTS, i.e., to a cell.

Each frequency channel carries eight TDMA channels, or eight time slots (TS). The eight TSs form a TDMA frame. Each frame has a duration of 4.615 ms. Thus one slot takes 0.577 ms, and is able to carry a data burst of 148 bits, which is 8.25 bit shorter in duration to realize a guard time between data bursts in order to avoid overlapping. The 148-bit burst contains two data pieces of 57-bit each and some tail and control bits. The recurrence of one particular time slot defines a physical channel.

In traditional GSM, a channel is allocated to a particular user for the entire call period (even if no data is transmitted). With packet-switching introduced in GPRS, channels are only allocated when data packets are sent or received, and released immediately after the transmission. So multiple users can share one physical channel. For bursty traffic like Internet access, this leads to much more efficient resource usage [49]. Moreover, the channel allocation in GPRS also allows multislot operation on a single TDMA frame by a MS. This results in a very flexible channel allocation.

To offer higher data rates (per timeslot), GPRS introduces three new coding schemes (CS-2 to CS-4). All four schemes are listed in Table 2.1 [47]. Coding in GPRS is always done for a single RLC/MAC [1] radio block which always has a coded length of 456 bits. Since each data burst in one TS can transfer (57*2=) 114 data bits, the radio block

[1] RLC/MAC stands for Radio Link Control/Medium Access Control. They are introduced in Section 2.1.3.

Group	Channel	Name	Direction	Function
PCCCH	PRACH	Packet Random Access Channel	UL	random access
	PPCH	Packet Paging Channel	DL	paging
	PAGCH	Packet Access Grant Channel	DL	access grant
	PNCH	Packet Notification Channel	DL	multicast
PBCCH	PBCCH	Packet Broadcast Control Channel	DL	broadcast
PDTCH	PDTCH	Packet Data Traffic Channel	UL/DL	data
PDCCH	PACCH	Packet Associated Control Channel	UL/DL	associated control
	PTCCH	Packet Timing Advance Control Channel	UL/DL	timing advance

Table 2.2: GPRS logical channels (UL: uplink; DL: downlink)

is always interleaved over four normal bursts. The pre-coded payload length depends on the coding scheme and varies from 181 to 428 bits. These coding schemes trade off transmission errors for data throughput. CS-1 to CS3 use the same convolutional code but different puncturing level, which leads to different code rates and thus different protection quality. CS-4 has no coding at all. Thus CS-1 has the lowest user data rate but the best error protection.

Table 2.1 also shows the achievable user data rates for a number of multislot combinations. Note that uplink and downlink are allocated separately (unlike GSM's symmetric allocation), which efficiently supports asymmetric data traffic (e.g., FTP downloading, Internet surfing, etc.) [47].

Logical Channels

GPRS defines a number of logical packet data channels (PDCH) that can be mapped onto GPRS physical PDCHs. Table 2.2 lists the GPRS logical PDCHs and their functions [11]. A detailed description of the channels may be found in Appendix A.2.

Logical to Physical Channel Mapping

The GPRS **radio interface protocol** refers to the physical and the RLC/MAC layer of the GPRS protocol architecture (see next section). The mapping principles include a master-slave concept and the capacity-on-demand principle [47]:

- The *master-slave* concept is related to logical channel assignment onto physical channels. The control channels are mapped only to a single physical PDCH lo-

cated on a single timeslot acting as master. The other PDCHs (on the other timeslots) for user data act as slaves.

- The allocation of PDCHs is based on the *capacity-on-demand* principle. Since radio resources are shared by both packet-switched (GPRS) data and circuit-switched (GSM) data and speech, there have to be some strategies on how to distribute the resources. The distribution (or allocation) depends on the current traffic load, the priority of the service, and the multislot class. It means that GPRS does not require fixed allocated PDCHs. Capacity assignment for packet data transmission can be done according to actual demand. For example, physical channels not currently in use by conventional GSM can be allocated as PDCHs to increase the quality of service (QoS) for GPRS; when there is resource demand for services with higher priority (like voice traffic), PDCHs can be de-allocated [11].

In GPRS, multiplexing on the radio interface is based on RLC/MAC packets. The various logical packet data channels are multiplexed onto physical channels using a cyclically recurring multiframe structure. More information on the multiframe structure may be found in Appendix A.2.2.

2.1.3 Protocol Stack

The GPRS protocol architecture follows the ISO/OSI model. It provides transmission of user data (see Fig. 2.3 [11]) and its associated signaling, e.g., for flow control, error detection, and error correction. Here, we only focus on protocol layers for packet data transmission.

GPRS Backbone: SGSN – GGSN

As mentioned earlier, packets are encapsulated within the GPRS backbone network. The GPRS Tunneling Protocol (GTP) [49] tunnels the encapsulated packets between the GPRS support nodes (GSNs). GTP is defined both between GSNs within one PLMN (Gn interface) and between GSNs of different PLMNs (Gp interface).

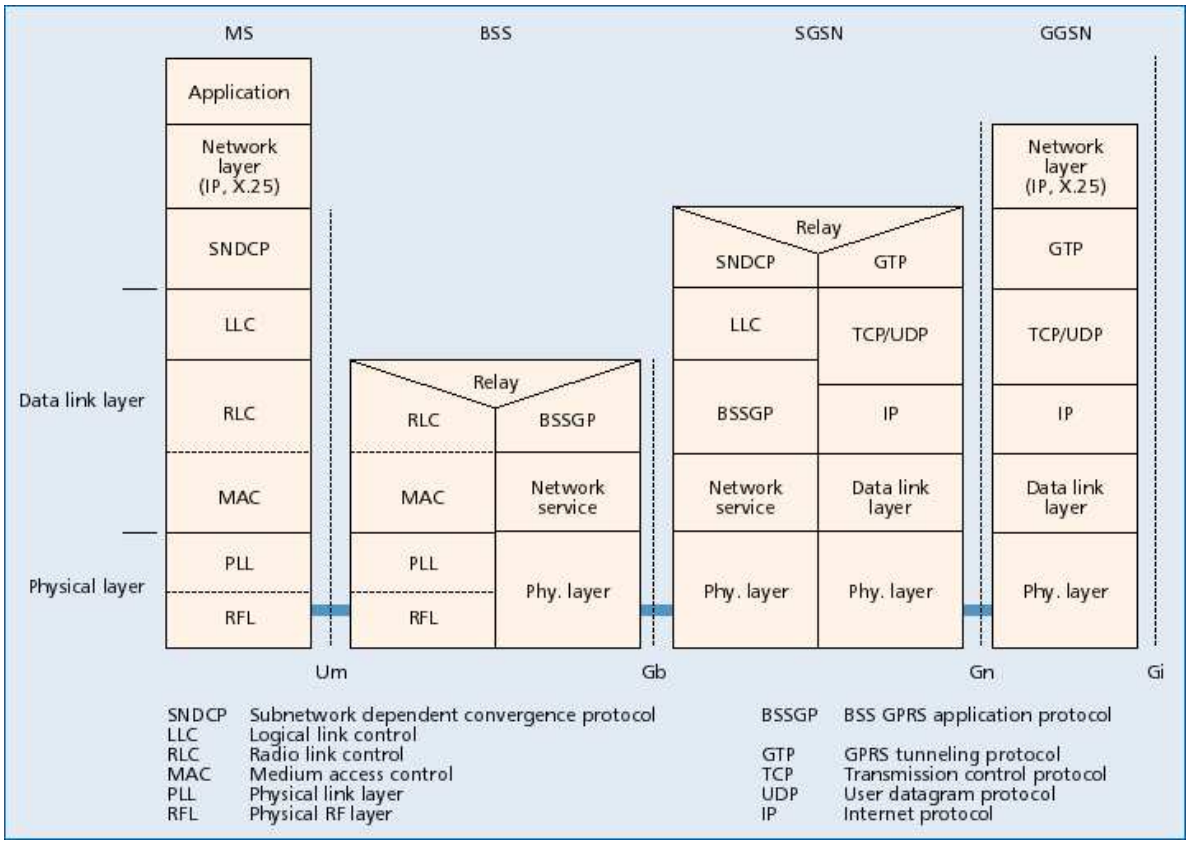


Figure 2.3: GPRS protocol stack

GTP packets carry the user's IP or X.25 packets. Below GTP, standard protocols like TCP or UDP are employed to transport GTP packets within the backbone network. IP is employed in the network layer to route packets through the backbone. Ethernet, ISDN, or ATM-based protocols may be used below IP.

Between SGSN and BSS

The Subnetwork Dependent Convergence Protocol (SNDCP) [47] is used to transfer data packets between SGSN and MS. Its main functions include: multiplexing of several network connections onto one virtual connection of the underlying LLC layer; compression/decompression of user data and redundant protocol control information (e.g., TCP/IP header compression [28]); segmentation and reassembly of network layer packets, if any packet is longer than the maximum payload size for an LLC frame.

Logical Link Control (LLC) [49] is part of the data link layer, responsible for the transportation of data packets between MS and SGSN. It provides a highly reliable log-

ical link independent of the underlying radio interface protocol. The essential functions are flow control and error protection (with ARQ and FEC mechanisms). More information on error protection in radio channels may be found in Appendix A.1.2. Other functions include: provision of one or more logical connections per user; sequence control and in-order delivery; ciphering and deciphering of the information field. There are two transfer modes [47]:

- In *unacknowledged mode*, neither error recovery nor reordering of packets is included. However, a checksum can detect errors, and packets in error are discarded.
- In *acknowledged mode*, packets are transmitted with sequence numbers, and error recovery and retransmission of packets in error are provided. The maximum number of outstanding packets is between 2 and 16, depending on the required quality of service (see Section 2.1.4). The maximum number of retransmission is set to 3, and after this number of tries, the recovery is left to the upper layer.

Each LLC packet [47] consists of: a 1-byte *address field*; a variable-length *control field* with a maximum of 36 bytes; an *information field* (or payload) ranging from 140 to 1520 bytes (defaults to be 1503 bytes); at end, a *frame check sequence* (FCS) of 3 bytes.

The BSS – SGSN Interface

Base Station Subsystem GPRS Protocol (BSSGP) delivers routing and QoS-related information between BSS and SGSN. The underlying Network Service (NS) protocol is based on the ATM Frame Relay protocol.

Data Link Layer over the Air Interface

Data Link Layer at the mobile Um interface can be divided in two sublayers: the *Radio Link Control* (RLC) layer and the *Medium Access Control* (MAC) layer [49] (see Fig. 2.3).

The main purpose of RLC is to establish a reliable logical link between MS and BSS. This includes segmentation and reassembly of LLC frames into RLC data blocks

(depending on the available coding schemes as listed in 2.1), and retransmission of uncorrectable errors by Automatic Repeat reQuest (ARQ). There are two possible modes: the acknowledged mode provides reliable data transmissions by using a selective ARQ protocol; the unacknowledged mode does not issue retransmission for error packets and is mainly for real-time services.

The MAC layer is responsible for providing efficient multiplexing of data and control signalling on uplink/downlink over the shared radio channels. It employs algorithms for contention resolution, multiuser multiplexing, and scheduling and prioritizing based on the negotiated QoS.

A RLC/MAC block consists of a 1-byte MAC header, a variable-length RLC header (2-3bytes), an information field and some spare bits. As mentioned before, the RLC/MAC radio block is the basic unit for GPRS channel coding. So the size of each block is constant for a specific coding scheme (see Table 2.1).

2.1.4 Quality of Service

Quality of Service (QoS) requirements of typical mobile packet data applications (e.g., web browsing, email transfer, etc.) are very diverse. Support of different QoS classes, which can be specified for each individual session, is therefore an important feature. GPRS allows defining QoS profiles using parameters like *service precedence*, *reliability*, *delay* and *throughput* [47] [49].

- Service precedence is the priority of a service in relation to another service. There exist three levels of priority: high, normal and low.

- Reliability indicates the transmission characteristics required by an application. Three reliability classes are defined, which guarantee certain maximum values for the probability of loss, duplication, mis-sequencing and corruption (an undetected error) of packets.

- Delay parameters define maximum values for the mean delay and the 95-percentile delay. The latter is the maximum delay guaranteed in 95 percent of all transfers.

- The throughput specifies the maximum/peak bit rate and the mean bit rate.

(Detailed information on the QoS classes may be found in Appendix A.2.2).

Using these QoS classes, QoS profiles can be negotiated between the mobile user and the network for each session, depending on the QoS demand and the current available resources.

2.1.5 Mobility Management

In this section, we will briefly discuss the complexity involved in handling the mobility of a wireless terminal. A more detailed description of GPRS mobility handling may be found in [15].

Before an MS starts to use any packet data service over a GPRS network, it must register with an SGSN of the network. The network checks if the user is authorized, copies the user profile from the HLR to the SGSN, and assigns a packet temporary mobile subscriber identity (P-TMSI) to the user. This procedure is called *GPRS attach*. The disconnection from the GPRS network is called *GPRS detach*. It can be initiated by the mobile station or by the network (SGSN or HLR).

After a successful GPRS attach procedure, the MS is permitted to use the mobile GPRS service. However, it needs to establish a session with the GPRS network before it can exchange packets with an external packet data network (PDN). In particular, a virtual connection has to be set up with that PDN. This virtual connection is called a *GPRS PDP Context*. The PDP Context activation and management is supported by GPRS Session Management [49].

GPRS's Mobility Management (MM) comes into function when the MS is on the move and roams between cells covered by different BTS. In order for the GPRS network to keep track of the current location of the moving MS, a number of mobility management procedures need to be executed, and databases for mobility management are updated accordingly in SGSN, GGSN, and/or HLR to track the MS's location. Location management consists of two levels: GGSN and HLR track an MS up to its serving

SGSN while SGSN keeps track of an MS at the routing area [2] or cell level depending on its mobility management state.

GPRS has its own set of mobility management signalling for different types of RA updates. During an RA update operation, downlink and uplink data transmission is momentarily interrupted. The latency of packet delivery is increased due to these interruptions. In addition, packets could be misrouted prior to RA update completion, and this results in packet loss.

(More information on GPRS MM and a detailed illustration of the three RA update scenarios may be found in Appendix A.2.4.)

2.2 UMTS: Universal Mobile Telecommunication System

The *Universal Mobile Telecommunication System* (UMTS) [26] is the 3G mobile radio system promoted by ETSI. It is currently under standardization by the 3rd Generation Partnership Project, (3GPP). 3GPP released its first version of the specifications for UMTS in 1999, referred to as Release 99, which mainly introduced the new WCDMA-based radio access network. Further releases include Release 4 in 2001, which specifies minor corrections and enhancements for efficient IP support enabling provision of services through an all-IP core network; in 2002, Release 5 introduced a new subsystem called the IP multimedia subsystem (IMS) and also enhanced WCDMA radio technology with high-speed downlink packet access, which can achieve up to 10Mbps on the downlink; Release 6 and following releases can even support high data rates up to 20Mbps.

The essential UMTS technologies are covered in releases up to Release 5, which will also be the prevalent versions deployed in the next few years [26], so our subsequent elaborations will mostly be based on them.

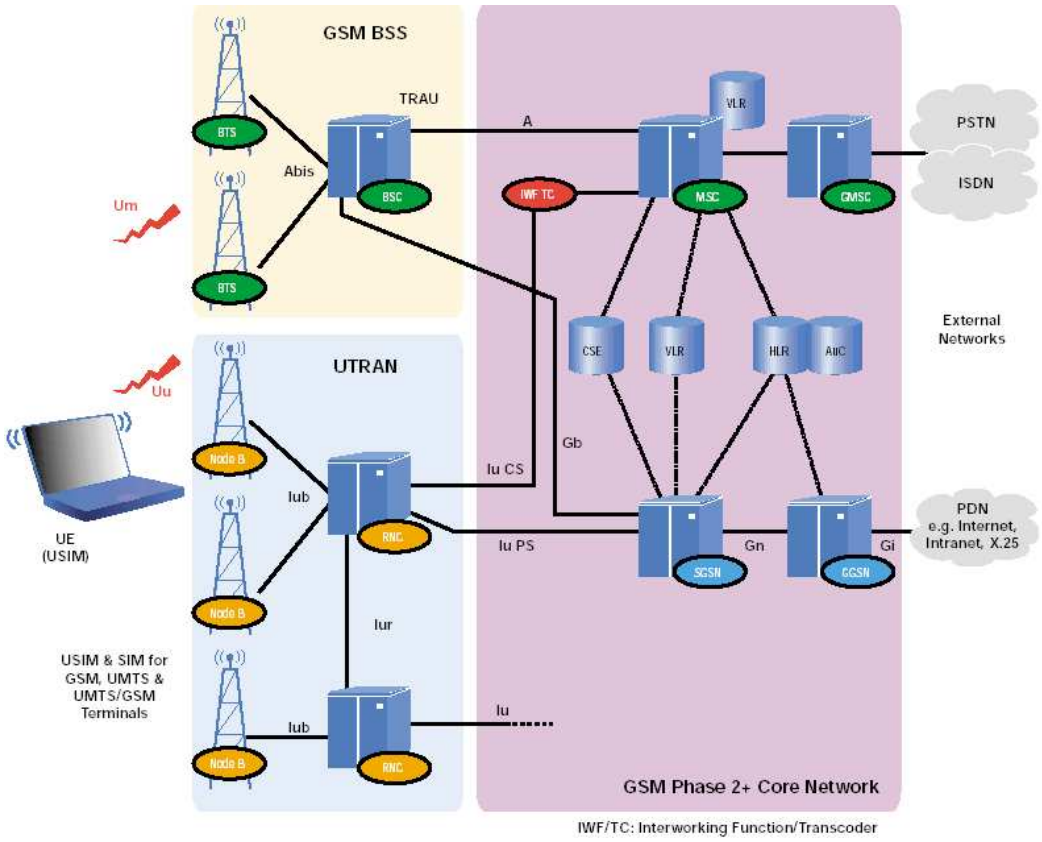


Figure 2.4: UMTS network architecture

2.2.1 System Architecture

Fig. 2.4 illustrates the overall architecture of a UMTS network. Conceptually, the UMTS network has four parts: *user equipment* (UE), *radio access network* (RAN), *core network* (CN) and *external networks*.

With the vision of building a mobile system that is accessible from anywhere at any time via different devices, UMTS's network architecture allows for a clear separation of RAN from CN [38]. This enables different types of RANs to be used independently of CN. In CN, packet core improvements have been made to enable the user of a common CN with any access technology. Compared with its predecessors like GPRS, UMTS's improvements have been made mostly in CN and RAN.

[2]One or more cells form a *routing area* (RA), and an RA is always served by just one SGSN.

Core Network

The UMTS core network presented in Fig. 2.4 is essentially the same as the existing GSM Phase 2+ (i.e., GPRS) core network. However, as we will see later, it moves all radio-related functionality into the RAN for access independence. The GPRS architecture has been introduced in Section 2.1, so here we only focus on the changes of UMTS from GPRS.

CN incorporates both the circuit-switched (CS) GSM and the packet-switched (PS) GPRS core network, and maintains a clear separation between the two. When communicating with the old BSS radio access, the existing A or Gb interfaces are used; however, the new UTRAN is connected to CN via the new Iu interface, which has *Iu-CS* and *Iu-PS* for either CS or PS traffic. Header compression is required to improve bandwidth usage over the air interface. In GPRS the compression resides in the SGSN, whereas UMTS moves it into the radio network controller (RNC). Thus, the UMTS SGSN knows nothing about the compression or other access-specific low-bandwidth optimization. This may slightly increase the transmission capacity between SGSN and RNC due to the full headers used.

UTRAN: UMTS Terrestrial Radio Access Network

UMTS differs from GPRS mostly in the new principles used for air interface transmission. A completely new radio interface is specified based on W-CDMA, which should provide data rates up to 2Mbps.

W-CDMA stands for *Wideband Code Division Multiple Access* [3]. For a CDMA system, the sender and receiver should agree on using certain codes for signal transformation and communication. In a W-CDMA system, the sender and receiver use a digital code that can spread narrowband input signals into wideband (as the name indicates) through transformation [38]. Two duplex modes exist [47]:

- **Frequency Division Duplex** (FDD) is the mainstream mode for UTRAN. In this mode, the base station and the mobile station transmit on different radio frequen-

[3] A brief introduction of different multiple access methods is provided in Appendix A.1.1.

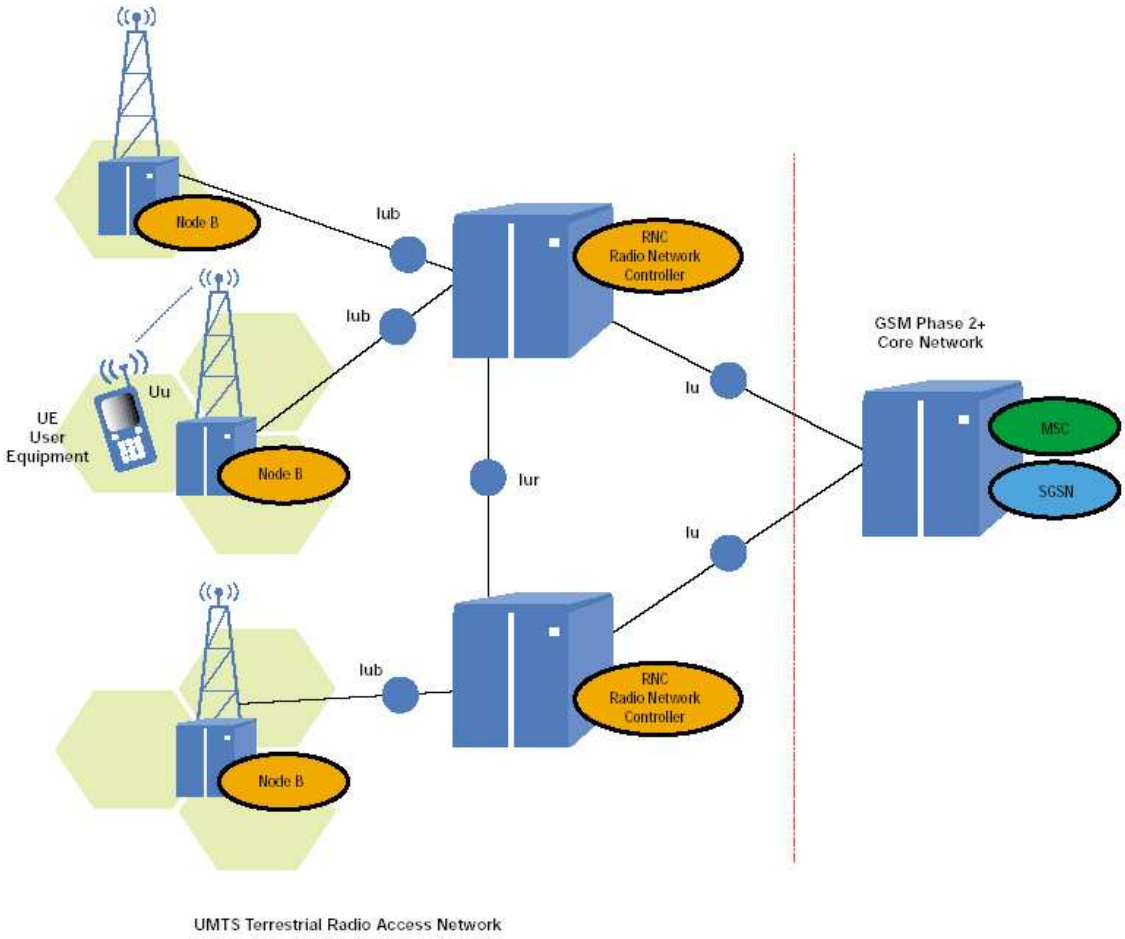


Figure 2.5: UMTS Terrestrial Radio Access Network

cies, so the downlink and uplink frequency bands are allocated separately and in pair.

- **Time Division Duplex** (TDD) The base station and the mobile station transmit alternatively on the same radio frequency.

The radio access network, which is based on the new radio interface, is called *UMTS Terrestrial Radio Access Network* (UTRAN). Referring to Fig. 2.5 [26], the two new network elements in UTRAN are: Radio Network Controller (RNC) and Node B. UTRAN is subdivided into individual radio network systems (RNSs), where each RNS is controlled by an RNC. The RNC is connected to a set of Node B elements, each of which can serve one or several cells.

The RNC performs the same functions as the GSM BSC, providing central control for the RNS elements (RNC and Node Bs). As noted in Fig. 2.6 [26], the functions of

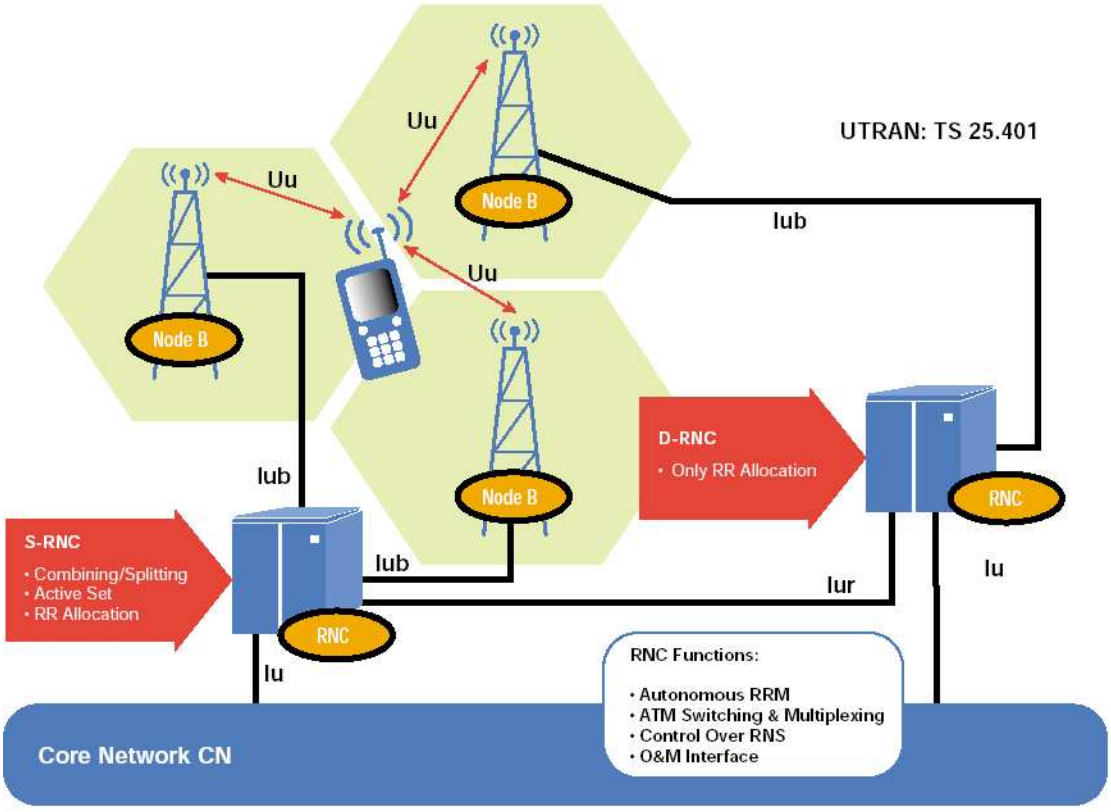


Figure 2.6: Functions of UTRAN elements

RNC are:

- Radio Resource Management (RRM) through the Iur interface, eliminating the burden from the Core Network.
- Protocol exchanges between the Iur, Iub and Iu (CS or PS) interfaces. User data transmissions (CS, PS or multimedia) are multiplexed via these interfaces between CN and UE.
- Centralised Operation and Maintenance of the entire RNS.

Node B is the physical unit for radio transmission/reception with cells. A single Node B can support both FDD and TDD modes, and it can be co-located with a GSM BTS to reduce implementation costs. Node B connects with the UE via the W-CDMA Uu radio interface and with the RNC via the Iub ATM-based interface. The main task of Node B is the conversion of data to/from the Uu radio interface, including channel coding and interleaving, rate adaptation, etc. on the air interface. It also measures the quality and

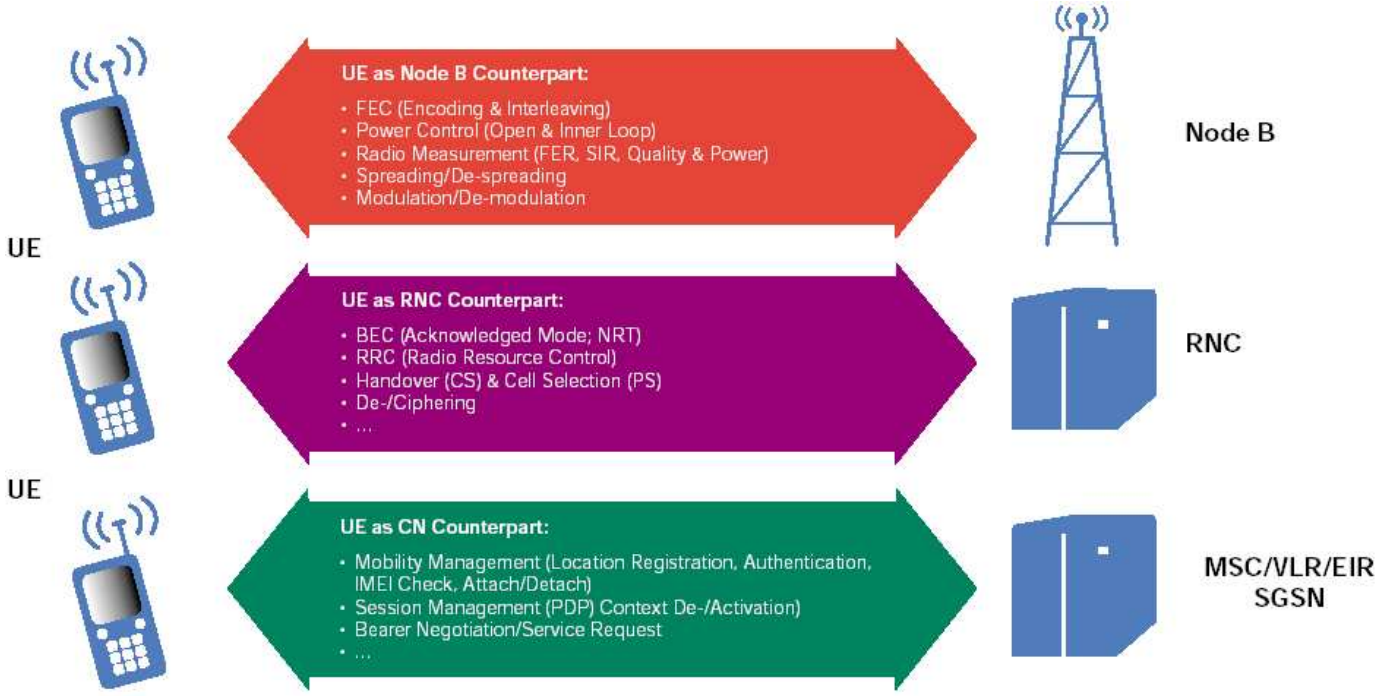


Figure 2.7: Functions of UMTS UE

strength of the connection and transmits related data to the RNC as for handover and macro-diversity signal combining.

The UE, a new synonym in UMTS for the mobile station, is the counterpart to the various network elements in many functions and procedures (Fig. 2.7 [26]). Every UE has a primary connection to a RNS via a Node B, called the Serving RNS (S-RNS). If required, a second RNC can support additional connection to the UE. This second RNC is the Drift RNC (D-RNS). The communication between S-RNC and D-RNC, via the Iub interface, typically occurs during an inter-RNC handover, or a *soft handover*. The D-RNC then becomes the new S-RNC for the UE. The D-RNC may also be involved in macro-diversity signal combining and splitting. So, a UE can only have one S-RNC at a time, but may communicate with either zero, one or more D-RNCs at any given time.

2.2.2 Protocol Stack

The entire protocol architecture is divided into a control plane and a user plane. Actual data transfer is done within the user plane, so we only describe the user plane protocols here. We also restrict the overview to packet-switched data transmission modes.

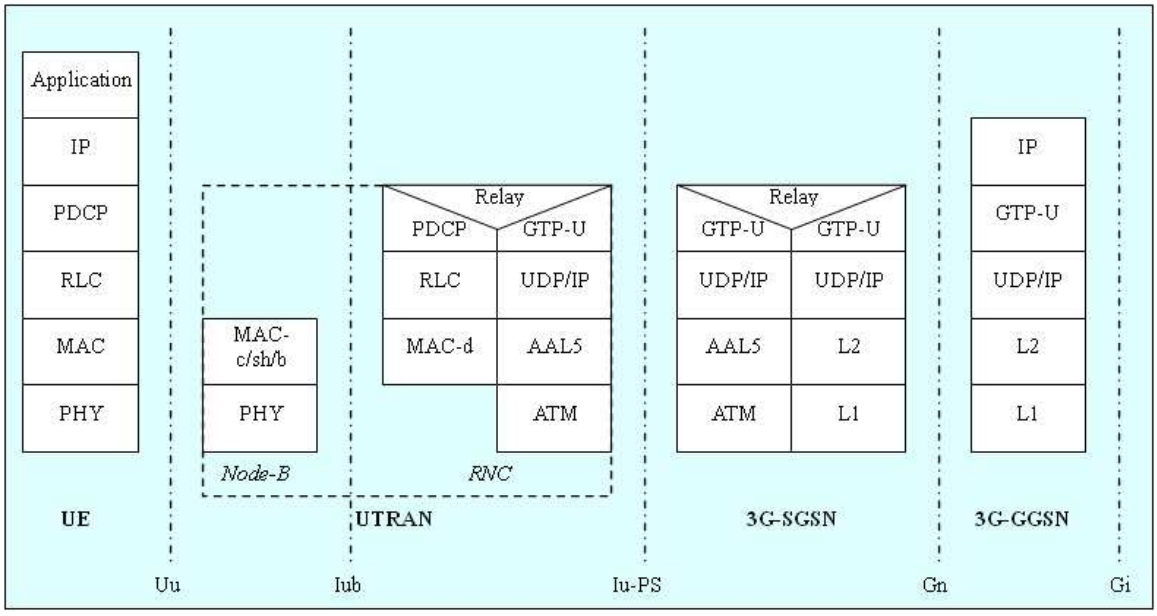


Figure 2.8: UMTS user plane protocol architecture

The user plane of the UMTS protocol architecture is given in Fig. 2.8 [38]. It features few changes from the user plane of GPRS (Fig. 2.3), especially in the CN.

As in GPRS, GTP is used again to tunnel network layer packets between 3G-SGSN and 3G-GGSN. The underlying transport tunnel is UDP/IP, and the lower data link (L2) and physical layer (L1) can be any type of transport technology (ATM, IP-based, etc.).

Protocols in UTRAN

Since 3G-SGSN eliminates the burden of any radio-related functions, it does not need radio protocol layers (i.e., LLC and SNDCP) any more [38]. So the GPRS tunneling protocol (GTP) can further extend to RNC through 3G-SGSN. Below GTP, UDP/IP is applied as the transport layer. 3G-SGSN is physically connected to RNC using ATM, so in between ATM Adaptation Layer 5 (AAL5) is used for segmenting the IP packet into ATM cells.

The Packet Data Convergence Protocol (PDCP) [47] is the interface from the UTRAN protocols to the network layer protocol. It provides protocol transparency to the application protocols (such as IPv4, IPv6, PPP, etc.) over the radio interface. So new protocols can be supported in the future without changes to the radio interface. This is unlike SNDCP, which also provides IP payload compression and so is limited to use with IP.

PDCP only provides IP header compression based on the standard schemes defined in [14]. It then passes the packet down to RLC/MAC. The RLC/MAC layer essentially provides the same functions as in GPRS, with changes to fit with the new radio access network. A comprehensive introduction of these channels may be found in [49].

2.2.3 Quality of Service

Release 99 defines serveral more QoS parameters with finer grained properties in order to meet the requirements on different levels of service for applications, such as delivery order, residual bit error ratio, allocation/retention priority, etc. [49]. Additionally, UMTS defines four distinct traffic classes [47], with different parameters specifying their QoS requirements: *conversational*, *streaming*, *interactive* and *background*. Delay-sensitive services like telephony and telnet belong to the conversational class, while for traffic in the background class no time contraints are specified and it only requires data integrity. The streaming class has a one-way delay limit of less than 10 seconds and is mainly for audio/video streaming and applications like FTP. In the interactive class, data is transferred in a request-response pattern. It is used for voice messaging or web-browsing which requires a shorter delay than applications in the streaming class. For all the traffic classes, the maximum bit rate is 2 Mbit/s, which is limited by the radio interface. More information on UMTS QoS may be found in [47].

2.3 Summary

In this chapter, we have provided a detailed description on certain aspects of GPRS, which are related to the discussions in the following chapters. We have also paid some attention to the new 3G UMTS system.

In the next chapter, we will first give an overview of the basic concepts and algorithms of TCP. We will then look at the problems related to the use of TCP over the wireless links that we have introduced in this chapter. We will also discuss the current works on solving these problems.

Chapter 3

TCP and Its Behavior over Wireless Links

The Transmission Control Protocol (TCP) is a widely used transport protocol designed for reliable delivery of data across various network paths. According to some recent experimental measurements [10], about 90% of the overall IP traffic is carried by TCP, making it by far the most widespread transport protocol. As Internet access through wireless networks takes off, TCP also becomes a natural choice for use over wireless links. Moreover, because of the current dominant usage of TCP in the ordinary networking and Internet access, the compatibility concern with existing systems also demands usage of TCP in wireless Internet access [47].

However, TCP algorithms were mostly developed empirically and were based on assumptions that hold in wired networks but not necessarily in wireless networks. In fact, many problems have arisen in recent years for TCP over wireless links.

In the following sections, we will first provide a brief description of the fundamental concepts and algorithms of TCP. Then, we will identify the main problems of TCP caused by the distinct characteristics of wireless links. We will also have an overview of the existing works on handling those identified problems.

3.1 Basics of TCP

The original specification [41] of TCP dates back to 1981. Many enhancements have since been made over the years, but the basic protocol concepts and assumptions underlying its reliability are still valid and remain unchanged. A comprehensive description of TCP may be found in [46]. The follow-on modifications are explained in detail in the corresponding Request For Comments (RFC), such as TCP Congestion Control in RFC2581 [4], TCP NewReno in RFC2582 [17], TCP Selective Acknowledgment in RFC2018 [35], etc.

TCP provides a connection-oriented, reliable service. It achieves *reliability* by doing the following [46]:

- The data passed down from the upper layer is divided into what TCP considers the best sized chunks to send. The unit of each chunk of data handed over to IP is called a *packet*.
- To inform the other end about the correctly transmitted packets, TCP sends acknowledgments (ACKs). The acknowledgments may not be sent immediately but delayed a fraction of a second.
- When TCP sends a packet, it initiates a timer. If the sender does not receive an acknowledgment for the packet when the timer expires, it will retransmit the packet.
- TCP includes a checksum of its header and data in each packet. This is an end-to-end checksum for detecting any modification of the data in transit. If a packet arrives with an invalid checksum, a TCP receiver discards it without any acknowledgment, but expects the sender to timeout and retransmit.
- TCP packets are transmitted as IP datagrams. Since IP datagrams can arrive out of order, so do TCP packets. A TCP receiver is responsible for re-sequencing the data packets if needed, and then pass the reordered packets to the upper layer.
- A TCP receiver discards duplicate TCP packets.

- TCP also provides flow control to adapt its transmission rate to the available capacity of the connection and the buffer spaces at both ends. This prevents a fast host from overloading a slower host.

3.1.1 TCP Transmission and Acknowledgment

Each TCP packet contains a header and a data portion. A standard TCP header has a normal size of 20 bytes, unless options are present. Basically, it consists of the source and destination *port number* identifying the sending and receiving application at each end host; a *sequence number* specifying the first byte of data the packet represents; an *acknowledgment number* if valid, indicating the next sequence number that the sender of the acknowledgment expects to receive; six flag bits, among which the ACK flag is used to indicate the validity of the acknowledgment number and the SYN flag is for synchronizing sequence numbers to initiate a connection; a *window size* advertising the number of bytes that the sender of the packet is willing to accept; a *checksum* covering the whole packet (header and data), etc. If the header also includes certain TCP options, such as Timestamp, or Selective Acknowledgment etc., it may be extended by another 20 bytes at the most. If TCP uses IP as the underlying network layer, a TCP packet will be encapsulated by IP with a 20-byte IP header.

The TCP acknowledgment number is a *cumulative* value. Cumulative means that the number always acknowledges the sequence number of the highest in-order piece of data that has arrived at the receiver side. If a packet that arrives is the one it expects, a TCP receiver generates a *positive acknowledgment* (ACK) with a sequence number of one plus that of the packet just arrived, i.e., the next sequence of data the receiver is willing to receive. If the incoming packet is either below (a duplicate) or above (an out-of-order packet) the packet it is waiting for, the receiver responds with a *duplicate acknowledgment* (DUPACK) with the same sequence number as the last sent ACK. Without the SACK option [35] (which was added into TCP at a later time), the receiver has no way to inform the sender which (out-of-order) packet has been received corrrectly. There is also no means for a TCP receiver to negatively acknowledge an error packet (such as a

packet that has checksum error and is thus dropped by the receiver).

TCP can handle a full-duplex connection where data can be flowing in each direction, independent of the other direction. But it is possible that at the time a TCP end receives data, it may not have any data to send back, so a pure acknowledgment packet is returned. Such a packet contains only the TCP and IP header, an overhead without any useful data. So, normally a TCP receiver does not respond with an ACK immediately after it receives a packet. Instead, it delays the ACK, hoping to have data going in the same direction as the ACK, so that the ACK can be sent along (or *piggybacked*) with the data. This is called *Delayed Acknowledgment*. Most implementations of TCP use a 200ms delay – that is, TCP will delay an ACK up to 200ms to see if there is data to send with the ACK.

As TCP is a connection-oriented protocol, the two applications using TCP must establish a TCP connection with each other before they can exchange data. A *three-way handshake* is used for connection establishment. It determines unique initial values for the sequence and acknowledgment number as follows: the requesting host sends a packet with its own sequence number and with the SYN flag set; the receiving host knows the acknowledgment number to send, so it responds with a packet containing both its sequence number and the acknowledgment number and with the SYN and ACK flag set. The requesting host then gets to know the receiving host's sequence number, and it sends another packet with only the ACK flag set, acknowledging the number. The successful transmission of these three packets completes the three-way handshake. This process may also negotiate the use of some TCP options, such as the Timestamp option.

3.1.2 TCP Flow and Congestion Control

The basic form of TCP flow control is a *sliding window* protocol. It allows a TCP sender to transmit multiple packets before it stops and waits for an ACK. The packets that can be sent out at one time forms a window, or a *send window*. The protocol is visualized in Fig 3.1 [46].

In the figure, there are 11 packets. Packets 1, 2, 3 have been transmitted by the sender and also acknowledged by the receiver. The send window currently covers six packets.

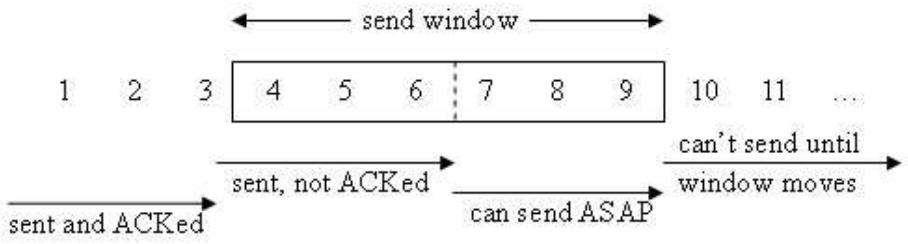


Figure 3.1: Visualization of TCP sliding window

The first three packets (packets 4, 5, 6) have been sent but not acknowledged yet. Now, the sender can still send three more packets (packets 7, 8, 9). As packets at the left edge of the send window are acknowledged by the receiver, the send window slides to the right.

The size of the send window is determined by the minimum of the *advertised window* and the *congestion window*. The advertised window is *advertised* by the recevier to the sender, indicating the amount of available buffer space at the receiver. While the advertised window is flow control imposed by the receiver, the congestion window (cwnd) is flow control imposed by the sender (cwnd is closely related to TCP congestion control, which we will introduce in the next few sub-sections).

In addition to the sliding window protocol, TCP acknowledgments also provide some basic flow control. In the stable state, the spacing of the ACKs arriving at the sender determines the spacing of the outgoing data packets. Since the receiver can only generate ACKs when packets arrive, the spacing of the ACKs at the sender identifies the packet arrival rate at the receiver, which is limited by the bandwidth of the bottleneck in the connection. This is called the ack- or self-clocking behavior of TCP.

Slow Start

When a TCP connection is just established, the TCP sender has no idea about the speed at which it can inject its data packets into the network. It probes for the available bandwith, using an algorithm called *slow start*. The algorithm operates by observing that the rate at which new packets should be injected into the network is the rate at which the ACKs are returned by the other end.

At the start, the sender initializes *cwnd* to one packet [1]. Then, each time ACK is received, the sender increases *cwnd* by one. Being limited by the initial value of *cwnd*, the sender first transmits one packet. When the ACK for this packet is received, it increments *cwnd* from one to two. Now, it can send two packets immediately. When each of the two packets is acknowledged, it further increments *cwnd* to four. As transmission continues, *cwnd* will be eight, sixteen, and so on. As visualized in Fig. 3.2, this is an exponential increase.

Slow start cannot continue forever as the network has limited resource and can only support a bandwidth up to a certain level. So, slow start should be terminated at some later time. We will talk about its termination and other follow-on actions soon.

Congestion

TCP is designed for reliable links. One important assumption it makes is that the packet loss caused by damage is very small (much less than 1%), therefore the loss of a packet signals congestion somewhere in the network between sender and receiver. There are two indications of packet loss: an expiry of the retransmission timer and the receipt of duplicate ACKs.

Retransmission Timeout

As mentioned in the section "TCP Transmission and Acknowledgment", there is no way for TCP to tell the other end about the missing or out-of-order packet. So, to ensure reliability, every outgoing packet is secured by a timer started at the transmission time. If the ACK for a packet is not received when the timer expires, the sender transmits the packet again. This is a *retransmission timeout*. The length of timeout determines how long a TCP sender should wait before retransmitting a packet. It can largely affect TCP performance. If it is too high, loss recovery will be delayed; if it is too low, unnecessary retransmissions occur.

The computation of retransmission timeout (RTO) should relate to the round-trip

[1] The size of the packet is Maximum Segment Size (MSS) in bytes. Although the congestion window is counted in bytes, it is always a multiple of MSS. So we usually refer to its size in units of packets.

times (RTTs) of packets, which reflect the current network propagation and transmission delay. The standard TCP retransmission timer is defined in RFC2988 [39]. It computes the RTO value as follows:

It first sets to an initial value. Then when the first RTT measurement is made, it is re-initialized as:

$SRTT = RTTnew$

$RTTVAR = RTTnew/2$

$RTO = MAX(SRTT + k * RTTVAR, 2 * G)$

The timeout value is then computed with each subsequent RTT measurement as:

$\delta = RTTnew - SRTT$

$SRTT = SRTT + \alpha * \delta$

$RTTVAR = RTTVAR + \beta * (|\delta| - RTTVAR)$

$RTO = MAX(SRTT + k * RTTVAR, 2 * G)$

RTTnew is the new RTT measurement. SRTT is the smoothed RTT estimator, with a smoothing factor α (=1/8). RTTVAR is the smoothed RTT variation estimator, with a smoothing factor β (=1/4). α and β are also called the estimation gains. δ is the difference between the new measured value, RTTnew, and the current estimation, SRTT. SRTT and RTTVAR are updated every time the sender completes a new RTT measurement, and then they are used to calculate the next RTO value. G is the clock granularity, which is operating system dependent. In common TCP implementations, its value is 0.5 second. k is the variation weight, whose value is 4.

This is an adaptive timer where the value of RTO is changed gradually based on measured RTTs. However, RTT measurement cannot be taken during retransmissions, prohibited by *Karn's Algorithm* [30]. This is due to the *retransmission ambiguity* problem where a TCP sender cannot differentiate the ACK of an originally-transmitted packet from the ACK of the packet's retransmit. Moreover, after a timeout, the RTO is doubled so that the sender has to wait for a longer time before the next timeout. This doubling is called the *exponential backoff*. Before the retransmitted packet is acknowledged, every time a subsequent timeout occurs, the RTO is backed off again. This exponential backoff

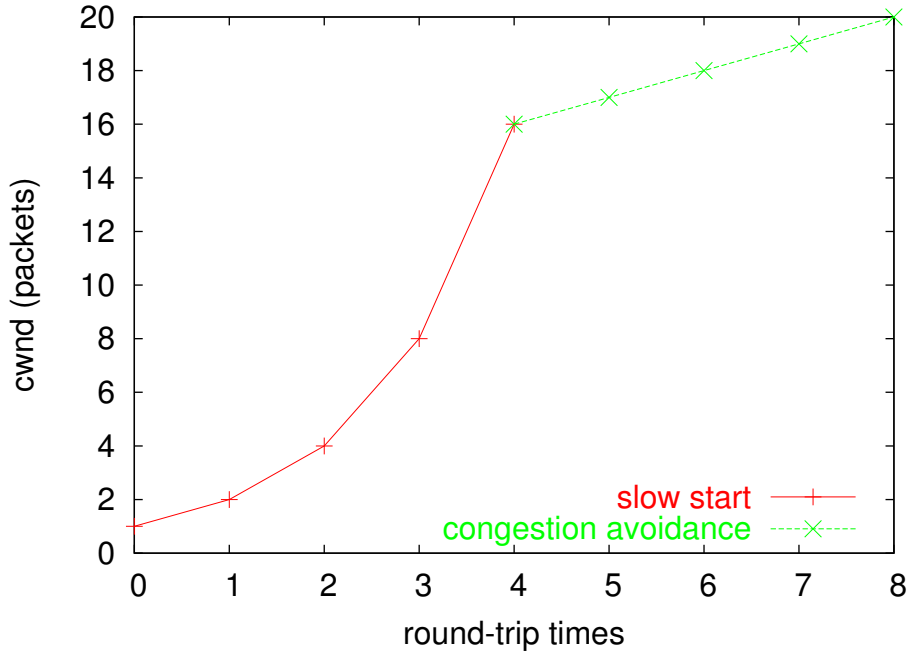


Figure 3.2: Visualization of slow start and congestion avoidance

can provide stability for the protocol.

Congestion Avoidance

Slow start initiates the data transmission of a TCP connection. But at some point, the limit of the network, or more precisely, the limit of an intermediate router is reached, and packets are dropped. This is why timeout occurs. To deal with it, TCP introduces another algorithm called *congestion avoidance*.

In order to determine whether slow start or congestion avoidance should be used to control data transmission, TCP needs *cwnd* and another variable called the slow start threshold or *ssthresh*. In the beginning, *ssthresh* is set to an initial value. This initial value may be arbitrarily high, but as we will see, it will be reduced if congestion occurs. Usually, it is set to be the size of the receiver's advertised window. As slow start controls the transmission, *cwnd* is incremented exponentially, and it will either reach the predefined *ssthresh* or overload the network capacity.

In the first case, once *cwnd* is higher than *ssthresh*, congestion avoidance starts to take over control. Then, instead of being incremented by one on each incoming ACK, the increasing rate of *cwnd* slows down to one packet per RTT, or roughly 1/*cwnd* per

ACK. This is an additive increase (starting from RTT 4 in Fig. 3.2), compared to slow start's exponential increase.

But before reaching *ssthresh*, if *cwnd* is incremented to a value that cannot be supported by the available network resources, congestion occurs and packets are dropped at intermediate routers. At the sender, its timer goes off, indicating that it has reached the upper limit of the network, and needs to slow down. The sender stores one half of the current send window size (and at least two packets) in *ssthresh*. Additionally, it sets *cwnd* back to one packet and starts retransmitting. This is called the *go-back-N retransmission*, as the sender retransmits all the unacknowledged packets starting from the oldest one. The sender needs again to probe the network for the available capacity. Later, when the retransmitted packet is acknowledged by the receiver, the sender increases *cwnd*. However, depending on the current values of *cwnd* and *ssthresh*, it may follow slow start or congestion avoidance's increasing rule.

Fast Retransmit and Fast Recovery

A TCP receiver is required to generate an immediate DUPACK when an out-of-order packet is received. This DUPACK is aimed at letting the sender know that a packet has been received out of order, and to tell it what sequence number is expected. In addition, the receiver should also send an immediate ACK when the incoming packet fills in all or part of a gap in the sequence space of the recevier buffer. These DUPACKs can provide more timely information to the sender, which enables it to use the *fast retransmit* algorithm to detect and repair loss and to avoid a long wait before a retransmission timeout.

Since the sender does not know whether a DUPACK is caused by a dropped packet or a reordering of packets, it is assumed that if there is just a reordering of packets, there will be only one or two DUPACKs before the reordered packet arrives and triggers a new ACK. So, the fast retransmit algorithm uses the arrival of three consecutive DUPACKs (i.e., four identical ACKs without the arrival of any other intervening packets) as an indication of packet loss. After receiving three DUPACKs, the sender immediately performs

a *fast retransmit* of the oldest unacknowledged packet.

Following the fast retransmit, another algorithm called *fast recovery* takes control of the new data transmission until a non-duplicate ACK arrives. In addition to indicating packet loss, the arrival of DUPACKs also reveals that some packets have left the network (it is these packets that trigger the DUPACKs). The network resources previously occupied by those packets can now be used. Moreover, the ACK-clocking is preversed. So, it is reasonable and meaningful for the TCP sender to continue transmitting new data.

The arrival of DUPACKs is a less severe signal of congestion than a timeout, since there are still packets/DUPACKs flowing in the network. So fast retransmit and fast recovery together can implement a more efficient loss recovery than the slow-start retransmission after a timeout: upon receiving the third DUPACK, *ssthresh* is set to half of *cwnd*. *cwnd* is then set to be *ssthresh* plus 3*MSS (as three DUPACK have arrived), and is incremented by one per subsequent DUPACK, which indicates the left of a packet from the network. If allowed by the new value of cwnd and the receiver's advertised window, the sender can transmit new packets. When the first non-duplicate ACK after the retransmit arrives, *cwnd* is reset back to *ssthresh*. Then, from now on the sender exits fast recovery and directly enters congestion avoidance.

Slowstart, congestion avoidance, fast retransmit and fast recovery form the foundation of TCP congestion control. A more detailed description of them may be found in [4]. The most widely used TCP version today is TCP Reno [2], which implements all the four algorithms. An important modification of it is TCP NewReno [17], where TCP is allowed to respond to *partial acknowledgments* [3] with retransmits while remaining in fast recovery. This can improve TCP performance during multiple packet losses in a single window.

[2] It is named according to its first implementation in the Berkeley Software Distribution, BSD Unix.

[3] When multiple packets are dropped in a window, the ACK for the retransmitted packet will acknowledge some but not all of the packets transmitted before the fast retransmit. This ACK is known as a partical acknowledgment.

3.2 TCP over Wireless Links

In the previous section, we have briefly discussed the basic concepts and algorithms of TCP. We have also noted that as TCP was originally developed for wired networks, some of its design assumptions could be violated in wireless links.

3.2.1 Wireless Link Characteristics

Here, we will first look at the link characteristics of 2.5G/3G wireless systems.

Latency

The latency of 2.5G/3G links is high compared with wired networks. This is mostly due to the limited transmission capacity in radio access networks and the extensive processing required at the physical layer of these links for FEC (refer to Appendix A.1.2), interleaving, etc. Local retransmissions on the link layer can also introduce extra delays. Typically, the RTT of a TCP packet flowing through one such link can vary from a few hundred milliseconds (ms) to one second.

Bandwidth and Bandwidth Asymmetry

Referring to Table 2.1 in Section 2.1.2, the theoretical maximum data rate over a GPRS link is 171.2 kbps. However, since a mobile terminal is seldom allocated all the eight timeslots, data rates in 2.5G GPRS networks are normally between 10 to 40 kbps in downlink and 10 to 20 kbps in uplink [27]. In the initial 3G systems, such as Enhanced Data Rates for GSM Evolution (EDGE) [49] and UMTS, higher data rates can be achieved: up to 384 kbps in downlink and around 64 kbps in uplink. With further improvements, UMTS may reach an even higher data rate of up to several Mbps (as suggested in its latest releases).

2.5G/3G systems usually run asymmetric uplink and downlink data rates. Asymmetry is inherited in these wide-area wireless system. This is because mobile terminals often have to contend with each other to gain access to the shared up-channel. Uplink is

also limited by the battery power consumption of a terminal. As discussed in previous sections, the performance of TCP relies on feedback in the form of cumulative ACKs from the receiver to ensure reliability. In addition, TCP is ACK-clocked, relying on the timely arrival of ACKs, to make steady progress and to fully utilize the available bandwidth of the path. Any limitation or disruption in the feedback path could potentially impair the performance of forward TCP transmission. However, in reality, bandwidth asymmetry does not exceed three to six times [24]. For Internet applications such as web access, FTP, etc., downlink is the main data traffic path and uplink usually contains only ACKs acknowledging the arrival of data at mobile terminals. A pure acknowledgment containing only the TCP/IP header (or even with options) can be no more than 100 bytes while a TCP/IP data packet usually ranges from several hundreds to one thousand and a half bytes. In addition, delayed acknowledgment, if used by the receiver, can further reduce the amount of ACK traffic along uplink. So, bandwidth asymmetry in 2.5G/3G links can be tolerated by TCP without the need for any explicit ACK control techniques such as ACK filtering [5].

Error Losses and Corruption

Due to the intrinsic properties of radio interface, wireless links were originally characterised as a transmission media with high non-congestion loss. As TCP congestion control algorithms (refer to Section 3.1.2) infer packet losses as indication of network congestion, such non-congestion losses can incorrectly trigger TCP congestion control and lead to transmission rate reduction in TCP-based applications. However, in current 2.5G and the coming 3G systems, error losses are not a main concern any more because these systems have already incorporated error protection mechanisms like FEC and ARQ into their link layers. We have discussed the background information on link layer protocols in the previous chapter, along with detailed explanation of different error protections in Appendix A.1.2. We will discuss the interaction between wireless link layer retransmissions and TCP shortly.

Delay and Bandwidth Variation

A sudden increase in the latency of the communication path is called a *delay spike*. 2.5G/3G links are likely to experience delay spikes exceeding the typical RTT by several times [27] due to reasons like: link layer retransmission, handovers (see later), or arrival of high priority traffic on shared channels. TCP uses its retransmission timer for congestion control and loss recovery. The timer is set according to the measured path RTTs, and its value is gradually adapted with the change in RTTs. When facing abrupt delay variations over wireless links, the timer expires and incorrectly triggers retransmission timeout, which leads to unnecessary retransmission and a reduced transmission rate.

Because radio resource is dynamically allocated in 2.5G/3G systems, according to actual demand, an MS may experience bandwidth variation or bandwidth oscillation from time to time due to radio channel scheduling. Bandwidth variation also happens when an MS moves from a fast/slow network into a slow/fast one. When bandwidth varies from high to low, the packet transmission times (along with packet RTTs) increase suddenly, and the low RTO value previously adapted causes spurious timeouts. Periods of low bandwidth can also result in congestion [21]. Bandwidth variation from low to high could result in underutilization of the link.

Larger delay and bandwidth variations can also cause bursty ACK arrivals (also called ACK compression [13]). Since TCP uses ACK-clocking to control packet sending, bursty ACK arrival leads to the release of a burst of packets. When the packet burst arrives at the bottleneck link with bandwidth or delay burst, multiple packet losses could result.

Handovers

Handovers can happen within a single cellular network at different mobility management levels, or among different networks such as when an MS moves from a GPRS network into a UMTS network in a process called a *intersystem handover*. Some have been illustrated in Section 2.1.5 with typical scenarios.

Handovers can cause delay and/or bandwidth variations which trigger spurious time-outs on the TCP layer. Furthermore, as shown in the GPRS MM scenarios, during an inter-SGSN handover, the old SGSN bufferring the unacknowledged data may run out of buffer space before the handover is completed. T unacknowledged data would then be dropped, and TCP (at the upper layer) would need to recover the packet losses.

On-demand Resource Allocation

Wireless channels are dynamically allocated to mobile terminals based on their actual needs. Resource allocation involves the communication of control signals between different wireless network nodes. Due to the high latency over wireless links, resource allocation is forced to introduce an additional delay.

Packet Reordering

Packet reorderings may happen during link layer retransmissions. Significant reorderings can incorrectly trigger TCP congestion control and retransmission. Currently, packet reordering is disabled in 2.5G/3G wireless links, so at the moment, it is not a major concern. However, out-of-order delivery of packets may be beneficial for unreliable real-time applications by decreasing delays. Thus if TCP can be made considerably more robust to reordering with an acceptable cost, future development of wireless technologies may enable packet reorderings to provide better services for real-time services [21].

Regarding the problems caused by wireless links into existing TCP implementations, we cannot force TCP to completely adapt to wireless characteristics, or accept TCP as it is and require future wireless systems to match the design assumptions of TCP. Intuitively, there should be an interplay or compromise from both sides, depending on the characteristics of each problem.

3.2.2 Interactions between TCP and Link Layer Retransmission

TCP was designed for reliable links and it assumes low non-congestion loss over the underlying links. So high packet corruption in old wireless links could incorrectly trig-

ger TCP congestion control and degrade TCP performance. In the current and future wireless systems, this problem has been greatly mitigated by the employment of several error protection mechanisms in the wireless link layers (see Chapter 2).

GPRS has both a FEC and a ARQ capability in its LLC layer between mobile terminals (MS) and the SGSN. Retransmission protection comes into effect during handovers (as in the scenarios in Appendix A.2.4) where the SGSN keeps the unacknowledged frames and re-routes them to the new MS or SGSN during a cell change. The lower RLC layer is introduced mainly for providing a reliable link between MS and BSS. It also includes a selective ARQ mechanism for transmitting error blocks. UMTS W-CDMA incorporates FEC and ARQ mechanisms similar to those of GPRS. These link layer FEC and ARQ can ensure a much more reliable wireless transmission with a negligibly small probability of undetected error, and a low level of loss for the upper-layer traffic [27]. As verified in some study [48], when the packet loss rate over wireless links is one-order of magnitude lower than the one in wired networks, TCP performance is essentially not affected by the wireless impairment. In addition, as link layer retransmission is done locally and in small units, it is generally more efficient than TCP retransmission.

Link layer retransmission needs to take a certain amount of time, so instead of packet losses, TCP will experience a delay jitter during the transmission, which may trigger spurious timeouts. Link layer retransmission may also introduce packet reorderings at the transport layer (TCP). However, packet reordering is generally not allowed in the current systems. For example, UMTS RLC explicitly preserves the order of packet delivery [27]. Moreover, 2.5G/3G systems generally employ the higher-performance selective ARQ, so competition between two retransmissions at different layers (link layer and TCP) is no longer a serious problem [47].

However, packet losses can still happen over current wireless links due to handovers, though not error/corruption. This kind of packet losses is caused by buffer overflow at the SGSN, and is therefore congestion loss.

3.2.3 Proposed Solutions in TCP

Instead of fixing the problems by improving wireless technologies (as the case of link layer retransmission), a lot of work has also focused on adapting TCP to wireless links.

Among them, quite a number of the proposed schemes are aimed at improving TCP performance in the presence of frequent packet losses in wireless links. In Indirect-TCP [9], a TCP connection is split into two independent connections, one over the fixed network and the other over the wireless link. The second connection can recover from losses over wireless links quickly. In the Snoop protocol [6], the network layer at the base station is modified so that packets can be cached at the base station and local retransmission can be performed only in the wireless link. Cumulative Explicit Transport Error Notification (CETEN) [16] tries to mitigate the impact of corruption-based loss by not attempting to derive the cause of specific packet losses but using aggregate information from the network to ensure TCP's long-term average sender rate. As link layer enhancements for reducing wireless link losses including ARQ and FEC are already part of the 2.5G/3G wireless systems, these schemes for TCP only provide overlapping functions that introduce little performance improvements. For example, Snoop TCP is reported [47] to not work well over GPRS because high delay over the GPRS radio interface can trigger duplicate retransmissions in the Snoop agent.

There have also been works that examine the impact of wireless link delay or bandwidth variations on TCP performance [32], [8], [45], [13], [50]. As we have discussed above, these variations cause TCP performance degradation due to spurious timeouts and multiple packet losses at the congested router buffer. The *spurious timeout* problem is a common consequence shared by several wireless characteristics, including handover, resource allocation, link layer retransmission, etc. It is the main problem that TCP is posed with by current wireless systems. In our opinion, it is worth looking at this problem in more detail. We will also try to mitigate the damage of multiple packet losses in TCP.

3.3 Spurious Retransmission

Spurious retransmission [32] refers to the case that TCP's congestion control and loss recovery algorithms are falsely triggered when in fact there are no genuine packet losses at all. As the four intertwined TCP congestion control algorithms (which have been discussed in Section 3.1) provide two basic mechanisms for packet loss detection and recovery: *timeout retransmission* and *DUPACK-based retransmission* (as discussed in Section 3.1.2), there are two kinds of spurious retransmissions.

Because of the frequent wireless link variations due to various causes, *spurious timeout* is a common phenomenon in TCP connections spanning wireless links. *spurious fast retransmit* is less frequent where it is mostly caused by packet reorderings due to link layer retransmissions. Packet reordering is deliberately restrained in current 2.5G/3G systems but with the possibility of being re-enabled later (see previous section). As far as we know, there is no numerial measurement capturing the occurrence of packet reorderings in wireless links either. Therefore we will turn our focus in the following sections on spurious timeouts. We also pay some attention to spurious fast retransmits.

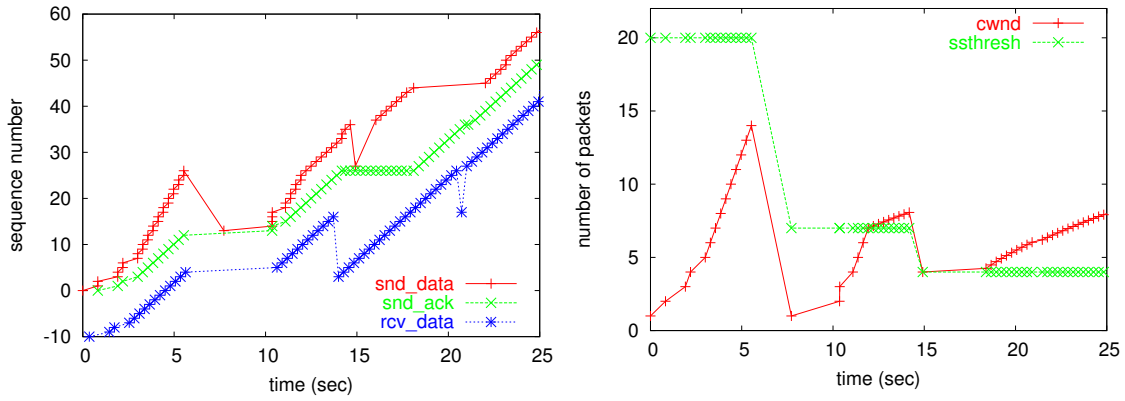
3.3.1 Problem Formulation

Spurious Timeout

As presented in Section 3.1.2, RTO is a prediction of the upper limit of RTTs seen in the current connection. In common TCP implementations, an adaptive retransmission timer [39] accounts for RTT variation. A spurious timeout occurs when the delay in the data and/or the ACK path suddenly increases to the extent that it exceeds the RTO that has been determined a priori. The spurious timeout would not have occurred had the TCP sender waited longer.

Fig. 3.3 shows the time sequence [4] and the corresponding changes in congestion control state when a spurious timeout occurs. Both figures are generated using TCP Reno sender agent in the NS-2 network simulator [37]. We will elaborate on NS-2 in

[4] In Fig. 3.3(a) and subsequent time-sequence plots in the thesis, all the sequence numbers of *rcv_data* are reduced or shifted down by some units (usually by 5 or 10) for a clearer presentation.



(a) Time sequence

(b) Congestion control state

Figure 3.3: A spurious timeout

Chapter 5.

Spurious timeouts affect TCP performance severely as follows: when a spurious timeout occurs (at the 7th second in Fig. 3.3(a)), the TCP sender has to assume that all the outstanding packets are lost and will then retransmit them unnecessarily. Ludwig and Katz pointed out in [32] that the fundamental cause for this go-back-N retransmission is the retransmission ambiguity problem (introduced in Section 3.1.2). Shortly after the timeout, ACKs for the delayed, original transmits arrive at the sender one at a time (starting from the 10th second in Fig. 3.3(a)). But the sender can only assume that they are in response to the retransmits and the unnecessary retransmission continues until all the preassumed lost packets have been retransmitted (from the 10th to the 12th second in Fig. 3.3(a)). From the plot of *rcv_data*, we can see that all the packets, both the original packets and their retransmits, are in fact correctly received.

The unnecessary go-back-N retransmission may cause another problem: the receiver generates a DUPACK for every packet received more than once, and DUPACKs arriving at the sender would incorrectly trigger a fast retransmit (at the 15th second in Fig. 3.3(a)). Note that this fast retransmit can be avoided if either *bugfix* [17] or SACK [35] is enabled.

In response to the timeout, the TCP sender also reduces *cwnd* and *ssthresh*. As shown in Fig. 3.3(b), the default initial values for *cwnd* and *ssthresh* in NS-2 are 0 and 20 respectively. From time 0, as packets are transmitted correctly, *cwnd* increases exponentially. When the timeout occurs around the 8th second, *ssthresh* is reduced to 7 (at

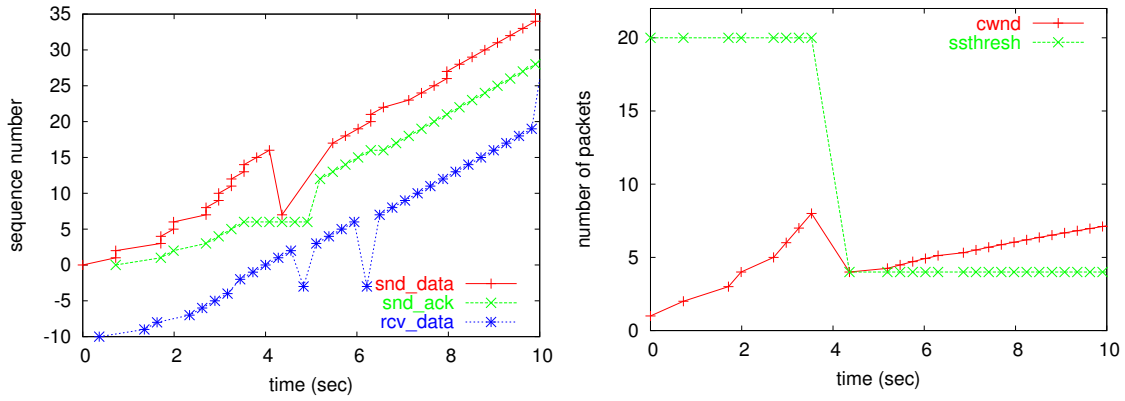
the 8th second in Fig. 3.3(b)) – half of the current value of *cwnd*, which is 14. When the first non-duplicate ACK after the timeout arrives at the 10th second, *cwnd* is set to 1. This is the standard slow start and congestion avoidance process (see Section 3.1.2). With the fast retransmit at the 15th second, *cwnd* and *ssthresh* are further reduced to 4 (at the 15th second in Fig. 3.3(b)). These congestion control reductions are totally unnecessary since there is in fact no congestion loss at all. This leads to an underutilization of the available network capacity and thus largely degrade TCP performance.

Assuming that none of the outstanding packets and their corresponding ACKs were lost, the packets would get retransmitted unnecessarily. While original transmits are probably queuing at and draining from the bottleneck link, retransmits are being sent in slow start. Thus, for each packet that leaves the network and that belongs to the original flight, two duplicate retransmits are injected into the network in response (assuming the receiver generates one ACK per packet). If the bottleneck queue size is limited, this aggressive sender behavior aggravates the problem and may lead to real packet losses due to congestion.

Spurious Fast Retransmit

Link layer retransmission may cause out-of-order data delivery to the upper layer, i.e., the IP layer. IP is a connectionless protocol, which does not guarantee in-order delivery of packets. So, the reordering packets are pushed up further to the transport layer. In response to each reordering packet, a TCP receiver generates a DUPACK. In such a so-called *packet reordering* event, the number of reordering packets that arrive before the in-sequence packet is the *reordering length*. If the reordering length is equal to or greater than three, and at least three of the resulting DUPACKs arrive at the sender, a *spurious fast retransmit* will be triggered.

Spurious fast retransmits affect TCP performance in that the TCP sender incorrectly retransmits the oldest outstanding packet and reduces its sending rate by half, following the fast retransmit and fast recovery algorithms. Fig. 3.4 illustrates the consequences of a spurious fast retransmit. In order to trigger a spurious fast retransmit in the simulated



(a) Time sequence

(b) Congestion control state

Figure 3.4: A spurious fast retransmit

connection, we use a tool called *hiccup* available in NS-2 [42], which can cause a packet reordering event with a predefined reordering length. (A more detailed description of *hiccup* will be given in Section 4.2.) In this example, the reordering length is 5. Packet 7 is queued by *hiccup* while the succeeding five packets are let through. Then the single packet in queue is sent. In Fig. 3.4(a), we can see five DUPACKs generated by the receiver arriving at the sender and triggering the spurious fast retransmit at the 4th second. In the meantime, the sender halves its load and enters congestion avoidance as shown in Fig. 3.4(b).

Retransmission ambiguity is again the essential reason behind spurious fast retransmits: on receipt of the first non-duplicate ACK after a series of DUPACKs, the sender must interpret this ACK as having been triggered by the fast-retransmitted packet but in fact it was triggered by the reordered original packet.

Note that throughout the thesis, we use the term *spurious retransmission* when it does not make a difference whether it is a spurious timeout or a spurious fast retransmit.

3.3.2 Related Works on Spurious Retransmission

Both spurious timeouts and spurious fast retransmits will cause suboptimal TCP performance because of the following symptoms:

- The TCP sender unnecessarily adjusts the TCP congestion control parameters and

reduces its sending rate.

- After a spurious timeout, the TCP sender often retransmits several packets unnecessarily because after the first retransmit, the ACKs for the original transmits appear at the sender one at a time, triggering further retransmits. Often a full TCP window is retransmitted quite unnecessarily.

No known way can effectively prevent such unnecessary retransmissions from happening in the first place. However, we can do some recovery after a retransmission to reduce its toll on TCP performance. In the area of fixing spurious retransmissions, there are three categories of mechanisms:

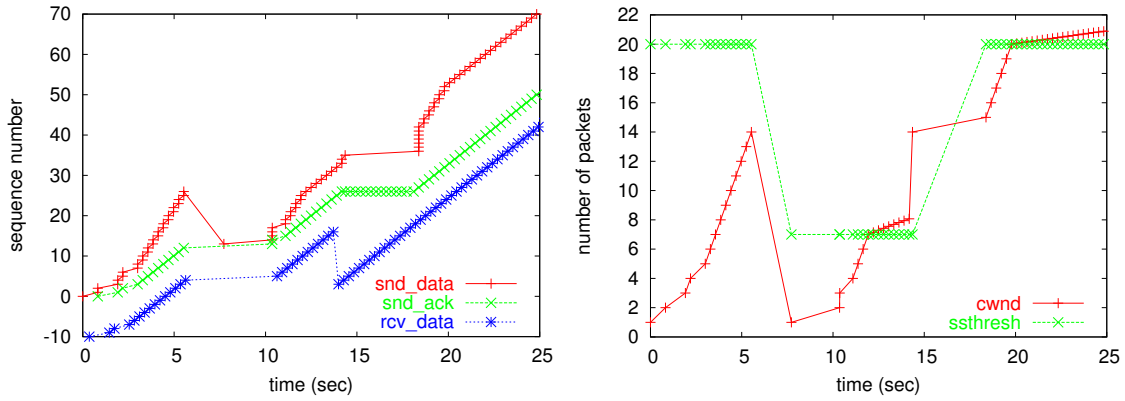
- Mechanisms for detecting spurious retransmissions
- Mechanisms for undoing needless changes to TCP congestion control state from values saved before the retransmissions
- Mechanisms for making TCP more tolerant to variable delays or packet reorderings such that future spurious retransmissions can be avoided

While the first category deals with detection, the second and the third categories responds to the spurious retransmissions. Several algorithms for TCP have been proposed for handling spurious retransmissions, and they include at least the first two categories of mechanisms. Different algorithms may have different means of detecting spurious retransmissions, but they may use the same mechanism for restoring the congestion control state. Some of the algorithms also incorporate mechanisms for adapting either the TCP retransmission timer or the value of *dupthresh*.

Here, we will give an overview of three such algorithms: the DSACK-based algorithm, the Eifel algorithm and the F-RTO algorithm, and we will investigate the pros and cons of each algorithm in detail.

The DSACK-based Algorithm

The TCP Selective Acknowledgment (SACK) option [35] is used for acknowledging out-of-sequence data not covered by TCP's cumulative acknowledgment field. Duplicate



(a) Time sequence

(b) Congestion control state

Figure 3.5: A spurious timeout using TCP SACK with DSACK

SACK (DSACK) [18] is an extension to SACK, specifying the use of the SACK option for acknowledging duplicate packets. In RFC2883 [18], the suggestion is that when duplicate packets are received, the first block of the SACK option can be used to report the sequence numbers of the packet that triggered the ACK.

Duplicate packets can be caused either by a spurious retransmit sent by the TCP sender or by some quirk in the network that causes packet duplication. Paxson shows in [40] that packet duplication by the network is exceedingly rare. A DSACK block therefore has a high probability of reporting a spurious retransmission. To ensure the correctness of the detection, the TCP sender can verify that the packet reported as arriving multiple times has actually been retransmitted.

Once the sender detects a spurious retransmission by using the DSACK information, it can restore the reduced congestion control parameters (*cwnd* and *ssthresh*) from values saved before the retransmission. In case of a spurious fast retransmit, the sender can also adjust *dupthresh* to prevent such unnecessary fast retransmit in the future. [7] discussed several possible ways for adapting the value of *dupthresh*. However, this DSACK-based approach has no way to adapt the retransmission timer after a spurious timeout.

Plots of the time-sequence and the corresponding congestion control changes in Fig. 3.5 show how a DSACK-enabled TCP handles a spurious timeout. The plots are generated from simulation traces using NS-2 [37]. We will discuss in greater detail NS-2, the implementation of DSACK detection and response in NS-2, and the experiment settings

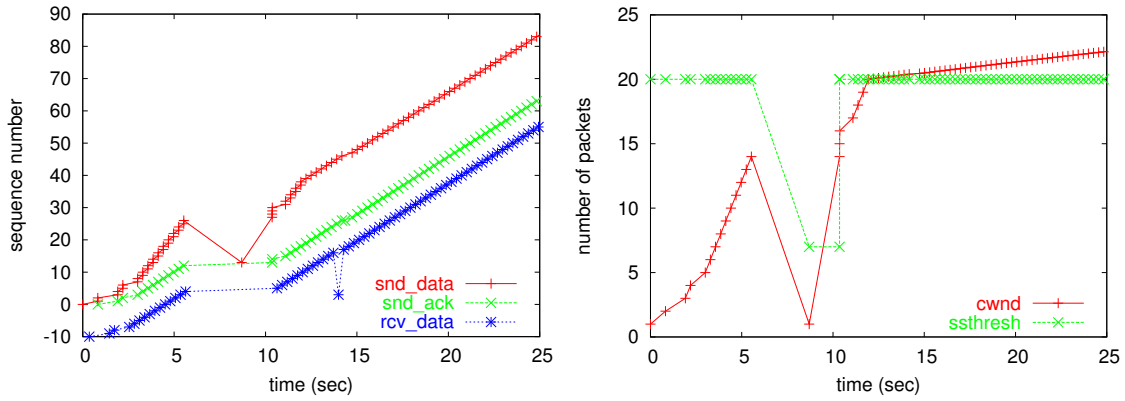
in Chapters 5 and 6.

Because the first ACK carrying a DSACK block arrives at the sender only after loss recovery has already terminated, this approach cannot avoid the go-back-N retransmission. In Fig. 3.5(a), the sender sends this series of retransmits from the 10th to the 14th second. In fact, before the 14th second, Fig. 3.5(a) is the same as Fig. 3.3(a), which shows a common TCP without any detection of spurious timeouts. However, as Fig. 3.5(a) shows TCP SACK, the spurious fast retransmit which would be triggered by the series of DUPACKs with TCP Reno is avoided. When receiving the ACK with the DSACK block, the sender detects the unnecessary timeout and restores the congestion control state. We can see that the transmission rate, i.e., the slope of *snd_data*, in Fig. 3.5(a) after the retransmission is much faster than its counterpart of the common TCP in Fig. 3.3(a). By the 25th second, more than 10 (70 minus 56) packets have been transmitted. The restoration can also be seen in Fig. 3.5(b) after the 15th second, where *ssthresh* is set back to 20 and *cwnd* is set back to 14.

A single DSACK notification is sent in one ACK for each duplicate packet that arrives. Because DSACKs are only sent once, this DSACK-based approach is quite vulnerable to ACK losses. That is, if an ACK containing DSACK information is dropped or corrupted by the network, the information about that particular packet is lost and the sender will never detect the spurious retransmission.

The Eifel Algorithm

The Eifel algorithm [32] suggests that the TCP sender includes some extra information in every packet sent, indicating whether the packet is transmitted for the first time, or whether it is a retransmit. When the information is echoed back in the ACK, the sender is able to eliminate retransmission ambiguity (see Section 3.1.2). It can then determine whether the original packet has arrived at the receiver and declare the retransmission either genuine or spurious. Based on this knowledge, the sender can either continue in the conventional way assuming a real packet loss, or revert the congestion control reductions and continue with new data. The latter alternative is likely to be the correct action to take



(a) Time sequence

(b) Congestion control state

Figure 3.6: A spurious timeout with Eifel

when the original packet is acknowledged after the retransmission, indicating a spurious retransmission.

Currently, the Eifel algorithm uses the TCP Timestamp option [29] as the piece of *extra information*. It works as follows:

Given that timestamps are enabled for a connection, the TCP sender always stores the timestamp of the first retransmit sent at the beginning of a retransmission (caused by either a timeout or DUPACKs. Once the first acceptable ACK that acknowledges the retransmit arrives, the sender compares the value of the echoed timestamp in the ACK with the stored value. If the echoed timestamp is smaller than the stored one, the sender can determine that the retransmission is spurious. In response, as the sender detects it upon the first acceptable ACK, it can avoid the subsequent go-back-N retransmission. Because timestamps are present in all the packets and ACKs, the sender is also able to adapt the conservativeness of the retransmission timer for avoiding future spurious timeouts. Likewise, it adapts *dupthresh* in response to a spurious fast retransmit. In both cases, the algorithm also restores the congestion control state. So the penalty of a spurious timeout or fast retransmit is reduced to just a single unnecessary retransmit. However, the exact way for congestion control state restoration, retransmission timer adaptation or *dupthresh* adaptation may vary.

Fig. 3.6 shows how the algorithm works. (Information about the implementation of the Eifel algorithm in NS-2 will be given in Section 5.2, and the experiment settings

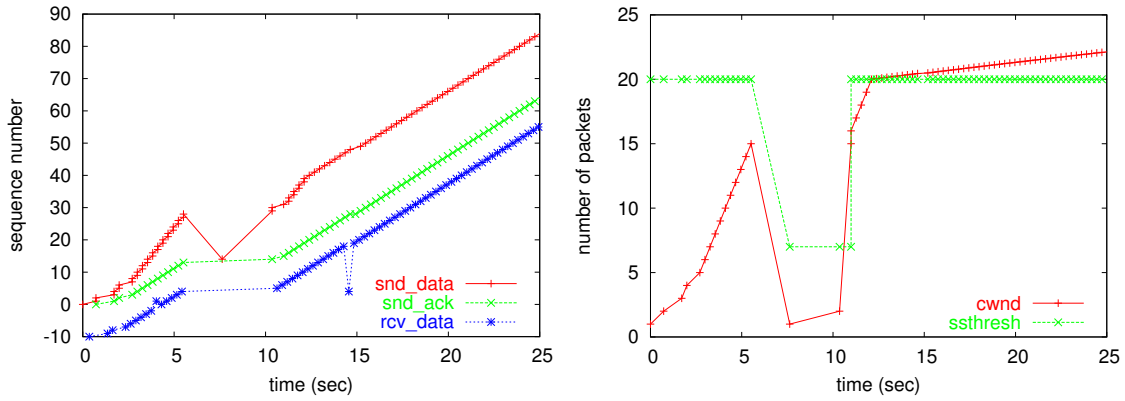
will be listed in Chapter 6.) In Fig. 3.6(a), by detecting the spurious timeout upon the first acceptable ACK, the TCP sender starts to transmit from the next unsent packet. Comparing with the common TCP in Fig. 3.3(a) and TCP with DSACK in Fig. 3.5(a), TCP with Eifel avoids the series of unnecessary retransmits and the consequent fast retransmit. It also transmits the maximum number of packets by the 25th second. In the corresponding congestion control changes in Fig. 3.6(b), the sender resets *cwnd* and *ssthresh* to their previous values, allowing it to utilize more bandwidth. By the 25th second, *cwnd* has reached 22 while in Fig. 3.3(b) and Fig. 3.5(a), *cwnd* is only 8 and 21 respectively at that point.

One advantage of the current implementation is that the detection is quite robust against ACK losses as there is a window worth of ACKs all carrying timestamps that are smaller than the stored value, so the arrival of any one of them would be sufficient to help the sender make the correct decision. Another advantage is that with the timestamp in every packet, the sender is able to get RTT measurements for all acknowledged packets, even for retransmits, allowing a more up-to-the-time RTO estimation.

F-RTO: Forward RTO Recovery

Forward RTO Recovery (F-RTO) [45] is a proposal for only addressing spurious timeouts by modifying the TCP sender. It does not require the use of any TCP options; instead, it uses a set of heuristic rules for detecting spurious timeouts. It works as follows:

When a timeout occurs, the F-RTO sender retransmits the oldest outstanding packet normally. But deviating from normal timeout recovery, it then tries to transmit new, previously unsent data for the first ACK that arrives after the timeout given that the ACK acknowledges new data. If the second ACK that arrives after the timeout also advances the window, the sender can declare the timeout spurious, exit timeout recovery, and restore its congestion control state. However, if either of the two ACKs is a DUPACK, there would be no sufficient evidence of a spurious timeout. Therefore, the sender retransmits the unacknowledged packets in slow-start in a manner similar to traditional



(a) Time sequence

(b) Congestion control state

Figure 3.7: A spurious timeout with F-RTO

TCP implementations.

The plots in Fig. 3.7 show how F-RTO handles a spurious timeout. In Fig. 3.7(a), upon the receipt of the first ACK (just after the 10th second), the sender responds with two unsent packets. When the second ACK arrives (at the 11th second), the sender detects the spurious timeout and exits timeout recovery. In Fig. 3.7(b), we can see the difference between F-RTO and Eifel clearly. The congestion control state is restored at a later time in response to the second ACK.

After detecting a spurious timeout, F-RTO cannot make any active attempt in adapting the conservativeness of the TCP retransmission timer. However, it has a side-effect on RTT measurement. With F-RTO, the TCP sender avoids most retransmits in the go-back-N retransmission, and is able to take RTT samples on the delayed packets. If normal timeout recovery is used, without timestamps, this would not be possible due to retransmission ambiguity. As a result, the estimated RTO value is likely to have a more accurate and larger values with F-RTO than with normal TCP. However, because F-RTO does not solve the retransmission ambiguity problem, it still suffers from the problem as normal TCP does when a genuine timeout occurs – neither of them would be able to sample the retransmits because of Karn's Algorithm (see Section 3.1.2).

Moreover, F-RTO does not always make the right detection [44]. For example, if packet reordering or packet duplication occurs in the packet that has triggered the spurious timeout, F-RTO may not detect it as spurious. Additionally, if a spurious timeout

occurs during fast recovery, F-RTO often cannot detect it. In such cases, F-RTO simply follows standard TCP behavior. In [44], a SACK-enhanced version of F-RTO is specified. This enhanced version is able to detect spurious timeouts in most cases when packet reordering or packet duplication is present, or when the TCP sender is under loss recovery. Its difference from the basic F-RTO algorithm is that the sender may declare a timeout spurious even when DUPACKs follow the timeout, if the SACK blocks acknowledge new data that has not been transmitted after a timeout.

Comparison of the Three Algorithms

The three algorithms presented above work differently in handling either spurious timeouts or both spurious timeouts and spurious fast retransmits. Here, we attempt a comprehensive comparison of DSACK, Eifel and F-RTO:

- First of all, DSACK and Eifel can handle both spurious timeouts and spurious fast retransmits while F-RTO is only capable of fixing problems caused by spurious timeouts.

- As mentioned before, the root cause for spurious retransmissions is *retransmission ambiguity*. So, in order to detect such retransmissions, it is essential to eliminate the ambiguity at the very beginning. DSACK uses the DSACK block to inform the sender about a duplicate packet arriving at the receiver. When the ACK acknowledging the first arrival (at the receiver) of a packet arrives, DSACK cannot tell whether it is in response to the original packet or its retransmit. When the ACK triggered by the duplicate arrival arrives later, DSACK knows that the retransmission is spurious. But DSACK can never differentiate an original ACK from the ACK of the retransmit. Retransmission ambiguity is still there. F-RTO is a heuristic approach, which does not eliminate the ambiguity either. Among the three algorithms, only Eifel gets rid of retransmission ambiguity as it has extra information in every ACK just for the purpose.

- Since DSACK still suffers from retransmission ambiguity as we have just ex-

plained, it cannot avoid the unnecessary go-back-N retransmission. Eifel and F-RTO are able to have early detection upon the first one or two ACKs after a spurious timeout, so they can avoid the retransmission.

- In principle, the loss of the unnecessary retransmits during spurious retransmissions should also be taken as a congestion signal. When a spurious retransmit arrives at the receiver, as either Eifel or F-RTO does not change the receiver, the receiver just generates a DUPACK for the duplicate packet. So the sender has no way of knowing exactly which packet has triggered this DUPACK, and it can hardly detect the loss of the packet. As a DSACK block is generated for every duplicate packet, DSACK is able to explicitly inform the sender about which packet has arrived more than once. However, DSACK's detection would again be affected by the loss of the ACK with the DSACK block.

- F-RTO is only a heuristic approach. As we have mentioned in the previous section, it may not detect a spurious timeout in certain cases. DSACK or Eifel has no such concern.

- DSACK is vulnerable to ACK losses as each duplicate packet is reported only once. Eifel is quite robust because it has a window of ACKs for detection. Likewise, F-RTO is also more robust than DSACK. However, because F-RTO needs two ACKs for detection, it is slightly more prone to ACK losses than Eifel.

- As SACK blocks are appended to TCP headers only when necessary, DSACK introduces less overheads than the persistently-used timestamps in Eifel. F-RTO has no such overhead.

- In response to a spurious timeout, DSACK can do nothing in adapting the TCP retransmission timer; F-RTO has some effect on the timer as it can take RTT samples in the presence of *spurious* timeouts; Eifel can sample every acknowledged packet as it uses timestamps. This allows it the flexibility of more choices [23] in timer adaption.

- In response to spurious fast retransmits, Eifel and DSACK can take same approach in adjusting *dupthresh*. Some possibilities have been discussed in [7].

- The way to restore the congestion control state is independent of how the detection is made, so the three algorithms can take the same means in restoration. The most intuitive way is full restoration by setting the values of *cwnd* and *ssthresh* back to their values before the retransmission. Another way that has been found to be efficient is to set both variables to the previous *cwnd* value. Then the sender directly gets into congestion avoidance rather than the aggressive slow start. Some other possibilities have also been suggested in [23].

From the comparison above, we can see that each algorithm has its own advantages but also suffers from certain weaknesses. No one has definite superiority over the other two.

3.4 Multiple Packet Losses

All three algorithms discussed above focus solely spurious retransmissions. They do not fix other wireless-related problems, such as multiple packet losses due to link variations or handovers.

[13] proposed a network-based solution, called *ACK regulator*, which can help avoid multiple packet losses in TCP. It achieves this by implementing a regulator at the RNC in the 3G network, at the layer just above RLC, which needs to maintain a per-TCP-flow queue in order to regulate the flow of ACKs in uplink back to the TCP sender. By regulating ACK flows, it can reduce the burstiness of ACK arrivals at the sender so that they will not trigger multiple packet losses at the bottleneck any more. However, as the regulator maintains a queue for ACKs and controls the transmission of ACKs in the RNC amid the TCP connection between the mobile node and the remote host in a wired network, it breaks the end-to-end semantics of TCP connection. ACKs could be kept in the queue for an amount of time. As TCP depends on the ACK feedback for data transmission and flow/congestion control, this artificial delay of ACKs may impair

TCP performance in certain cases. For example, if the acknowledgments are DUPACKs indicating a fast retransmit, the delay can cause the TCP retransmission timer to expire before the DUPACKs arrive.

To fix the packet burst in response to an ACK burst, TCP can have a threshold to limit the maximum number of packets sent in a burst. Moreover, the delay or bandwidth variation which tiggers bursty ACKs ususally also triggers a spurious timeout at the sender waiting for the ACKs. Upon receiving the ACKs, with any of the detection algorithms presented in the previous section, the sender would detect the spurious timeout and restore its congestion control state. By using a non-aggressive restoration, the sender can also avoid packet burstiness. In our opinion, it is better to let TCP handle the problem rather than to introduce some extra agent in-between which may negatively affect the TCP end-to-end semantics.

At the time bursty ACKs arrive, the wireless link mostly has changed from a good to a poor condition. In such a case, packet loss would eventully happen as the current network capacity could not support the sender's previously-probed transmission speed any more. So instead of avoiding multiple packet losses, it is more meaningful to improve TCP's ability to recover from such packet losses efficiently.

Changing from a fast to a slow cell normally is handled well by TCP due to the self-clocking property. However, a sudden increase in RTT in this case can cause a spurious TCP timeout. In addition, a large TCP window used in the fast cell can create congestion resulting in overbuffering in the slow celling.

3.5 Summary

In this chapter, we have first provided a brief introduction of TCP concepts and algorithms. We have looked at the characteristics of wireless links and discussed their impacts on TCP performance. We have also done a more detail study of one major issue – spurious retransmissions, and also examined the existing works for solving problems caused by spurious retransmissions. Lastly, we have discussed another problem – multiple packet losses, and considered related works on the problem.

In the next chapter, we will propose a new approach for handling spurious retransmissions by improving on the existing approaches. Our proposal also enables TCP to have an RTO estimation in the presence of retransmits and a more efficient recovery from multiple packet losses.

Chapter 4

Eifel-I: the Improved Eifel Algorithm

In Chapter 3, we have looked at the existing approachs for handling spurious timeouts, which are frequently seen over wireless links. While they can avoid the problems caused by spurious timeouts to some extent, they also suffer from their own weaknesses. Moreover, none of them makes any attempt to solve another wireless-related problem: multiple packet losses, which usually co-occurs with spurious timeouts.

In this chapter, we propose a new approach for handling spurious retransmisssions by modifying the Eifel algorithm. With the modification, our proposal can also improve TCP's RTO estimation in the presence of retransmits and enhance TCP's capability in recovering from multiple packet losses.

In the following sections, we first look at the problems of the current Eifel algorithm. Then we describe our basic proposal and other related enhancement to TCP in detail.

4.1 Limitation of the Timestamp Option

Among the three algorithms we discussed in Section 3.3.2, only Eifel eliminates the retransmission ambiguity problem, and effectively detects spurious retransmissions. However, as its operation requires the use of timestamps, it works rather inefficiently. In [45], experiment results show that F-RTO can achieve slightly more improvement than Eifel. Here, we will look the problematic issues of Eifel introduced by the use of timestamps:

First, the TCP Timestamp option and its padding in the TCP header increase the size

of each TCP packet by 12 bytes [29]. For a Maximum Transmission Unit (MTU) [46] of 1500 bytes, the overhead is 0.8%. However, for a small MTU of 296 bytes, the overhead is actually 4%. In addition, not all packets carry the maximum payload, which results in an even larger relative overhead. For example, in the acknowledgment path, since the main traffic is along downlink, the uplink traffic mostly consists of pure ACKs. A pure acknowledgment packet, without any data or options and consisting of only the TCP and IP headers is 40 bytes. Then, including a 12-byte timestamp in every ACK would introduce a 30% overhead. This is a very high percentage. Given that sudden delays are often a problem on wireless links with low bandwidth, the increase in the TCP header overhead by including timestamps would make the communication less efficient. In the other way, as the MTU for a connection is fixed, the presence of timestamps also leads to more transmissions as each packet can now hold less data. Note that avoiding unnecessary transmissions is often more important than improving throughput over slow wireless links due to terminal energy and bandwidth savings.

Second, using the Timestamp option also reduces the available space for other TCP options, mainly the SACK option [35], since there are only 40 bytes for the TCP options field. For example, in the presence of timestamps, the number of SACK blocks that can be held by each TCP packet is reduced by 1. Without the information in the last missing SACK block, the sender may unnecessarily retransmit a series of packets. As minimizing unnecessary traffic is important for wireless links, it is desirable to avoid such unnecessary retransmits. If a more efficient SACK representation is used, as has been suggested in [43], the space occupied by the timestamp may not allow more SACK blocks. This can result in even more wasteful retransmits.

Lastly, current TCP/IP header compression schemes are limited in their handling of TCP options. Both schemes transmit only the difference from the previous header in order to reduce the large header overhead. For RFC1144 [28], a change in the options effectively disables TCP/IP header compression altogether and such header is always sent uncompressed. For RFC2507 [14], any change in the options renders the entire field uncompressible (leaving the TCP/IP header itself compressible, however). This is

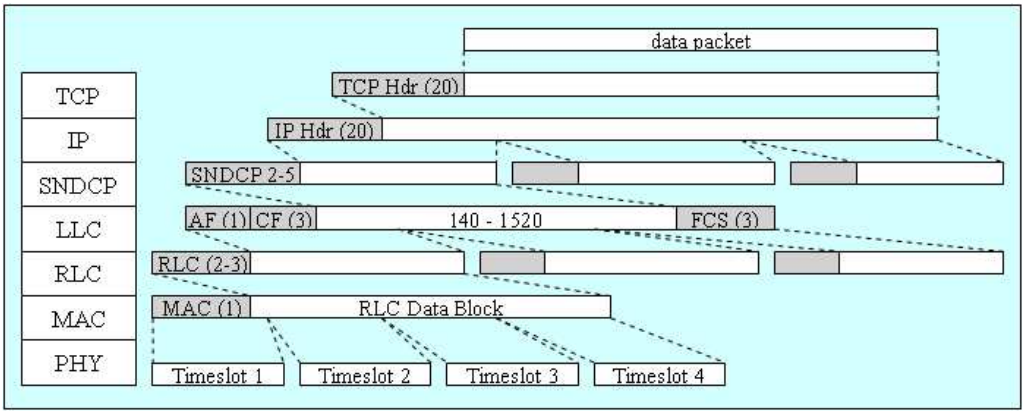


Figure 4.1: Packet-framing in GPRS protocol stack

the case when using the Timestamp option. This option is used in both the data and the ACK path for every packet, and its value typically changes from one packet to the next. The compression schemes can compress the header into just a few (3-5) bytes, and they have been shown to have significant improvement over slow links [49]. So the overhead caused by disabling header compression may overcome the improvement that TCP can gain by using timestamps for avoiding spurious timeouts. Even though the Robust Header Compression (ROHC) working group is developing new specifications to remedy this problem, those mechanisms are not yet fully developed nor deployed, and may not be generally justifiable.

TCP/IP Header Overhead

The overhead introduced because of the use of timestamps is closely related to the TCP/IP header. We deem that it is necessary for us to examine the relation of the TCP/IP header overhead to the overall protocol overhead in wireless links in more detail.

In Section 2.1.3, we have presented the GPRS protocol stack. Regarding the part over the radio interface between an MS and the BSS, the packet encapsulation process from TCP down to RLC/MAC is shown in Fig. 4.1 [47]. When a data packet is passed down through the protocol stack, each layer will prepend its own header (and sometimes also add a trailer), consisting of some control or sequencing information.

For a small data packet with no segmentation at any intermediate layer, without TCP/IP header compression, the total protocol overhead will be between 52 and 56

Coding Scheme	RLC data block (bytes)	Number of RLC blocks generated from an 1510-byte LLC frame
CS-1	22	72
CS-2	32 7/8	50
CS-3	38 3/8	41
CS-4	52 7/8	30

Table 4.1: RLC data block size for the four GPRS coding schemes. No MAC header is included here.

bytes; with TCP/IP head compression [1], the overhead can be reduced to between 14 and 20 bytes (approximately 70% reduction). The maximum LLC payload size is 1520 bytes (see Section 2.1.3). The RLC data block also has its size limit depending on the coding scheme used. We show the four coding schemes with their corresponding RLC block size in Table. 4.1. So for a data packet to be small enough that no segmentation would be triggered at any layer, its encapsulated unit down to RLC should be no more than the upper limit of RLC blocks. Such a packet can only be a pure TCP acknowledgment, which can possibly fit an RLC block of CS-4. In this case, employing the TCP/IP header compression can result in significant overhead saving.

If the data packet delivered to TCP is of a larger size, segmentation may occur at IP, SNDCP, and/or RLC/MAC. [47] illustrates that as the data packet on top of TCP increases and more LLC and RLC/MAC header overheads are introduced, the difference between the two cases, with and without TCP/IP header compression diminishes. However, we think that the assumption of the data packet increasing freely to any large size is unrealistic. Referring to [46], TCP always tries to avoid fragmentation and it is nearly impossible for an application to force TCP to send packets large enough to require IP fragmentation. In the Internet, most hosts stay within an Ethernet-based network, which has an MTU of 1500 bytes. Minus the 40-byte TCP/IP header, a larger data packet is usually limited to be no more than 1460 bytes. Prepended with the SNDCP header, the size of the data payload received by LLC roughly equals to its default payload size, 1503 bytes, and with LLC header and trailer, the resulting LLC frame is 1510 bytes. We list the number of RLC blocks resulting from this frame, depending on the coding scheme used, also in Table. 4.1. So the total protocol overhead can ranges from 110 to 270 bytes.

[1] Suppose the size of a compressed TCP/IP header is 4 bytes.

Using a compressed TCP/IP header, we can still get a 10 to 30% reduction in protocol overhead. In a full size TCP/IP packet of 1500 bytes, the saving of using a compressed header is nearly 3%.

The TCP Timestamp option was originally defined in RFC1323 [29]. It is used in conjunction with window scaling to prevent wrapping of the TCP sequence number space on very high speed links. It is also used to improve TCP RTT estimation by providing unambiguous round-trip timing for each acknowledged packet. However, sampling every packet for RTT estimation does not have much of an impact on RTO calculation [3]. The bandwidth of wireless networks currently ranges from tens of kbps to a few Mbps. For our discussion, we will take 10Mbps (1.25Mbps) as a maximum. Since the TCP sequence number is 32 bits, the time needed for the wrap-around of the sequence number space is: $2^{31}/(1.25M) = 1718seconds = 29minutes$. This duration is much larger than the two-minute Maximum Segment Lifetime (MSL) assumed by the TCP specification [41], so there is no worry about sequence number wrapping over wireless links. As suggested in RFC1323: "For low-speed networks, it might be a performance optimization to NOT use these mechanisms." A study by Mark Allman [2] on the deployment of TCP options also noted that the percentage of hosts in the Internet connecting to a web server during a one-and-a-half-year period using timestamps is only around 1% to 1.5%.

We therefore conclude: although the Timestamp option is already a proposed standard, it is in fact not widely deployed in the Internet. In high-speed networks with a bandwidth of up to a number of gigabits per second, we may see more use of timestamps. However, in low-speed wireless environments, its originally proposed functionalities, such as Round-Trip Time Measurement (RTTM) and Protect Against Wrapped Sequence Numbers (PAWS) [29] are of no use. So timestamps are not a good choice in Eifel for eliminating retransmission ambiguity.

The basic Eifel algorithm requires some kind of *extra information* to be included in TCP headers for eliminating retransmission ambiguity. Timestamps fulfill most require-

ments of being the extra information. Its major drawback is the overhead caused by the persistent use of timestamps. To avoid the overhead presented above while still keeping the ability in timing retransmits, we propose using timestamps only for retransmits and their corresponding ACKs. In the following sections, we look at how this alternative approach works in detail, and compare it with Eifel and other related approaches. We will refer to our proposed approach as Eifel-I during the discussion.

4.2 Selective Use of Timestamps in Eifel-I

To eliminate retransmission ambiguity, we still make use of timestamps as the piece of extra information. In the original specification of the Timestamp option [29], once a connection is initialized to use timestamps, it requires the TCP sender to place a timestamp in every outgoing packet and the receiver to echo a timestamp back in every ACK throughout the whole life of the connection. As discussed above, this persistent use of timestamps in all packets and ACKs can impair TCP performance over wireless links. To avoid the problems just discussed, Eifel-I modifies the option to allow selective use of timestamps. This "use-on-demand" idea is enlightened by the usage pattern of SACK blocks in the TCP SACK option [35].

In Eifel-I, most of the time, the TCP sender does not include any timestamp in the header of each outgoing packet. The sender only adds timestamps into retransmitted packets. On the other side, the receiver only echoes a timestamp in the ACK in response to an incoming packet with a timestamp in its header. Note that here we only focus on the modified use of timestamps in one direction. Since the option is used independently in each direction, it can be similarly applied to the bi-directional case.

This selective use of timestamps requires changes in both the TCP sender and receiver. The modification at the sender is straightforward: the sender only needs to take note whether the next outgoing packet is an original packet or a retransmit. If the packet is a retransmit, the sender takes the current value of the timestamp clock and writes it into the header of the packet. Similarly, at the receiver, only a few changes are required: the receiver needs to take care of *which timestamp to echo* when incoming packets are

not in their sending sequence or when Delayed Acknowledgment [46] is enabled [29]. The receiver's behavior can be fully illustrated by the following three cases:

An original transmit: When the original transmit of a packet arrives, the receiver can just behave as if the Timestamp option is not enabled. Since the incoming packet has no timestamp in its header, the receiver does not need to echo any value either.

A genuine retransmit: When a packet which first arrives at the receiver has a timestamp, it is supposed to be a genuine retransmit. The receiver reacts as if the Timestamp option is enabled according to its original specification in RFC1323.

A spurious retransmit: If the spurious retransmit of a packet successfully arrives at the receiver after the original transmit, it is a duplicate of the original packet that has already arrived. So the receiver generates a DUPACK which echoes the timestamp contained in this retransmit. As mentioned earlier in Section 3.1, a TCP receiver sends out a DUPACK immediately after the receipt of a duplicate packet, so there is no worry about delayed acknowledgment. A very rare case is that the spurious retransmit of a packet arrives before the original transmit due to severe reordering in the network. In this case, the receiver would take the retransmit as a genuine one, and when the original transmit arrives later, the receiver just generates a DUPACK for it without any timestamp.

Retransmission ambiguities can then be easily removed. To detect spurious retransmissions, the TCP sender only needs to check for any timestamp in the first acceptable ACK during loss recovery. If a timestamp is present in the header of this ACK, then the ACK is triggered by a retransmit, and the retransmission is not spurious; otherwise, the ACK is triggered by an original transmit, and a spurious retransmit has been sent.

The comparison of Eifel-I with other approaches with respect to their capabilities in handling spurious retransmissions is presented in Table. 4.2. As Eifel-I follows the basic algorithm of Eifel, it eliminates retransmission ambiguity and is able to handle both spurious timeouts and spurious fast retransmits. It can detect spurious retransmissions early enough to avoid the go-back-N retransmissions. As there is a window of ACKs

	Eifel-I	DSACK	Eifel	F-RTO
Handle both spurious timeouts and spurious fast retransmits	Yes	Yes	Yes	
Eliminate retransmission ambiguity	Yes		Yes	
Avoid the go-back-N retransmission	Yes		Yes	Yes
Robust against ACK losses	Yes		Yes	Yes
Always make a correct detection	Yes	Yes	Yes	
Can detect the loss of spurious retransmits	Yes	Yes		
No overhead	Yes	Yes		Yes
More space for other TCP options	Yes			Yes
Use of TCP/IP header compression	Yes	Yes		Yes
Measure RTTs of retransmits	Yes		Yes	

Table 4.2: Comparison of Eifel-I and other approaches

for original packets containing no timestamps, Eifel-I is very robust against ACK losses. During a spurious retransmission, only a single unnecessary retransmit would be sent out. Other packets before or after it are all original ones with no timestamps. As this retransmit arrives as a duplicate packet, an immediate timestamped ACK will be generated. So the arrival of a timestamped ACK later indicates the successful transmission of the retransmit while the non-arrival of this ACK until the ACKs of its following window of packets have arrived signals the loss of the packet or its ACK. Since retransmits form a relatively small part of the total transmitted packets of a connection, the timestamp overhead introduced by these packets can be considered negligible. Avoiding timestamps also saves more TCP option space for other options. Moreover, since the TCP options field is rarely changed by timestamps in retransmits, the TCP/IP header compression schemes [28] [14] can be used with Eifel-I. Lastly, the use of timestamps in retransmits also allows TCP to collect RTT samples from retransmits, which was prohibited by common TCP implementations. In summary, Eifel-I captures all the advantages of the other approaches.

In certain cases, Eifel-I is more accurate in detection than Eifel. As with the corner case of timeout due to loss of all ACKs discussed in [33], if all ACKs for the window of original transmits are lost while the oldest outstanding packet arrived at the receiver, the retransmit arrives as a duplicate. Although all the data packets have correctly arrived,

the sender should take this as a genuine timeout as there have been quite a number of losses along the ACK path. In response to duplicate packets, RFC1323 requires that the timestamp of the last packet that arrives in-order should be echoed. Then the timestamp carried by the ACK for the retransmit, which is also the first acceptable ACK that arrives at the sender during loss recovery, is commonly smaller than the timestamp carried by the retransmit. Consequently, Eifel misinterprets such a timeout as being spurious. Eifel can only detect that the ACK is triggered by the retransmit, if either the receiver echoes the timestamp of the duplicate packet as in some TCP implementation that does not follow RFC1323 strictly, or the DSACK enhancement [18] is enabled in the connection. However, Eifel-I can easily figure out that the ACK is triggered by the retransmit, since this ACK contains a timestamp. Essentially, Eifel-I is accurate in its detection because it uses the existence of timestamps for detection while Eifel on the basis of comparing timestamp values. In fact, it is this property of Eifel-I that enables a TCP sender to detect the loss of the unnecessary retransmit after a spurious retransmission.

A drawback of Eifel-I is that it needs a few changes at both the TCP sender and receiver. However we think that as long as the advantages gained with the use of Eifel-I are significant enough to justify its usability, the small drawback is tolerable.

4.2.1 Negotiating the Use of Eifel-I

In Section 3.1.1, we have briefly described the three-way handshake for TCP connection establishment. The use of the original TCP Timestamp option is also negotiated during the handshake: the initiating TCP end may send the initial SYN packet with a timestamp; at the other end, if the receiving TCP receives the timestamped initial SYN, it can include timestamps in all its subsequent outgoing packets.

The selective use of timestamps could be similiarly negotiated with slight modification. Here we present one possible negotiation process.

Eifel-I is aimed at improving TCP performance over wireless links. As the TCP on an MS knows that it is behind a wireless network, we can let it initiate the use of selective timestamp, i.e., Eifel-I. In current wireless networks, the common scenario is

that a wireless link is used as a last-hop link to a host and the host deploys the wireless link to connect to the rest of the Internet. The main traffic flow is from a fixed host in the Internet to the mobile host, where the mobile requires data from the fixed host, such as web browsing, FTP, etc.

When the TCP on the MS initiates the connection, it can send the initial SYN packet with a special timestamp value such as 0. If the other host is Eifel-I enabled, it will know that the initiating end is requesting the use of Eifel-I. Then in its SYN and ACK packet, it acknowledges that special value and also includes a special value as its own timestamp. After the SYN and ACK packet arrives at the MS, TCPs at both hosts can use Eifel-I now. If the receiving host is not Eifel-I enabled, it will send a normal timestamp as its own and the TCP on the MS will abort the use of Eifel-I.

We have described one method for negotiating the use of selective timestamps or Eifel-I. Others are also possible for catering to other network scenarios, such as where neither of two end hosts is a mobile one but the connection between them involves some wireless links.

We have so far explained the detection mechanism of Eifel-I. In the next section, we will discuss the response mechanisms that work with Eifel-I's detection mechanism, including an improvement on TCP RTT estimation.

4.3 Responses to Spurious Retransmissions

When a spurious retransmission is detected, a response algorithm generally needs to perform three tasks in recovery. First, if it is a spurious timeout and the detection is made during the go-back-N retransmission, the TCP sender stops the unnecessary retransmission and continues from the next unsent packet. Second, it restores the congestion control state from values saved before the retransmission. Lastly, the sender can try to avoid future spurious retransmissions by adapting either the retransmission timer for a

spurious timeout or *dupthresh* for a spurious fast retransmit.

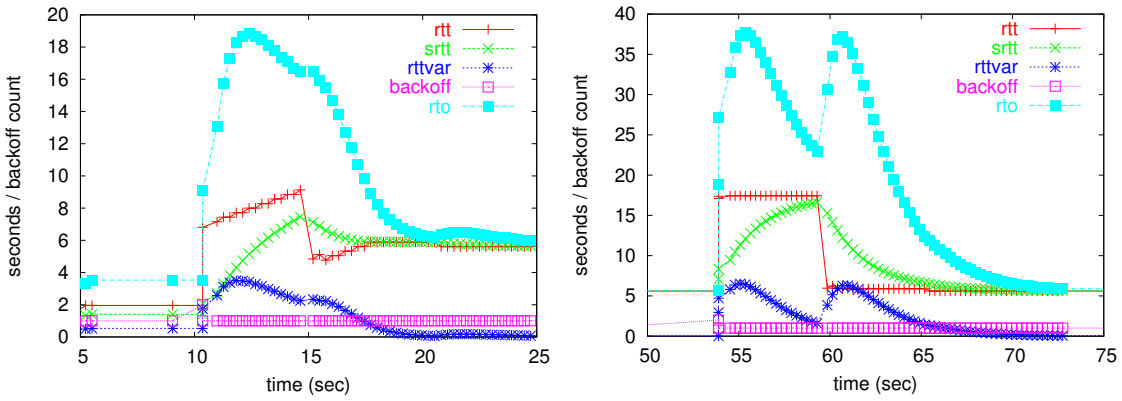
The first task is straightforward. As we have mentioned in Section 3.3, the second task is independent of how the detection is made. So Eifel-I can also take any of the restoration methods [23] that have been proposed. In Chapter 6, we will discuss further the various restoration methods based on our experiment results. How the value of *dupthresh* is adjusted after a spurious fast retransmit is not affected by Eifel-I's detection method, either. Therefore, in our discussion of the response mechanism, we will focus on the adaptation of the TCP retransmission timer.

4.3.1 Adapting the TCP Retransmission Timer

In Section 3.1.2, we have explained how TCP computes each RTO value. RTO is computed from RTT estimations in the current connection, where the RTT estimator is gradually changed with each incoming RTT sample. However, [3] has pointed out that the timer adapts fairly slowly to changes in network conditions. The TCP sender takes one RTT measurement at a time, so SRTT and RTTVAR are updated by new RTT samples only once per RTT. But according to Karn's algorithm [30], RTT samples must not be made using retransmitted packets because of the retransmission ambiguity problem. So any time a retransmission occurs, without the Timestamp option, the sender cannot safely measure RTTs for those retransmits, and thus will take a long time to adapt its RTT estimates in order to improve its broken RTO estimation.

With timestamps, Eifel removes retransmission ambiguity. Since each acknowledged packet can be used for RTT sampling now, it is able to solve the slow adaptation problem. However, the definition of the RFC2988 timer has been based on the premise of one RTT measurement at a time, and the estimation gains and variation weight used in calculations have been tuned to this particular case. Then, when RTT samples are being taken at a much higher rate – one per packet – these parameters fail to perform well. Fig. 4.2(a) and 4.2(b) show how the RTO dynamics change with Eifel when a delay spike occurs during either the slow start phase or the congestion avoidance phase.

- In Fig. 4.2(a), during slow start, the values of RTT samples are small because



(a) During slow start

(b) During congestion avoidance

Figure 4.2: RTO Dynamics when a delay spike occurs – Eifel

the packets transmitted incur less queuing time compared with those transmitted during congestion avoidance. A delay spike occurs from the 5th to the 10th second. After the 10th second, the TCP sender starts to get RTT samples from the ACKs of delayed packets. As we can see, the value of RTT suddenly increases. At the meantime, SRTT gradually increases because of the smoothing function presented at the beginning of this subsection. RTTVAR first increases, but then decreases (down to 0 from the 19th second) as SRTT approaches RTT. As the value of RTTVAR is amplified by four times when computing RTO, its change dominates the change of RTO. So RTO first increases to a considerably large value, and then decreases to the level of the current RTT value.

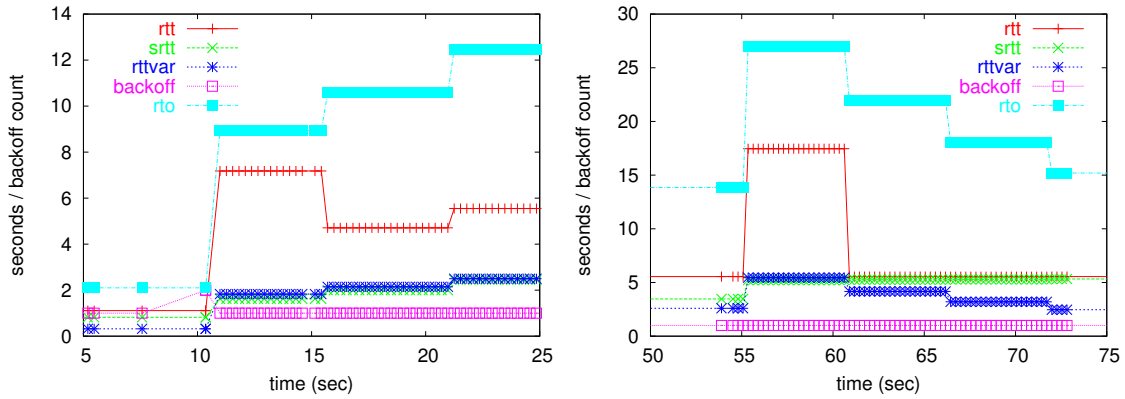
- In Fig. 4.2(b), during congestion avoidance, the values of RTT, SRTT, RTTVAR and RTO have similar changing patterns as their counterparts during slow start. However, after the increase just following the delay spike, the value of RTO shows a second sudden increase. During slow start, after the spurious timeout, the TCP sender will restore its transmission rate and keep on sending packets at an exponential rate. As there will be more packets being injected into the network, the queuing time for packets increases and RTT will increase quickly to a level comparable to the previous delay. So the change in RTTVAR is small (about 3 seconds at the 16th second in Fig. 4.2(a)) when the ACKs of the new packets sent after

the timeout start to arrive at the sender. However, during congestion avoidance, RTT is quite stable. So after the window of delayed packets, RTT will return to its usual level (after the 60th second). As the value of SRTT has already increased to the level of delayed RTTs, this drop-back in RTT will introduce a sudden increase in RTTVAR (6 seconds at the 61st second in Fig. 4.2(b)). Consequently, it leads to a second large increase in RTO.

During congestion avoidance, TCP with Eifel will always have such two consecutive increases after a spurious timeout. It may also have a similar double-increase during slow start if the delay spike is much larger than the RTTs from the new packets sent after the spurious timeout. The reason that there is only one increase in RTO in Fig. 4.2(a) is that the delay is not very large. In both cases, after the one or two increases, RTO quickly decreases to the level of RTTs. The reason is that the values of estimator gains are too high for the one-per-packet sampling rate. This causes SRTT and RTTVAR to decay very quickly. Thus, RTO eventually collapses into RTT, and then it cannot maintain at a reasonable level for avoiding future delay spikes. So Eifel performs poorly in avoiding future spurious timeouts, as shown in Fig. 4.4.

Ludwig and Sklower [34] have proposed a new TCP retransmission timer which makes the values of estimation gains and variation weight being dynamically adjusted by the rate at which the RTT is sampled. It is supposed to work better with the one-sample-per-packet case. However, this timer is not yet fully developed nor evaluated. It has also been shown that there is little help by taking more samples using the Timestamp option in RTO estimation [3]. As the RFC2988 timer is the current de facto standard and is widely deployed, we choose to improve it rather than to develop a totally new and uncertain one.

In conjunction with Eifel-I detection, we propose a simple yet effective modification to the RFC2988 timer for quick adaptation to changing network conditions, more specifically, the highly variable delay spikes. The modified timer is still based on one measurement per RTT. It ensures that the RTT estimator is always able to get an RTT sample, even with retransmits. In the absence of retransmissions, the RTT samples can



(a) During slow start

(b) During congestion avoidance

Figure 4.3: RTO dynamics when a delay spike occurs – Eifel-I

be taken as normal. In the presence of retransmissions, in case of a spurious retransmission, as retransmission ambiguity has been removed, the TCP sender can be sure that the ACK for the packet, whose RTT is being sampled, is triggered by the original transmit and then this RTT sample can be safely taken. This is similar to F-RTO's side-effect on the TCP retransmission timer in Section 2.3. In case of a genuine retransmission, since the ACK for the retransmit contains the echoed timestamp, the RTT sample can be computed using this timestamp.

The changes in RTO dynamics with Eifel-I's modified timer are shown in Fig. 4.3. During slow start in Fig. 4.3(a), similar to Fig. 4.2(a), after the spurious timeout during from 5th to the 10th second, the TCP sender is able to get the RTT value from the delayed packet that is being sampled. So its RTO can increase to a reasonably large value. Then with the sampled packets in the subsequent windows of packets, RTO increases with the increases of SRTT and RTTVAR. Since the values of SRTT and RTTVAR are updated once per window, RTO is also changing at this rate. This stable RTO value enables the sender to avoid future spurious timeouts. During congestion avoidance in Fig. 4.3(b), although the delay spike occurs, there is no timeout. This can be seen from the values of the backoff counter. If a timeout has ever happened, the counter would be set to 2. However, in Fig. 4.3(b), it remains at 1 all the time. The timeout is avoided because of the timer adaptation done earlier after the spurious timeout in Fig. 4.3(a). After taking the delayed sample, RTT drops back to its usual level (after the 61st second). However,

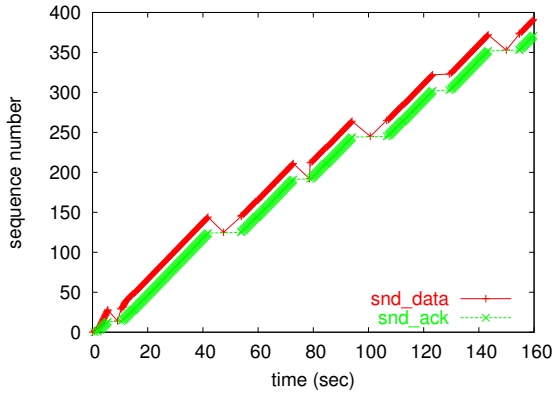


Figure 4.4: In the presence of delays – Eifel

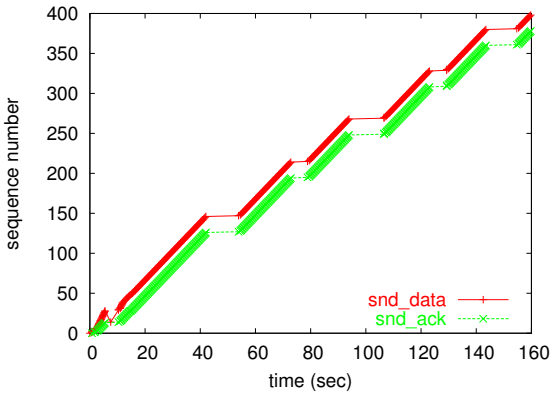


Figure 4.5: In the presence of delays – Eifel-I

since RTTVAR decreases once per window, RTO also decreases at the same rate. So RTO can remain at a reasonably high level for avoiding the next delay spike. Also note that Eifel-I's timer adaptation is less aggressive than Eifel's. After getting the delayed samples, the RTO value adjusted by Eifel reaches nearly 40 seconds (at the 55th and the 61st second in Fig. 4.2(a)), while Eifel-I's adapted RTO value is about 27 seconds (from the 55th to the 60th second in Fig. 4.3(a)). Getting an RTO value that is too large may lead TCP with Eifel to suffer from a long wait before timeout recovery when real packet losses occur.

With our proposed modification, the new retransmission timer can take samples from retransmits and make in-time adaptation when delay spikes occur. At the same time, as we keep the premise of one sample per window, the parameters used in the original RFC2988 timer still work well. The adapted SRTT and RTTVAR will not decay abnormally fast after spurious timeouts, so the new timer can have better performance in avoiding future spurious timeouts. In Fig. 4.4, a TCP with Eifel detects spurious timeouts caused by delay spikes and avoids the go-back-N retransmissions; in Fig. 4.5, after detecting the first spurious timeout, a TCP with Eifel-I can avoid the other timeouts later because of its superior timer adaptation. In fact, the figures in Fig. 4.2 are generated from the same set of simulation traces as Fig. 4.4, and the two delay spikes shown in Fig. 4.2(a) and 4.2(b) correspond to the first two in Fig. 4.4. So do Fig. 4.3 and 4.5.

One may say that the retransmission timer adaptation by Eifel-I is still slower than Eifel's, since a TCP sender with Eifel can adjust its RTO-related variables when it gets

the ACK for the first packet in the delayed window while a sender with Eifel-I can only do that when the ACK for the packet in the delayed window, which is currently being sampled, arrives. However, since a TCP sender usually sends out up to a window of packets at one time, getting the information about what network condition this window has encountered from just one of its packets should be enough. Moreover, keeping the sampling rate of RTT and the updating rate of RTO the same as the rate at which the TCP sender injects each window of packets is also more rational than at the rate of once per packet. Preliminary experiment results in Fig. 4.4 and 4.5 also show the strength of our modification in avoiding future timeouts. So we take this reasonable adaptation speed of our modification as an advantage over Eifel, rather than a weakness. In Chapter 5, we will show more experiment results in verifying the capability of our improvement in the retransmission timer.

To further adapt the conservativeness of the TCP retransmission timer, the TCP sender may also use mechanisms like resetting the retransmission timer once getting the first delayed RTT sample after a spurious timeout, resetting the backoff counter only on a genuine timeout, etc. [23]. However, adapting a too conservative RTO value may also cause the sender to suffer excessively from waiting for genuine retransmission timeouts. In order to achieve acceptable performance in other networks, such as where packet losses may frequently occur, we should also prevent the retransmission timer from being too conservative. Here we do not pay attention to these mechanisms any more. In Chapter 6, we will discuss these mechanisms and the long timeout wait problem based on the simulation results.

4.4 Avoiding Multiple Fast Retransmits

As discussed in the previous sections, with the selective use of timestamps, the Eifel-I algorithm can easily distinguish between ACKs triggered by originally-transmitted packets from ACKs of retransmits (each of which contains a timestamp). This capability also enables TCP to solve another existing problem, especially for non-SACK TCPs (i.e., TCP without SACK, such as TCP Reno, Newreno, etc.).

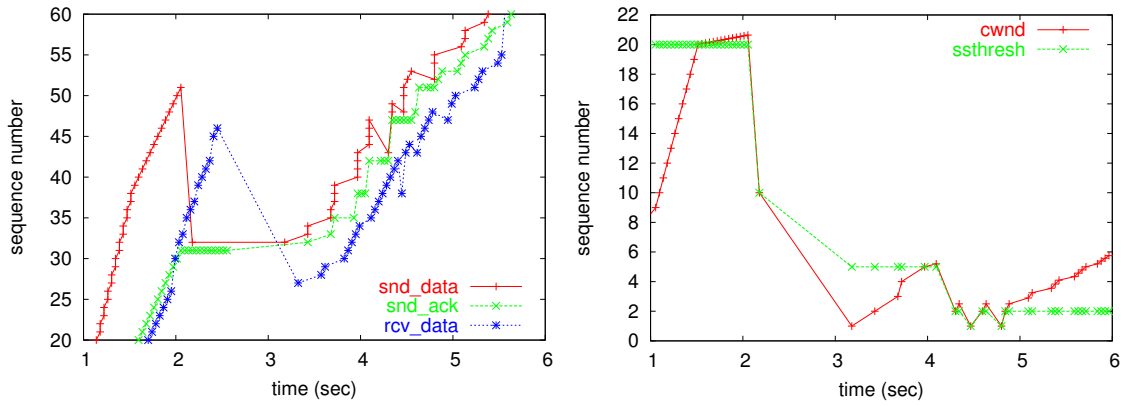
4.4.1 The Multiple Fast Retransmits Problem

Because of the cumulative nature of acknowledged sequence numbers in TCP acknowledgments, without selective acknowledgments or timestamps, a DUPACK cannot provide any information to identify the packet (or packets) at the receiver side that has triggered the DUPACK. Then, when it receives a DUPACK, a TCP sender is unable to decide whether the DUPACK results from a lost or reordered packet, or results from its own unnecessary retransmits that have already arrived at the receiver. (In Section 3.1, we have discussed TCP Acknowledgments in some detail.)

After a retransmission timeout, DUPACKs may arrive which are triggered by either unnecessary timeout retransmits or the loss of one or more retransmitted packets. Multiple packet losses in a single window of data would always cause a timeout for non-SACK TCPs. Since not all packets in the window have been dropped on the way, some packets retransmitted during slow-start would be unnecessary for the receiver and trigger DUPACKs. In such a case, if the sender always responds to three or more DUPACKs with a fast retransmit, the fast retransmit and fast recovery could be unnecessarily triggered and thus result in unnecessary reduction of the congestion window. As multiple packet losses usually trigger a fast retransmit before timeout, within the transmission of a single window of data, fast retransmit and fast recovery algorithms would be executed more than once [4]. This *multiple fast retransmits* problem was first identified for TCP Reno and Tahoe in [19]. One typical scenario may occur as follows (Fig. 4.6(a) [2]):

Multiple packet losses in a window first triggers a fast retransmit (after the 2nd second), and then the retransmission timer expires. The timeout starts the go-back-N retransmission where all the remaining unacknowledged packets in the window are resent (starting after the 3rd second). However, as we can see from *rcv_data*, some packets in the window have successfully arrived at the receiver, so the retransmits of these packets arrive as duplicates and each triggers a DUPACK. In the example, there are three such cases (at the 4.4th, 4.5th, and 4.8th second) where the sender unnecessarily retransmits three (or more) packets. Fig. 4.6(b) verifies that this indeed leads to multiple reductions

[2] rcv_data is shifted down by 5 units for illustration purpose.



(a) Time sequence

(b) Congestion control state

Figure 4.6: TCP Reno with unnecessary multiple fast retransmits

of the sender's transmission rate.

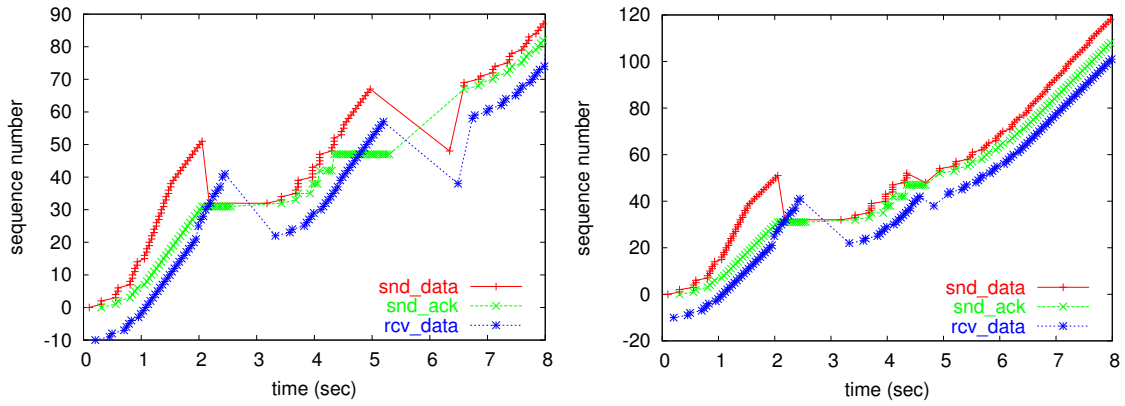
With NewReno [17], the TCP sender remains in fast recovery until the retransmission timer expires, or until all of the packets outstanding when fast retransmit started have been acknowledged. Thus with NewReno, the problem of multiple fast retransmits from a single window of data occurs only after a timeout, and the last two unnecessary fast retransmits in Fig. 4.6(a) may be avoided.

4.4.2 BugFix in TCP NewReno

To further resolve the problem, a modification of TCP called *bugfix* has been proposed with the introduction of TCP NewReno (RFC2582 [17]). When a retransmission timeout occurs, the TCP sender records the highest sequence number transmitted so far in a variable called "send_high" (and exit fast recovery if applicable). Then when a fast retransmit is to be triggered later, the sender checks to see if those DUPACKs triggering this fast retransmit cover *more* than "send_high":

- If not, as the sender has just retransmitted those packets no more than "send_high", it does not take the DUPACKs as an indication of a new congestion loss but an indication of unnecessary retransmits (as illustrated in Fig. 4.6(a)). Thus, the sender will not initiate the fast retransmit and fast recovery.

- Otherwise, it starts the fast retransmit as usual.



(a) With NewReno's bugfix

(b) With Eifel-I-based solution

Figure 4.7: TCP NewReno with a lost retransmitted-packet

This modification provides the avoidance of unnecessary multiple fast retransmits. It achieves this by blindly ignoring all the incoming DUPACKs after a retransmission timeout until retransmission recovery is completed. However, in case a retransmitted packet corresponding to a lost packet in the original window is itself lost, it would have been better for the sender to execute the fast retransmit. But for a non-SACK TCP sender which has implemented *bugfix*, it would not infer a packet loss from the series of DUPACKs, and as always, the retransmission timer is the backup mechanism for recovering the packet loss in this case. A sample scenario is presented in Fig. 4.7(a). The sender has to wait for a timeout retransmit in order to recover the previously lost retransmit (in this case it is packet 48). This would definitely hurt TCP performance. So, some explicit mechanisms may still be needed to resolve this problem.

4.4.3 The Eifel-I-based Solution

The fundamental cause of "multiple fast retransmits" is that a non-SACK TCP sender has no way of knowing which packet(s) triggers the DUPACK(s). *bugfix* in NewReno may avoid the problem, but at the cost of failing to detect a real retransmit loss if there is and suffering from lengthy wait for timeout retransmission (such as the scenario illustrated in the previous section). This is because it still fails to eliminate the ambiguity in DUPACKs.

A non-SACK TCP sender with Eifel-I enabled can differentiate the ACKs in response to retransmits from the ACKs of originally-transmitted packets by checking the existence of timestamps echoed in ACKs. As shown in Fig. 4.6(a) or 4.7(a), at the time the go-back-N retransmission starts (from the 3rd second), all ACKs for the original packets in the window (no more than "send_high") have already arrived at the sender (from the 2nd to 2.5th second). So all the ACKs (or DUPACKs) that arrive later should contain an echoed timestamp. As the transmission goes on, after finishing retransmitting all the unacknowledged and presumably lost packets in the window, the sender continues to send out new packets (roughly starts from the 4.5th second), each of which has a sequence number more than "send_high" and contains no timestamp. At the time, the sender is sure that there can only be the retransmits of the old window and the new packets above, and their corresponding ACKs flowing between the sender and receiver.

If the sender later receives a DUPACK acknowledging no more than "send_high" but without a timestamp, it knows that the DUPACK is triggered by a new packet which it has sent out only once and thus can be taken as a strong indication of loss of retransmitted packets in the network. Instead of waiting for a timeout retransmission, the sender can initiate a fast retransmit immediately. But if the DUPACK does contain a timestamp, then it should be triggered by a retransmit. The sender will then ignore it just as NewReno's bugfix does.

By comparing the plots in Fig. 4.7, we can see that the solution based on Eifel-I has two advantages:

- First, the TCP sender can recover the lost packet at an earlier time. In Fig. 4.7(b), the lost retransmit is of sequence number 48 and the old window (with multiple losses) covers packets of sequence number up to 51, so when the *non-timestamped* DUPACK triggered by packet 52 arrives at around the 4.5th second, the sender can start the fast retransmit immediately. The arrival of the *new* DUPACK should be within one round-trip time, while a retransmission timeout is usually several times of the average RTT. Thus if it were for a timeout retransmit, it would take a much longer time. In this scenario, the timeout occurs nearly two and a half seconds

later while the RTT of the DUPACK is no more than half a second.

- Second, the sender avoids the timeout and thus can keep transmission at a higher rate. When a retransmission timeout starts, the congestion window is first reduced to 2 and then incremented in slow-start up to half of its previous value, and after that it gets incremented in congestion avoidance; if it is a fast retransmit, the congestion window would have less reduction where it gets incremented directly from half of its previous value and in congestion avoidance.

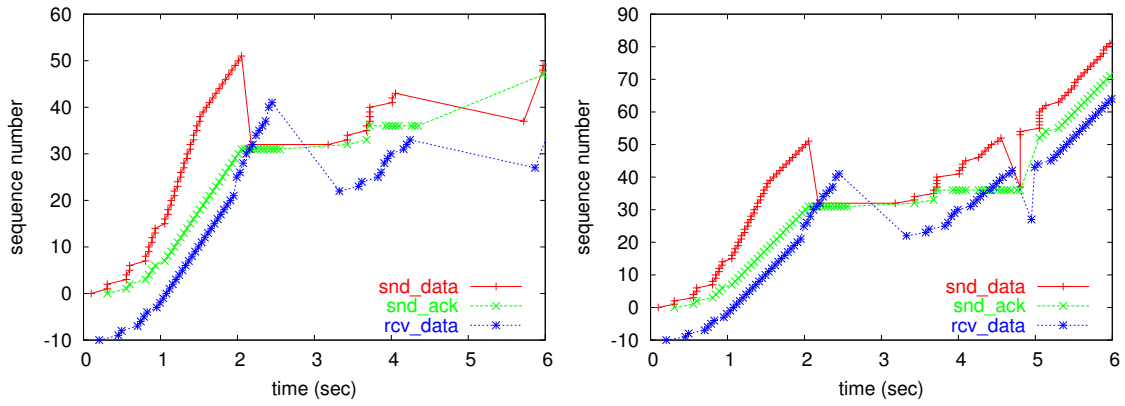
With both advantages, our proposed solution offers significant performance improvement. As presented in Fig. 4.7, by the end of the 8th second, it can help the TCP sender send about 35% more packets (118 packets) than NewReno's bugfix (87 packets).

Further Improvement

After a retransmission timeout caused by multiple packet losses, the Eifel-I-based solution can avoid unnecessary fast retransmits caused by duplicate retransmits, just as NewReno's bugfix. In addition, it handles recovery more efficiently if new loss indeed occurs in the retransmitted packets. We will explain in the following paragraphs how Eifel-I achieves more efficient loss recovery.

According to Section 5 in RFC2582 [17], if a fast retransmit is avoided by bugfix, the TCP sender should not invoke fast recovery upon subsequent DUPACKs either. Thus, if the DUPACKs arrive before any new packet has been sent out, the Eifel-I-based solution may not be able to detect the retransmit loss and recover it in time.

To improve this, we propose that even through the TCP sender avoid the fast retransmit upon receiving the third DUPACK acknowledging no more than "send_high", it should still follow Steps 3 & 4 in Section 3, RFC2581 [4]: for each additional DUPACK received, a TCP sender should increment its cwnd by one packet, and transmit a new packet if allowed by the new value of cwnd and the receiver's advertised window. This modification is consistent with the underlying concept for using fast recovery: since each DUPACK indicates that a packet has left the network, the sender should keep the transmission continuing and preserve the ACK clocking. Otherwise, the network resource



(a) The initial version

(b) The improved version

Figure 4.8: The Eifel-I-based approach's recovery upon a retransmit loss early in the window

would be underutilized.

Following this new-packets-on-DUPACKs, there are only two possibilities:

- A non-duplicate ACK arrives soon (possibly after some more DUPACKs) if the previous DUPACKs were triggered by duplicate retransmits. This ACK is in response to a retransmit that has filled a gap in the receiver's buffer. Since there is no loss at all, the network has enough bandwidth to support the sender's current transmission rate. So in response to each packet left out of the network, it is safe for the sender to inject one packet back.

- As the transmission continues upon incoming DUPACKs, the sender finishes sending all the retransmits and starts to transmit new packets (above the old window) into the networks. Later, a DUPACK caused by a new packet arrives, indicating that there is a real loss. Thus, a fast retransmit is invoked. As the fast retransmit will be initiated soon, there will not be many packets sent out before this retransmit (no more than the size of the old window).

Also note that this modification only changes TCP's behavior during retransmission recovery.

Referring to Fig. 4.8, with this further improvement, even if the loss happens early in the retransmitted window, as the sender continues transmitting upon subsequent DU-

PACKs, packets above "send_high" will be sent out and the lost packet will be recovered within one RTT.

The Eifel-I-based solution can therefore completely avoid the problems caused by DUPACK ambiguity after a timeout retransmission, no matter if the problem is unnecessary multiple fast retransmits, or a long timeout retransmission triggered by retransmit losses.

Related Works

While we were working on the Eifel-I-based approach outlined above for avoiding multiple fast retransmits, some other heuristics for solving the problem have been proposed [20]:

One heuristic (ACK heuristic) is based on the amount of advancement of the cumulative acknowledgment field. If a TCP sender has unnecessarily retransmitted at least three consecutive packets during a previous timeout, there would be a jump by at least four packets in the cumulative acknowledgment field. So each time the sender is to invoke a fast retransmit, besides other condition checkings, it need also verify that the difference between two consecutive acknownledgment numbers is at most three packets.

Another heuristic (timestamp heuristic) is based on the echoed timestamps when the Timestamp option [29] is enabled. A TCP sender needs to store the timestamp of the last ACK that has acknowledged one or more previously unacknowledged packets. When receiving the third DUPACK, the sender checks if the incoming echoed timestamp equals the stored timestamp. If so, the DUPACK indicates a lost packet.

Discussion and Comparison

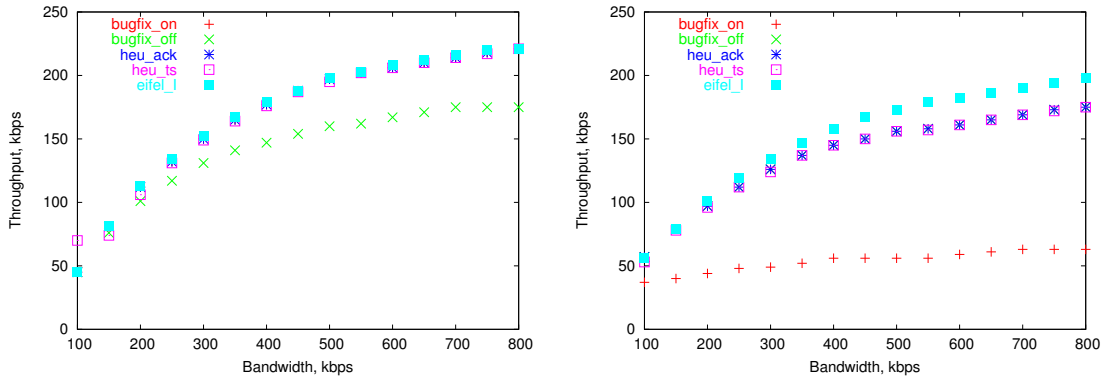
As noted in the same paper [20], both heuristics have certain limitions in application. For example, ACK heuristic is susceptible to ACK losses, as a small number of ACK losses is sufficient for a more-than-three-packet jump in the cumulative sequence number; timestamp heuristic fails if certain TCP implementations do not follow RFC1323 or its revision closely. Moreover, before applying either heuristic, a TCP sender should

check that the timeout is not spurious to avoid using ACKs generated in response to the original but not retransmitted packets.

We also noticed that timestamp heuristic needs the timestamp of the last acknowledged packet at the sender side. As the sender cannot know which would be the last acknowledged packet (that would be acknowledged by the receiver) in advance, it has to store the timestamps of all the unacknowledged packets. This requires extra storage space in the sender's TCP Control Block. If either the old congestion window is large or the sender is concurrently supporting numerous connections, the required space would be considerably large. Timestamp heuristic also introduces timestamp overhead in every packet.

The original Eifel-I algorithm, as described in Section 4.2, aims to cope with spurious timeouts. If a spurious timeout ever occurs, it will be handled by Eifel-I, and no further unnecessary retransmits except the first timeout retransmit will be sent out. Thus, only a real timeout can trigger the go-back-N retransmission. At the time new packets beyond "send_high" are sent, there should be no origingal ACKs for the old window left except for very rare and severe reorderings. As a double-check, a TCP sender implementing the Eifel-I-based approach needs to ensure that some new packets have really been sent out before it uses the non-timestamp DUPACK for detecting retransmit losses. Thus, whether the timeout is spurious is never a problem for the Eifel-I-based approach. Moreover, compared with the two heuristics, our approach introduces no overhead while being always able to make the correct decision.

In [20], the authors set up a three-state error model for evaluating ACK and timestamp heuristics by simulation. The initial state is loss-free. The second state is short and serves to trigger a retransmission timeout. In the loss scenario all packets are dropped in the second state. In the duplicate scenario the second state has a 50% loss rate and a fast retransmitted packet is dropped. The third state lasts two seconds and has a 10% loss rate. It causes a small number of retransmits to be dropped. The model is designed to capture the two scenarios discussed earlier in this section. As far as we know, there is no study showing whether such scenarios are common in the Internet. But they do happen,



(a) The "duplicate" scenario

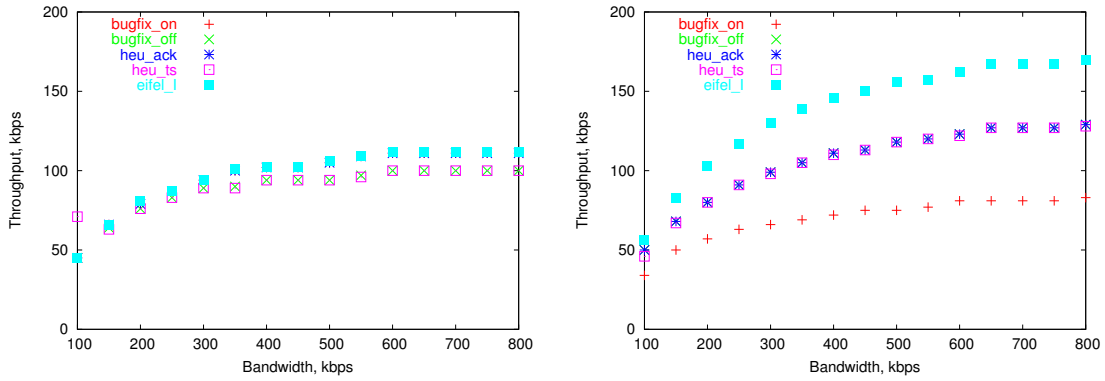
(b) The "loss" scenario

Figure 4.9: Comparison of different approaches for avoiding multiple fast retransmits

such as during a handover in wireless networks, and by using the two scenarios, we can have some idea of the possible improvement we may get by using a certain approach to explicitly fix the problems that follow a retransmission timeout. Here, we compare NewReno TCP using bugfix, without using bugfix, with ACK heuristic, with timestamp heuristic, and the Eifel-I-based approach.

In Fig. 4.9(a) & 4.9(b), the plots of TCP with/without bugfix, with ACK heuristic and with timestamp heuristic are the same as those in [20]. In the duplicate scenario, the performance of TCP with the Eifel-I-based approach is similar to that of TCP with either bugfix, ACK or timestamp heuristic. Without bugfix, TCP suffers from unnecessary fast retransmits because of duplicate timeout retransmits.

Fig. 4.9(b) shows the simulation results of the loss scenario. As expected, TCP with bugfix falls below the others in performance because it needs to wait for a timeout to retransmit the lost retransmit(s). TCP with the Eifel-I-based approach performs the best. It achieves more than 10% improvement over TCP with either heuritic or without bugfix and more than 230% improvement over TCP with bugfix. TCP with either heuristic determines the loss of a retransmit packet upon receiving the third DUPACK and invokes fast retransmit immediately. But TCP with the Eifel-I-based approach will wait until a DUPACK triggered by a new packet comes before initiating the fast retransmit. If the *new* DUPACK happens to be the third DUPACK, the three approaches produce the same result. But mostly the new DUPACK will either be before or after the third DUPACK.



(a) The "duplicate" scenario

(b) The "loss" scenario

Figure 4.10: Comparison of different approaches for avoiding multiple fast retransmits, – delayed acknowledgment is enabled

If it is the fourth or any subsequent DUPACK, TCP with the Eifel-I-based approach will keep transmitting and start the fast retransmit at a later time when the congestion window has increased; if it is the first or the second DUPACK, the lost retransmit can only be one of the last few packets in the retransmission window, and TCP with the Eifel-I-based approach can start the fast retransmit earlier by saving the time for waiting the second and/or the third DUPACK.

[20] only studied the scenarios when delayed acknowledgment is disabled. However, as delayed acknowledgment possibly introduces some complexity in differentiating unnecessary fast retransmits and real retransmit losses, we have experimented the same scenarios with delayed acknowledgment enabled. The plots in Fig. 4.10 presents our simulation results.

In the duplicate scenario (Fig. 4.10(a)), TCP with either bugfix, ACK heuristic or the Eifel-I-based approach performs slightly better than TCP without bugfix or with timestamp heuristic. This is because delayed acknowledgment reduces the number of ACKs, and therefore the number of DUPACKs, arriving at the sender. There are then fewer chances of triggering a fast retransmit. If a fast retransmit does get triggered, ACK heuristic always works in the correct way. As explained earlier, with few ACKs a more-than-three-packet jump in acknowledgment number is more likely to occur, so ACK heuristic is biased to suppress a fast retransmit after receving three DUPACKs.

In this scenario, this happens to be the right decision. As for timestamp heuristic, if more than one packets arrive at the receiver before the delayed ACK is sent out, the echoed timestamps will be different from the case when the ACK is not delayed, and it may result in a fast retransmit being wrongly initiated. The timestamp included in every packet also introduces extra overhead when transmitting the same amount of data. So timestamp heuristic offers no improvement at all.

In the loss scenario (Fig. 4.10(b)), avoiding the fast retransmit is no longer correct, so occasionally ACK heuristic needs to wait for a second timeout when a retransmit sent after the first timeout is lost. As explained in the previous paragraph, timestamp heuristic is more likely to invoke a fast retransmit even if it could make a wrong detection with fewer ACKs. So most of the time, it can recover loss of retransmit before the retransmission timer expires. However, its timestamp overhead is still there pulling it from performing better. So TCP with either heuristic only performs the same as without bugfix – up to 60% improvement over TCP with bugfix, which always suffers from waiting for a second timeout. In the previous loss scenario without delayed acknowledgment, we have mentioned that although a fast retransmit would be invoked eventually, TCP with the Eifel-I-based approach can maintain the reduced congestion window at a higher value. Thus, it can continue transmitting at a higher rate. In this scenario, it can achieve more than 110% improvement over TCP with bugfix and more than 30% improvement over TCP without bugfix or with either heuristic.

4.5 Summary

In this chapter, we have presented a detailed discussion on the proposed Eifel-I approach. This new approach is based on the modified selective use of timestamps in TCP headers. It avoids the problems caused by the persistent use of timestamps in the current Eifel approach. In conjunction with Eifel-I, we have also suggested a simple modification to the TCP retransmission timer for avoiding future spurious timeouts and an enhancement to non-SACK TCPs (such as TCP Reno or NewReno) for avoiding multiple fast retransmits. The modified retransmission timer is capable of getting RTT samples from

both originally-transmitted packets and retransmitted packets. This capability enables the timer to have a more responsive RTO estimation and avoid more future spurious timeouts.

Chapter 5

Implementations

For a quick implementation and evaluation of Eifel-I and also to have a controllable and repeatable environment with various TCP flavors (TCP Reno, NewReno and SACK), we have chosen to use the NS-2 network simulator [37] for conducting our evaluation experiments. In this chapter, we first present some background knowledge on the simulator. We then briefly describe the implementations of Eifel-I, Eifel and some other approaches in the simulator. The experiment results will be presented in the next chapter.

5.1 The NS-2 Network Simulator

NS-2 [37] is a public-domain, object-oriented, discrete-event simulator targeted at networking research. It is primarily used to simulate local and wide area networks. It provides substantial support for the simulation of common Internet protocols (e.g., TCP, UDP), routing, and multicast protocols over wired and wireless networks.

In the following subsections, we will present a brief overview of the basic simulator, consider how a link is simulated, and describe in detail the various TCP modules available in NS-2.

5.1.1 Overview of NS-2

In general, the simulator supports the following features:

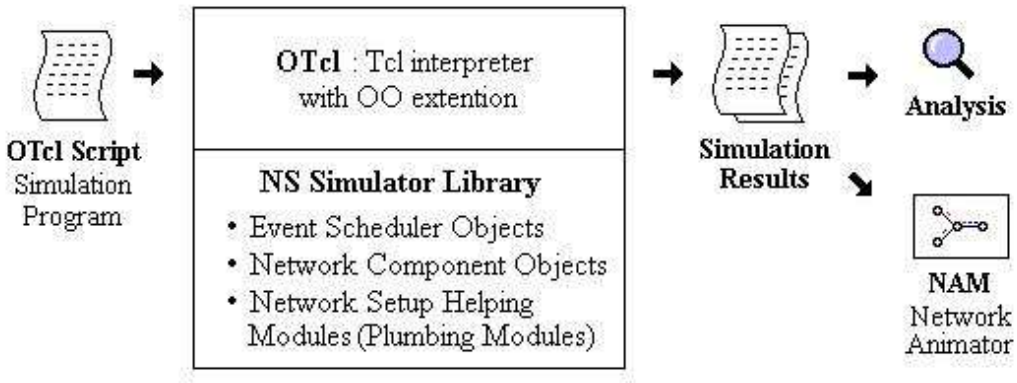


Figure 5.1: A simplified user's view of NS-2

- Elements for network topology – nodes and point-to-point links
- Network protocols – TCP, UDP, etc.
- Traffic source behaviors – FTP, Telnet, Web, constant bit rate (CBR) and variable bit rate (VBR).
- Router queue management mechanisms – drop-tail, random early detection (RED), class-based queue (CBQ), and more.
- Routing – unicast, multicast and hierarchical.
- Basic mobility – ad hoc networks, mobile IP.

All these features are implemented in NS-2 as objects or modules.

As shown in Fig. 5.1, NS-2 is an object-oriented Tcl (OTcl) script interpreter that is equipped with simulation event schedulers, network component object libraries, and network setup module libraries. To set up and run a simulation network, a user should write an OTcl script that initiates an event scheduler, set up the network topology using the network objects and the helping modules in the library, and tell traffic sources when to start and stop transmitting packets through the event scheduler. After the simulation program is executed by the simulator, the simulator generates traces as simulation results. A user can use some analyzing tools to examine the trace files for an in-depth understanding of the simulation, or choose to visualize the simulated network and the transfer of packets by the Network Animator (NAM), which works in a pair with NS-2.

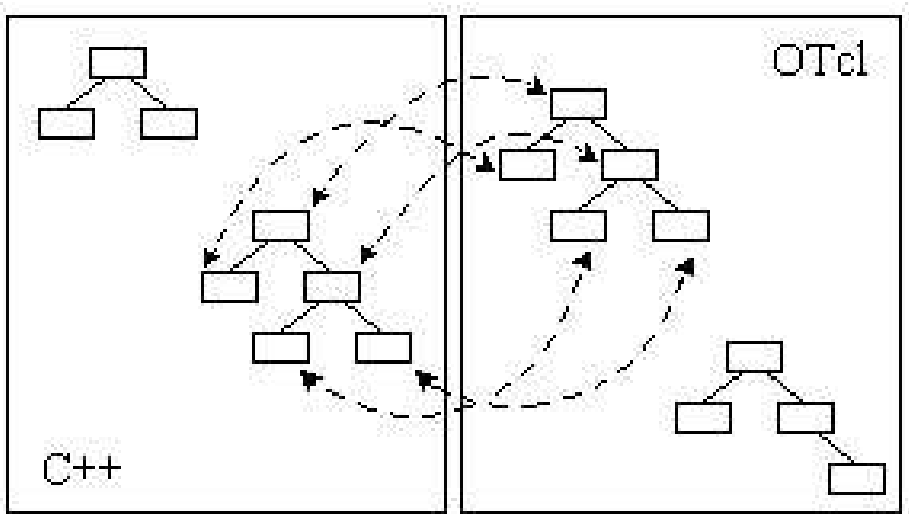


Figure 5.2: The correspondence between OTcl and C++

NS-2 is not only written in OTcl but in C++ also. It uses a split-language programming approach, and the objects in the simulation environment are mainly implemented using a combination of OTcl and C++. A class hierarchy in C++ is called the compiled hierarchy and a similar class hierarchy with the OTcl interpreter is the interpreted hierarchy. The interpreted hierarchy is for quick setup and configuration of simulations while the compiled one is for fast simulation processing. As in Fig. 5.2, from a user's perspective, there is a one-to-one correspondence of classes in both hierarchies. However, in the figure, there are also some objects in C++ that do not need to be controlled by the interpreter or internally used by another object. Hence, they do not need to be linked to OTcl. Likewise, an object can be entirely implemented in OTcl.

5.1.2 A Link in NS-2

A link is a major compound component in NS-2, and is used for connecting two simulated nodes. There are two kinds of links: a *simplex* link that forms a unidirectional connection from one node to another, and a *duplex* link that constructs a bi-directional connection from two simplex links.

Packets passing through a simplex link go through several components as shown in Fig. 5.3. The *Queue* is in fact the output queue of the starting node. However, it is implemented as part of the simplex link in NS-2. Packets dequeued from the output

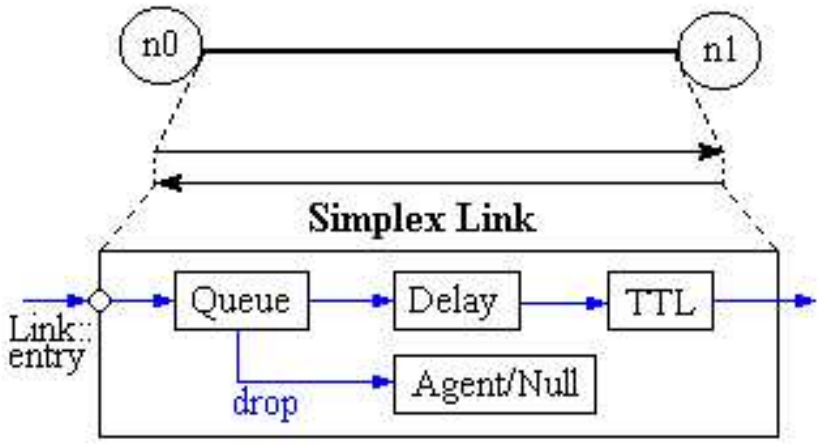


Figure 5.3: The structure of a link in NS-2

queue are passed to the *Delay* object that simulates the link delay, and packets dropped at the queue are sent to the *Null* Agent and are freed there. Finally, the *TTL* object calculates Time To Live parameters for each packet received and updates the TTL field of the packet. Then the packets get out of the link and reach the end node.

5.1.3 TCP Agents in NS-2

In this section, we briefly describe the various TCP modules available in NS-2. Referring to Fig. 5.1, the TCP modules are in fact *network component objects*.

There are two types of TCP agents in NS-2: one-way agents and a two-way agent. One-way agents are further subdivided into a set of TCP senders (which follow different congestion control and error recovery mechanisms) and receivers or sinks. The two-way agent is symmetric in that it represents both a sender and receiver. It is still under development.

The simulator supports several versions of an abstracted TCP sender, such as TCP Tahoe, TCP Reno, TCP NewReno [17], TCP SACK [35], and more that correspond to the various flavors of real TCP implementations. However, these objects only attempt to capture the essence of TCP congestion and error control behavior, but are not intended to be faithful replicas of real-world TCP implementations. For example, they do segment number and ACK number computations entirely in packet units; there is no SYN/FIN connection establishment/teardown; and no data is ever transferred (e.g., no checksums

or urgent data).

Each TCP sender object must pair with a corresponding TCP sink object. The TCP sink is responsible for returning ACKs to its peer sender. It can generate one ACK per packet received, less than once per packet to implement Delayed Acknowledgment [46], or model a selective acknowledgment after the description of TCP SACK in RFC2108 [35] and its extension in RFC2883 [18].

The two-way TCP agent (FullTCP) is a new addition to the suite of TCP agents supported in the simulator and is still under development. It is different from (and incompatible with) the other agents, although it does have some similarity in architecture with the others. It differs from the one-way agents in the following ways:

- Connections may be established and torn down (SYN/FIN packets are exchanged)
- Bi-directional data transfer is supported
- Sequence numbers are in bytes rather than packets

Currently, FullTCP is only implemented with Reno congestion control.

5.2 Implementation of Eifel-I and Others

The NS-2 network simulator is regularly updated by a group of maintainers. The version of NS-2 we have used is the *ns2.1b9a* all-in-one. The *all-in-one* is a package which contains the required components and some optional components used in running NS-2.

For simulation purpose, we have implemented the proposed Eifel-I approach into TCP Reno, NewReno, and SACK modules in ns2.1b9a. We have modified the Timestamp option to allow selective use of timestamps. In the existing TCP implementations, the Timestamp option may be used independently in each direction. However, for simplicity and efficiency, the timestamp for one direction and the timestamp for the other direction are combined into a single TCP Timestamp option. In our modified Timestamp option, if one direction does not need timestamps, the part in the option for this direction will always be set to 0, indicating that it is not in use. As mentioned in Section

3.1, the option's original functionalities are not needed for wireless links, so it is safe to change to selective use. The implementation provides for both detection and response to spurious timeouts and spurious fast retransmits. It provides several options for restoring the congestion control state, and in response to a spurious timeout, it also provides different options for adapting the retransmission timer. The default is to fully restore the congestion control state and ensure the one-sample-per-window rate regardless of retransmits. Currently, the implementation responds in the same way, independent of the number of consecutive timeouts. However, different responses depending on the number of timeouts occurred can be easily added.

We used the implementation of Eifel for NS-2 by Andrei Gurtov [25]. This implementation also allows the use of different TCP flavors including Reno, NewReno, etc. It provides both detection and response to only spurious timeouts in TCP using the Timestamp option.

The DSACK extension is already available in this version of NS-2, so we only added in the detection for whether the duplicate packet has been retransmitted before when an ACK arrives containing a DSACK block, and the response of several options of congestion control restoration. This implementation is contributed by Zhu Yingjie, to which we add our slight modification.

To trigger spurious fast retransmits in our experiments, we patched NS-2 with a module called *hiccup* [42]. The module allows for simulating delay spikes, packet reorderings, and losses of packets. It is derived from the Queue class in NS-2 and it is inserted into a Link before the Queue in Fig. 5.3. Although this module can introduce both packet reorderings and delays, we only used it to generate reorderings. As the Queue in a Link simulates the output queue of the sender, it is not reasonable to introduce a delay before the packets ever enter the network. So to introduce delay spikes, we used another small patch for the simulator [25]. This code is only for triggering delay spikes of arbitrary length (e.g., a few seconds) for all queue types in NS-2. Compared to *hiccup* this code has two benefits: it places delays after the Queue in Fig. 5.3 and it can be used in both directions simultaneously. A remaining drawback is that it allows packets currently "in

the air" to complete transmission after a delay spike begins.

Note that all the above implementations need modifications in both the background C++ and the forefront OTcl codes.

Chapter 6

Experiments by Simulation

In this chapter, we present detailed simulation results on evaluating Eifel-I, generated using the NS-2 simulator [37]. We include TCP with DSACK (TCPd), TCP with Eifel (TCPe), TCP with F-RTO (TCPf), and the original TCP with no enhancements (TCPo) for comparison with TCP with Eifel-I (TCPi).

6.1 General Settings for Experiments

The basic simulation topology used for all our experiments is shown in Fig. 6.1 [1]. S_i, i = 1, ..., n, corresponds to the set of TCP source nodes sending packets to the set of TCP sink nodes M_i, i = 1, ..., n. Each pair of S_i, M_i nodes forms a TCP connection. The BS (or the RNC in UMTS terminology) is connected to the M_i nodes via a virtual node (VN). The link between BS and VN is the simulated wireless link with independent uplink and downlink bandwidth and a drop-tail queuing policy. The virtual node, VN, is included for simulating the sharing of the wireless link among the mobile hosts, M_i. The latency and bandwidth between S_i and BS is set to 3Mbps and 50 milliseconds (ms) respectively. The set of S_i represents the remote hosts in the Internet, which reach the mobile hosts via BS. The latency and bandwidth between VN and M_i are 100Mbps and 1ms. The uplink and downlink's bandwidths and latencies are choosen from typical values of either GPRS [24] or UMTS [49] networks. For example, to simulate a GPRS link, we have used

[1] This topology is similar to the ones in [13], [23], or [50]

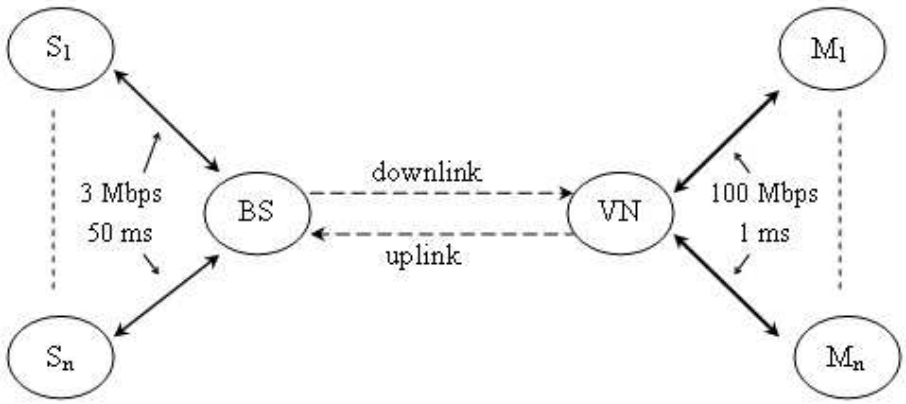


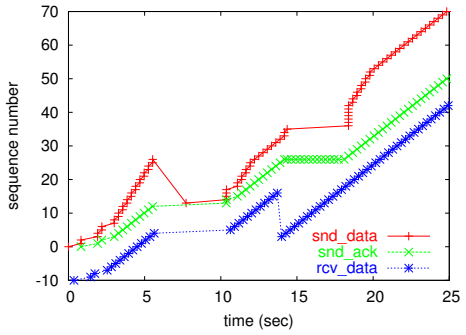
Figure 6.1: Simulation topology

350ms and 30kbps as the latency and bandwidth for downlink, and 350ms and 10kbps for uplink; for a UMTS link, we have used 150ms and 384kbps for downlink and 150ms and 64kbps for uplink. One-way TCP agents (TCP Reno, NewReno and SACK) with delayed acknowledgments in NS-2 are used. The delayed ACK timer is implemented as a heartbeat timer with 200ms granularity. To perform a simulation, a selected one-way TCP sender agent is attached to the S_i node and its corresponding sink agent to the M_i node.

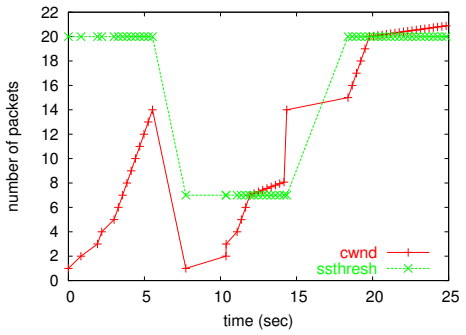
Our experiments are based on transferring bulk data by using the FTP agent. The TCP maximum (advertised) window size is set at 500KB. Using such a large window size ensures that TCP is never window limited in all the experiments. The TCP packet size used is 1KB. This is the default value in the simulator. Except those explicitly specified, the default settings for all parameters in NS-2 are used. We use both download time and number of transmitted packets as the metrics for performance comparison. All the values shown are averaged over 100 repetitions to ensure sound statistics.

6.2 A Single Spurious Timeout

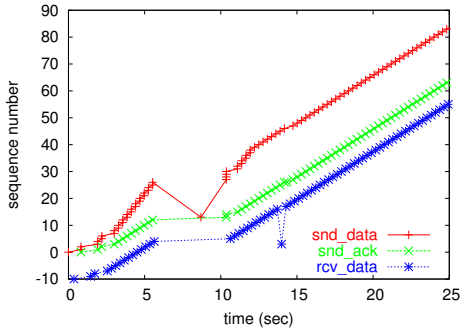
In this section, we illustrate the different approaches' behavior in the presence of a single spurious timeout. The figures for the first three approaches are the same as the ones presented in Section 3.3.2 for explaining each individual approach. They serve to verify the implementation of the algorithms into NS-2.



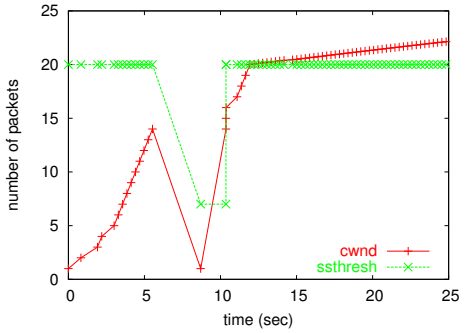
(a) DSACK: time sequence



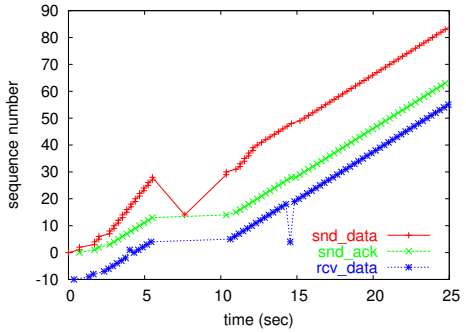
(b) DSACK: congestion control state



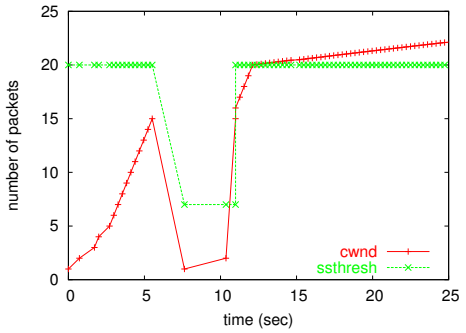
(c) Eifel: time sequence



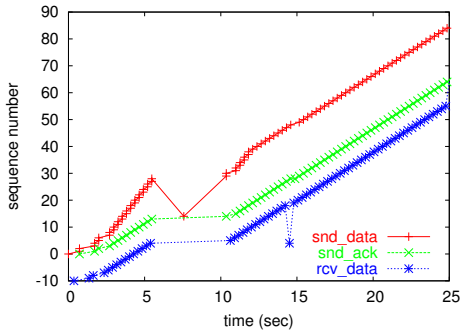
(d) Eifel: congestion control state



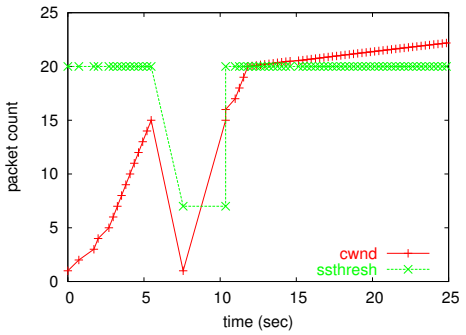
(e) F-RTO: time sequence



(f) F-RTO: congestion control state



(g) Eifel-I: time sequence



(h) Eifel-I: congestion control state

Figure 6.2: A spurious timeout

As expected, DSACK cannot avoid the go-back-N retransmission and it only restores *cwnd* and *ssthresh* after it receives the ACK containing the DSACK block. Eifel, F-RTO and Eifel-I are quite similar in their time-sequence plots: there is no go-back-N retransmission; the only penalty is one unnecessary retransmit. However, Eifel and Eifel-I have the spurious timeout occurring at slightly different times, indicating their difference in RTO estimation. As we have explained in Section 4.3.1, the one-RTT-per-packet sampling rate in Eifel leads to largely fluctuating RTO values. The plots of congestion control state also show F-RTO can make a decision only after the second ACK has arrived.

6.3 Scenarios and Discussions

We now present our detailed studies on the effect of delays and packet losses simulated using different distributions. Delays and losses can be caused by handovers, link layer retransmissions, etc. We will examine various scenarios in the following sections.

Wireless link mechanisms such as link layer retransmissions and inter-system handovers are complex and difficult to completely represent in a model. As suggested in [21], if the purpose of the wireless link models is only for evaluating the effect of link-level mechanisms on end-to-end transport protocols, simply changing link characteristics and introducing losses and delays to traffic are often sufficient for understanding transport protocol performance in the presence of the modeled wireless link. So in the following evaluation, instead of fully implementing the wireless link layer into the simulator, we model the various link layer scenarios by introducing delays and losses based on models that closely mimic the corresponding real-world scenarios. As mentioned before, because of the rather persistent link-layer error protection, 2.5G/3G wireless links have a negligibly small error rate [27]. In GPRS LLC, the maximum number of retransmissions is set to 3 and in UMTS RLC, the number of retransmissions can be set to a maximum value of up to 40. So in our simulations we assume a fully persistent link layer protection; packet losses are then mostly due to congestion. Studies [48] [36] on interactions between wireless link layer ARQ and TCP have shown that fully reliable

ARQ protocols are the best choices from the TCP perspective. In fact, whereas a residual packet loss left over by not fully reliable TCP protocols may not degrade appreciably TCP throughput performance as long as it is a fraction of the overall end-to-end TCP packet loss, no apparent performance advantages (e.g., energy savings) come from limiting the number of retransmission attempts at the wireless link layer. Thus, the use of a fully persistent link layer retransmission is reasonable.

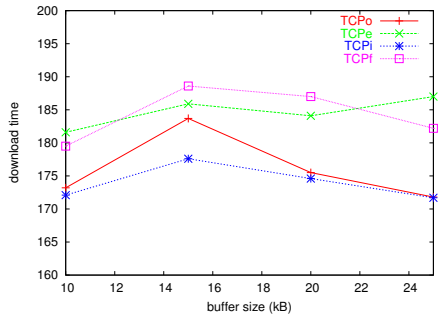
Over a GPRS link, the size of bulk data transferred is 500KB, and over a UMTS link, the data size is 3MB. TCPo, TCPd, TCPe, TCPf and TCPi are all evaluated in the same simulated network. The MTU is a fixed value for all of them. Since TCPo, TCPd, TCPf and TCPi do not need the 12-byte timestamp, they can include 12-byte more data in each TCP packet. We use the default TCP packet size in NS-2 (1000 bytes) for TCPe, making the packet size for the others 1012 bytes. So, for a 500KB data transfer, TCPe needs to send 500 original packets, while the others need only (500,000/1012=) 495 packets; for a 3MB data transfer, TCPe needs to send 3000 original packets, and the others send just (3,000,000/1012=) 2965 packets.

6.3.1 Variable Delays and Losses due to Handovers

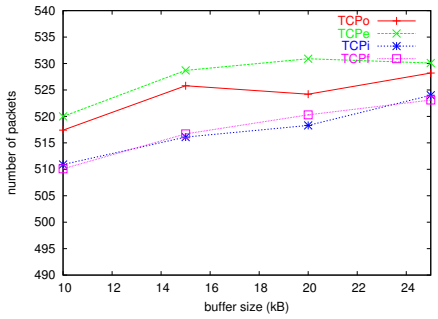
According to measurements over real GPRS networks [24], variable delays caused by handovers in a cellular network can be well modeled by two parameters: the interval between the occurrence of delays and the length of each delay. In the evaluation, the intervals are drawn from a uniform distribution between 20 and 140 seconds, with a length uniformly distributed between 3 and 15 seconds. Experiments have been conducted across different buffer sizes in the bottleneck wireless link.

Single Connection

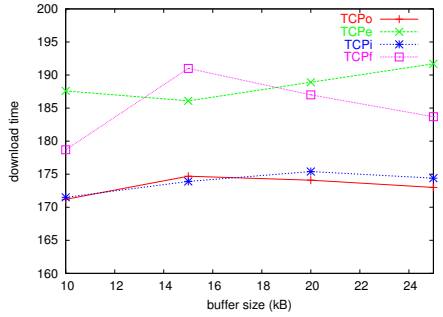
Plots in Fig. 6.3 show how TCP Reno and Newreno with or without one of the approaches vary with different buffer sizes. As we have used a 500kB maximum window size, for all TCP senders, their slow-start transmission will be terminated by multiple packet losses in a single window due to congestion. This is because the congestion



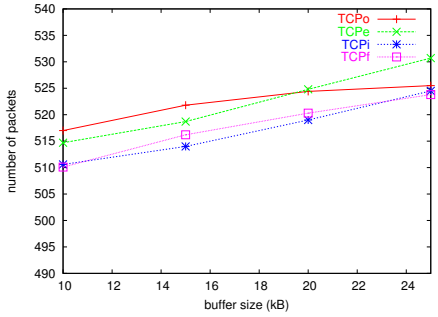
(a) Time for Reno with bugfix



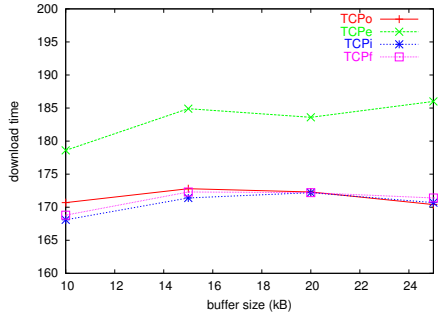
(b) Packets for Reno with bugfix



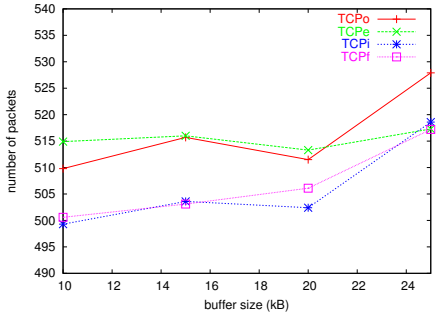
(c) Time for Reno without bugfix



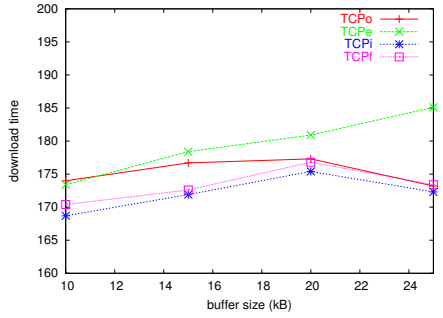
(d) Packets for Reno without bugfix



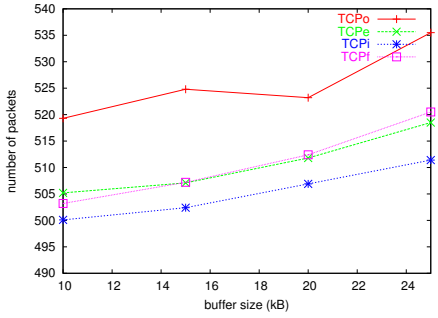
(e) Time for Newreno with bugfix



(f) Packets for Newreno with bugfix



(g) Time for Newreno without bugfix



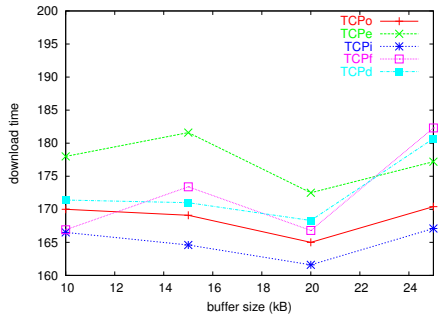
(h) Packets for Newreno without bugfix

Figure 6.3: TCP Reno and Newreno during a handover with different buffer sizes

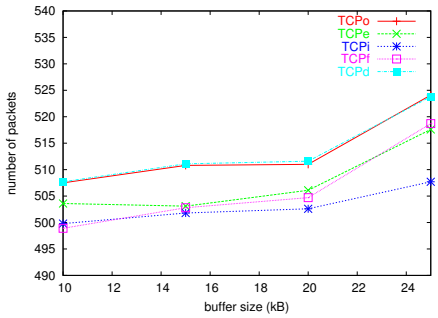
window will overload the bottleneck buffer after a certain amount of time. After this point, the senders enter the stationary congestion avoidance state with slow increase and periodic decrease of *cwnd* caused by a single congestion loss.

A Reno sender with either Eifel or F-RTO (TCPe and TCPf) performs badly, compared with Eifel-I (TCPi) in terms of download times. Referring to the discussion in Section 4.4, Eifel-I can efficiently handle multiple packet losses in a single window using fast retransmit and recovery, thus avoiding the lengthy retransmission timeout. As Eifel and F-RTO have no such capability, such multiple losses usually lead to timeout no matter whether bugfix is enabled or not. For the original TCP (TCPo), a timeout will often be triggered when bugfix is enabled while fast retransmit and recovery may recover some losses when bugfix is disabled. The reason that TCPe and TCPf are worse than TCPo in the case without bugfix is that: in the window with multiple losses, the first one or two packets usually arrive at the receiver successfully, which leads either TCPe or TCPf to detect the timeout as a spurious one and switch back to sending unsent packets but not retransmitting lost packets. This further delays the recovery of lost packets in the previous window and may also aggravate the problem by forcing the sender to wait for another timeout. So although the number of packets transmitted by TCPo, TCPe and TCPf are only slightly higher than TCPi, their download times can be much worse. TCPi can achieve up to 10% improvement over TCPe and TCPf and 5% improvement over TCPo.

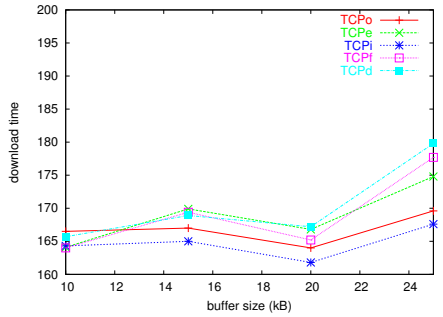
As NewReno can recover multiple losses more efficiently than Reno, the inefficiency in F-RTO's loss recovery is lessened. TCPf generally performs similiarly as TCPi in download times. Because both TCPf and TCPi are better than the original TCP (TCPo) in handling spurious timeouts, and so perform slightly better than TCPo. TCPe performs worse than TCPo most of the time. Similar to the plots in Fig. 4.2, during the slow-start period, the RTO value calculated by Eifel shortly collapses to a small value, so TCPe is prone to initiating a timeout during the multiple packet losses. It still suffers from the problem of delayed loss recovery and a second timeout, as we have explained in the last paragraph. In the plots of number of packets, we can see that: TCPi and



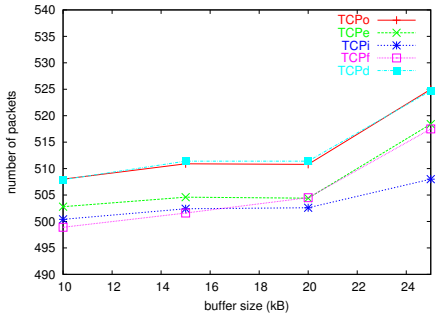
(a) Time for Sack with bugfix



(b) Packets for Sack with bugfix



(c) Time for Sack without bugfix

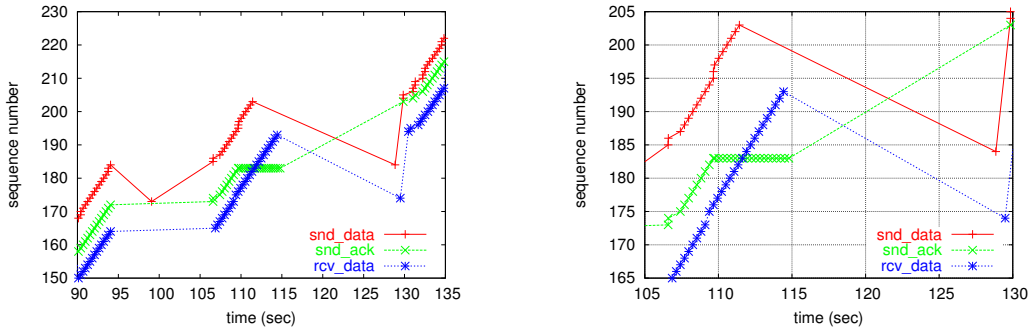


(d) Packets for Sack without bugfix

Figure 6.4: TCP Sack during a handover with different buffer sizes

TCPf transmit fewer packets than TCPo by avoiding some future spurious timeouts, and TCPi is better than TCPf in certain cases because it can also collect RTT samples during retransmission (see the explanation of Eifel-I's modified retransmission timer in Section 4.3). As illutrated in Section 4.3, with its over-aggressive RTO estimation, TCPe cannot avoid future spurious timeouts and it also incurs with a timeout similar to TCPo, so it often transmits more packets than TCPo.

Fig. 6.4 shows the results of TCP Sack with or without either Eifel, Eifel-I, F-RTO or DSACK. TCP SACK can recover multiple packet losses efficiently, so it mitigates Eifel and F-RTO's inability in recovering packet losses. Refering to Fig. 6.5, when bugfix is enabled, TCPo, TCPd, TCPe, and TCPf suffer from a single congestion loss after a delay spike. It is because the packets delayed at the bottleneck link are prone to trigger congestion loss and the congestion control restoration by Eifel, F-RTO and DSACK can aggravate the level of congestion. As bugfix suppresses the fast retransmit, they have to wait for the retransmission timer to expire before the loss packet is transmitted again. As



(a) A whole view

(b) A clearer view of the lost packet

Figure 6.5: A spurious timeout on a congested link - with bugfix

illustrated in Section 4.4, TCPi can determine that it is a real packet loss upon receiving the ACK of the first packet in the next window. It can recover the loss with fast retransmit and fast recovery, similar to the scenario presented in Fig. 4.8. After a spurious timeout, although Eifel-I takes the same congestion control restoration as the other approaches, it eliminates the impairment on TCP performance introduced by congestion losses caused by the aggressive retoration at the same time. So it can keep the TCP sender transmitting stably at a higher speed.

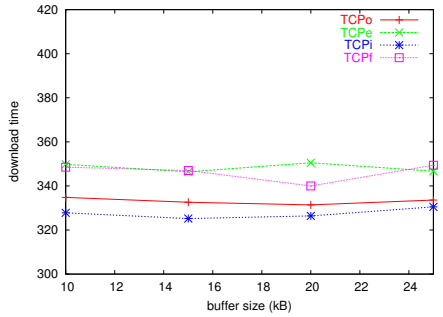
Comparing the plots in both Fig. 6.3 and 6.4 across different TCP flavors (Reno, NewReno and Sack), TCP Sack senders can transmit the same amount of packets in a shorter time than TCP NewReno, which further has a shorter download time than TCP Reno. This verifies the correctness of our results as more advanced loss recovery mechanisms like Sack and NewReno can handle packet losses more efficiently, and thus have a shorter download time. Because Eifel-I has enhanced TCP Reno and NewReno's capability in handling multiple packet losses, its download times remain quite stable around 170 seconds.

Compared with other approaches like DSACK, Eifel, F-RTO and the original TCP, Eifel-I delivers robust performance across different buffer sizes. This property is very important in the varying delay environment of a wireless system, since it is difficult to size the system with an optimal buffer size, given that the capacity of the link also varies over time.

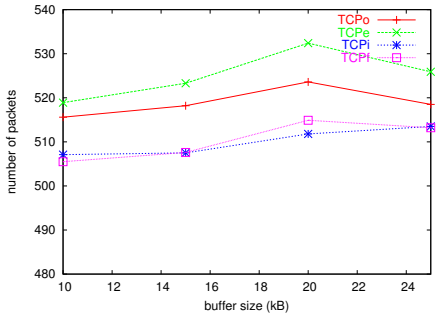
Multiple Connections

We have also done the same experiments using multiple connections. In the network topology presented in Fig. 6.1, in addition to the TCP connection between S_1 and M_1, we set up more connections between the other pairs of nodes, where all the pairs use the same TCP sender and sink agents (Reno, NewReno or Sack). Each pair transmit the same amount of bulk data during the same time period, so different pairs need to compete for the shared, limited wireless link between BS and VN. During the transmission period, we trace the performance of the TCP connection between S_1 and M_1. The results of the two-concurrent-connection case are shown in Fig. 6.6 and 6.7, and the results of the four-concurrent-connection case are shown in Fig. 6.8 and 6.9. Except that the download times are roughly doubled or quadrupled compared with their corresponding values in the single-connection case, the relative changing patterns and performance improvement among the different approaches (DSACK, Eifel, Eifel-I, F-RTO, and the original TCP) are quite similar. Compared with the single-connection case, the performance degradation of a Reno sender with either Eifel or F-RTO is less severe because the multiple packet losses are shared by the multiple concurrent connections. So the impairment on each individual connection is lessened. However, the overall degradation is still quite severe.

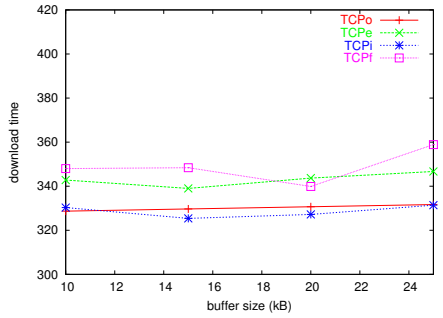
Interestingly, in the four-connection case, the download time of F-RTO is significantly worse than the others. In certain cases, the best performing approach Eifel-I can achieve more than 20% improvement over F-RTO. According to our observations, the reason for F-RTO's bad performance is because when it receives the first non-duplicate packet after a timeout, it responds with two packets. As all the concurrent connections would experience the delay and initiate a timeout around the same period, with more connections, more packets are injected into the network in response to the first non-duplicate ACKs. The aggressive transmitting behavior can lead to real congestion losses at the bottleneck link, thus greatly hurting TCP performance.



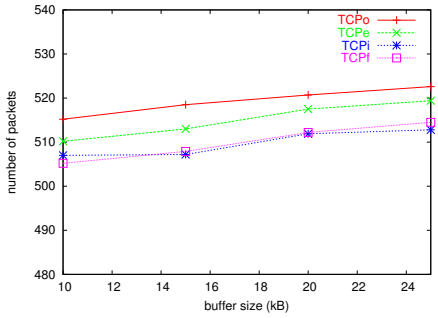
(a) Time for Reno with bugfix



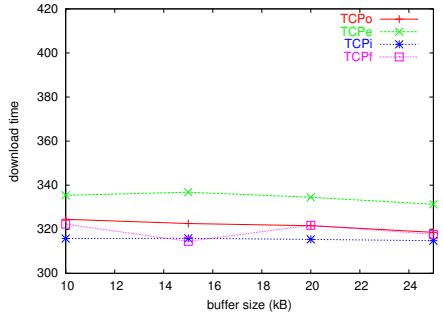
(b) Packets for Reno with bugfix



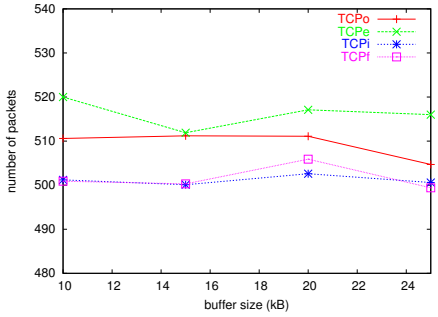
(c) Time for Reno without bugfix



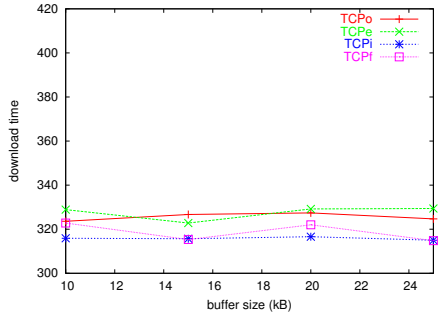
(d) Packets for Reno without bugfix



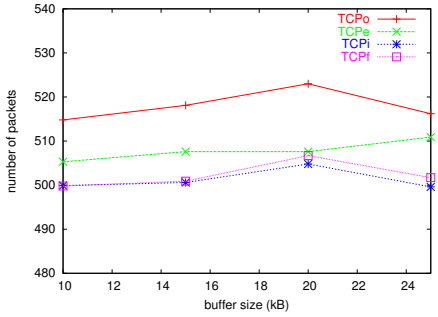
(e) Time for Newreno with bugfix



(f) Packets for Newreno with bugfix

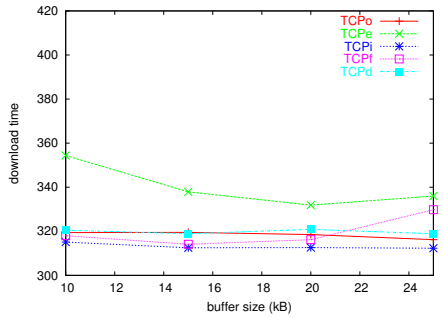


(g) Time for Newreno without bugfix

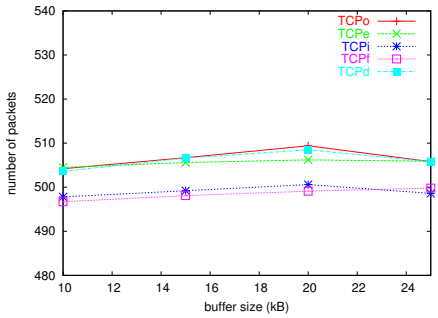


(h) Packets for Newreno without bugfix

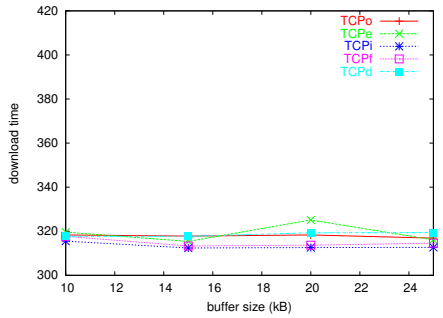
Figure 6.6: TCP Reno and Newreno during a handover with different buffer sizes – two connections



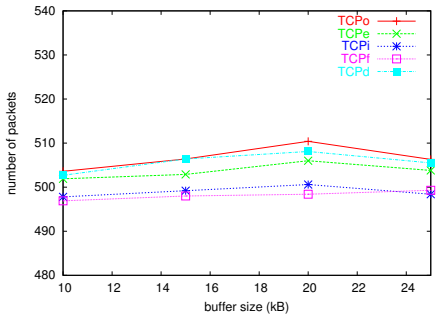
(a) Time for Sack with bugfix



(b) Packets for Sack with bugfix



(c) Time for Sack without bugfix



(d) Packets for Sack without bugfix

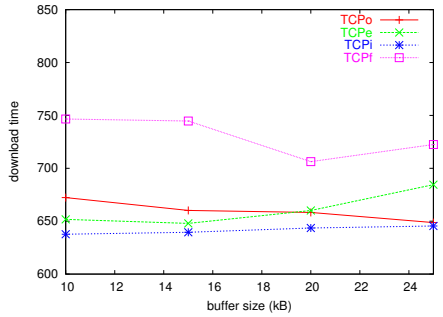
Figure 6.7: TCP Sack during a handover with different buffer sizes – two connections

In conclusion, we want to highlight here that in all the scenarios (of different TCP flavors, or different number of connections, etc.), TCPi is always better than or at least the same as the other approaches. In certain cases, it can achieve 10% to 20% performance improvement over the others. Its good performance is consistent and stable because it can handle both variable delays and packet losses efficiently.

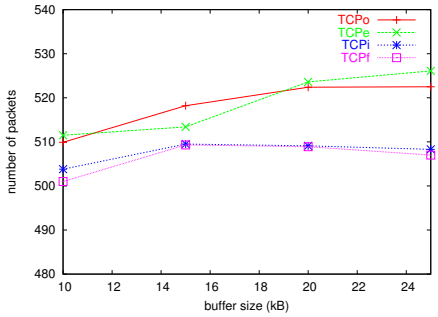
Papers evaluating approaches like Eifel [22] and F-RTO [45] adopt a small maximum window size. In those evaluations, the enhanced TCP performs better than the original TCP in facing handover delays. As Eifel-I is an improvement from Eifel, it should have similar or even more improvement over the original TCP. However, as the results presented in this section have shown, when the receiver has a larger window size, which would not limit the transmission of the sender before congestion occurs in the bottleneck link, the problems caused by multiple packet losses in a single window (before entering congestion avoidance) would largely degrade the improvement that can be gained with enhancements like Eifel or F-RTO. So without an efficient loss recovery mechanism, they perform even worse than the original TCP. As Eifel-I is introduced with an efficent loss recovery mechanism for non-Sack TCPs (see Section 4.4), it maintains its improvement over the original TCP in both cases (a small or a large window size), and when a large window size is used, it can perform much better than approaches like Eifel and F-RTO.

6.3.2 Variable Delays due to Link Layer Retransmissions

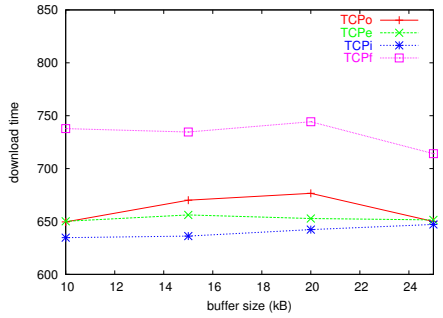
In addition to handovers, another major cause of variable delays is link layer retransmission (LLR), which has been widely deployed in 2.5G/3G wireless networks. As noted in several papers [45], [13], [21], the occurring pattern of delay jitters introduced by LLR can be well captured by an exponential distribution, and a mean value of up to one second appears to be reasonable. In this section, we evaluate the impact of delay jitters generated using different means (0.3, 0.5, 0.7 and 0.9 second).



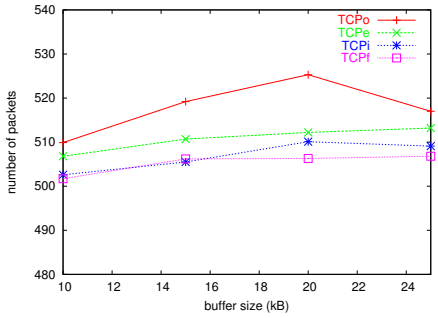
(a) Time for Reno with bugfix



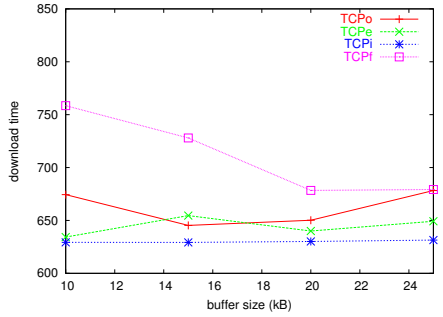
(b) Packets for Reno with bugfix



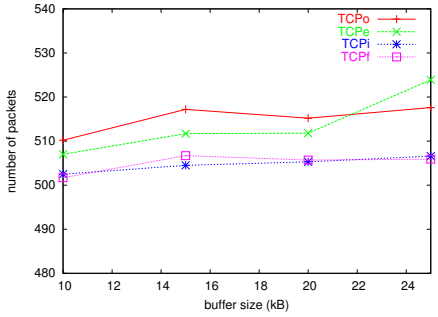
(c) Time for Reno without bugfix



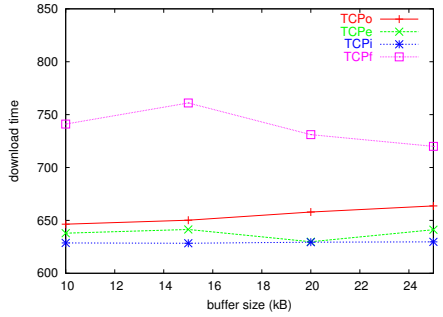
(d) Packets for Reno without bugfix



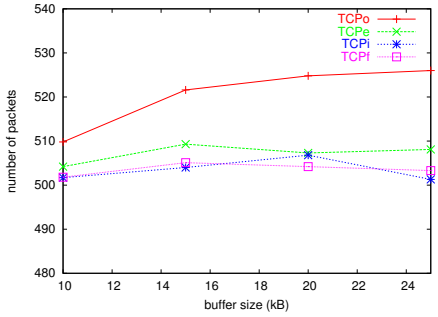
(e) Time for Newreno with bugfix



(f) Packets for Newreno with bugfix

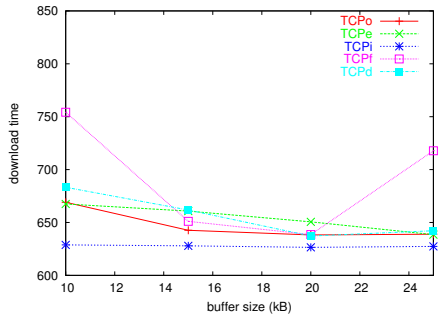


(g) Time for Newreno without bugfix

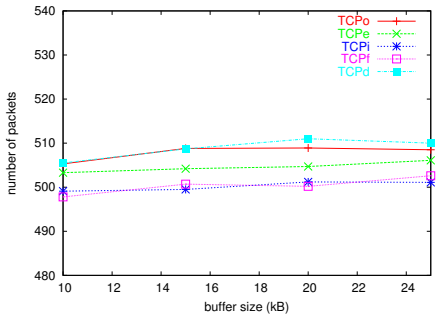


(h) Packets for Newreno without bugfix

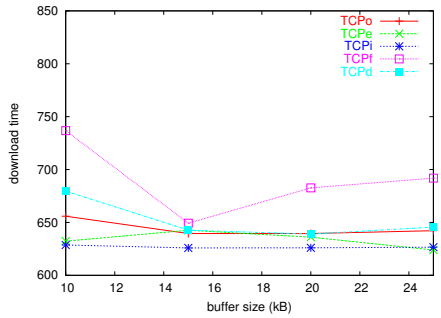
Figure 6.8: TCP Reno and Newreno during a handover with different buffer sizes – four connections



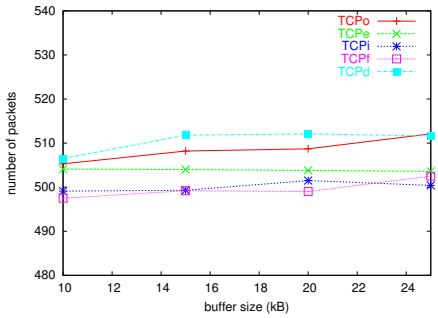
(a) Time for Sack with bugfix



(b) Packets for Sack with bugfix



(c) Time for Sack without bugfix



(d) Packets for Sack without bugfix

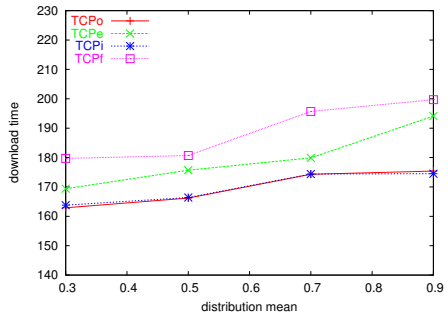
Figure 6.9: TCP Sack during a handover with different buffer sizes – four connections

Single Connection

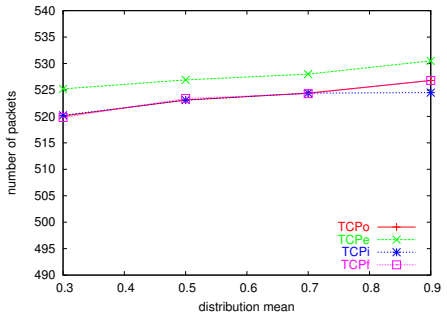
Fig. 6.10 and 6.11 show the performance evaluation results of different TCP senders in a GPRS network facing different levels of link layer packet losses and recovery. As expected, the increase in the distribution's mean value, which indicates the amount of link layer packet retransmissions, leads to an increase in download time.

Similar to the results shown in the previous section, Eifel and F-RTO suffers from multiple packet losses when working with a less loss-robust TCP such as TCP Reno. In such cases, they perform worse than the original TCP. In contrast, Eifel-I achieves at least the same download time as the original TCP while transmitting fewer packets by avoiding some unnecessary retransmissions and future spurious timeouts. This is possible as Eifel-I is capable of handling spurious retransmissions and recovering packet losses efficiently.

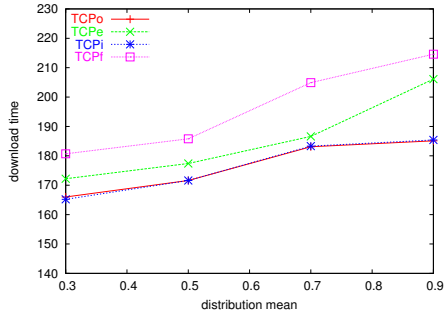
We have also experimented with the impact of link layer retransmissions in a UMTS network. The results are presented in Fig. 6.12 and 6.13. Over a UMTS network, the original TCP performs the worst in terms of download time. This is because compared with the GPRS link, the UMTS link has a smaller link latency and higher bandwidth, so the average RTTs over UMTS are smaller than those over GPRS. This in turn leads to a smaller RTO value estimated by TCP in UMTS. Therefor, in the presence of the same delay variations, TCP senders over UMTS are more sensitive and initiate a spurious timeout more frequently. In the other words, TCP with larger RTO values are generally more robust against delay spikes. When multiple packet losses can be recovered more efficiently (either through fast retransmit when bugfix is disabled, or with a more advanced loss recovery mechanism like Sack or NewReno), spurious timeouts become the main impairment to the TCP connection's performance. In these cases, TCPe and TCPf perform better than TCPo. TCPi always performs the best with any TCP flavor, with or without bugfix. In terms of download time and in certain cases, TCPi can achieve up to 40% improvement over TCPo, more than 30% improvement over TCPd, and up to 20% improvement over either TCPe and TCPf. TCPi and TCPf transmit the same amount of packets most of the time, which is a smaller amount than the others. TCPo



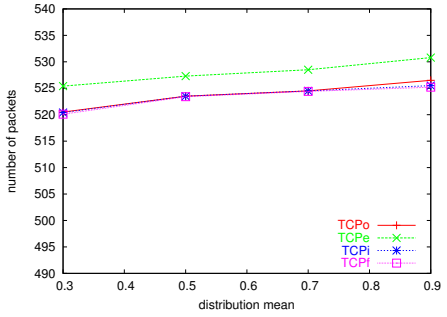
(a) Time for Reno with bugfix



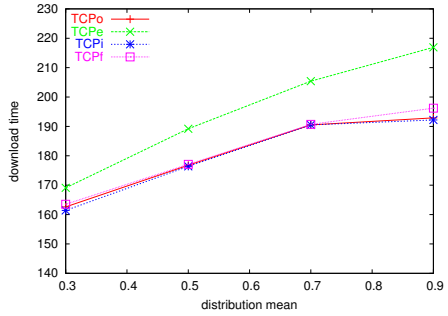
(b) Packets for Reno with bugfix



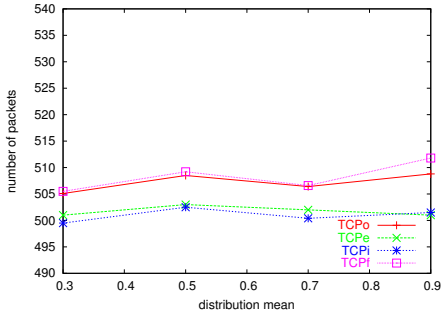
(c) Time for Reno without bugfix



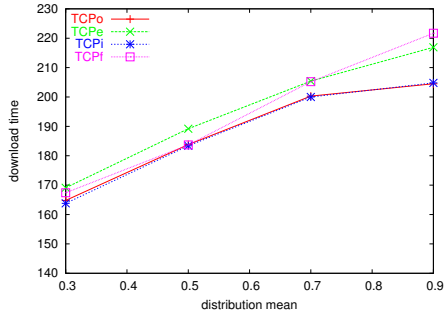
(d) Packets for Reno without bugfix



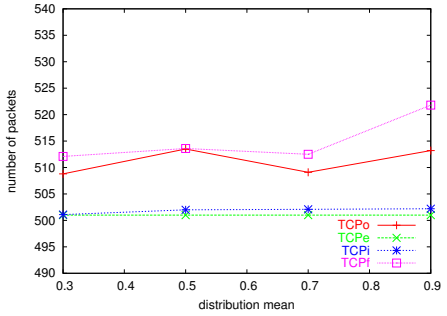
(e) Time for Newreno with bugfix



(f) Packets for Newreno with bugfix

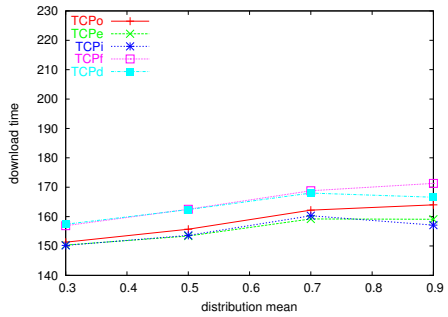


(g) Time for Newreno without bugfix

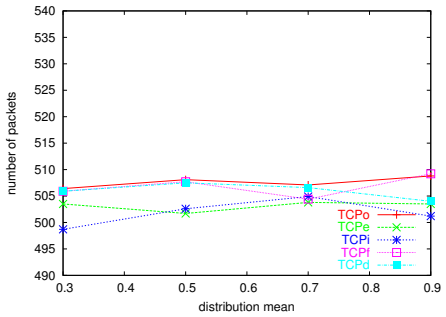


(h) Packets for Newreno without bugfix

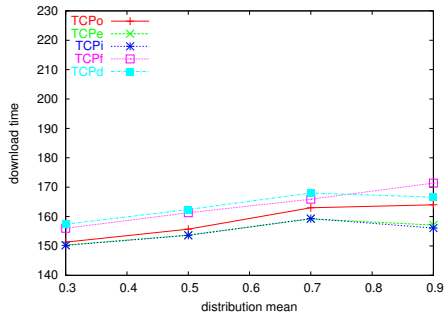
Figure 6.10: TCP Reno and Newreno in GPRS with LLR



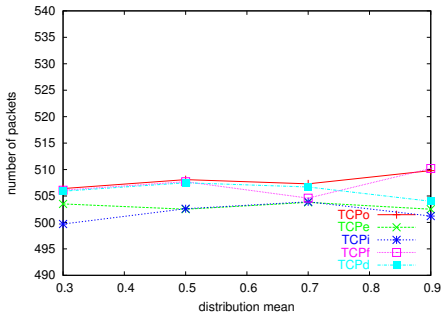
(a) Time for Sack with bugfix



(b) Packets for Sack with bugfix



(c) Time for Sack without bugfix



(d) Packets for Sack without bugfix

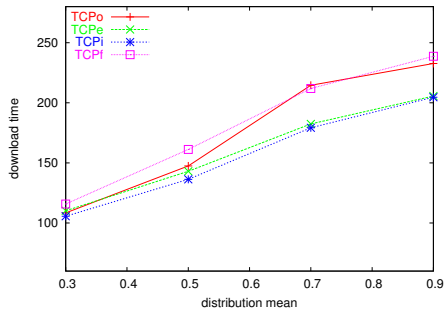
Figure 6.11: TCP Sack in GPRS with LLR

or TCPd transmit more packets mainly because of the go-back-N retransmission. TCPe also transmits more packets because of its RTO estimation, as explained in the previous section.

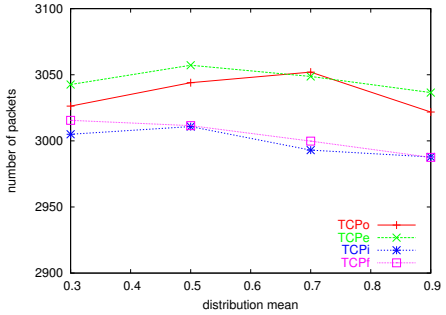
Multiple Connections

We have also experimented the same LLR condition with multiple concurrent TCP connections. Here, we present the results of the two-connection and four-connection scenarios in Fig. 6.14 and 6.15, and Fig. 6.16 and 6.17, respectively. The changing pattern and relative improvement among different approaches are similar to the single-connection case, where Eifel-I maintains its good performance across different scenarios over the other approaches.

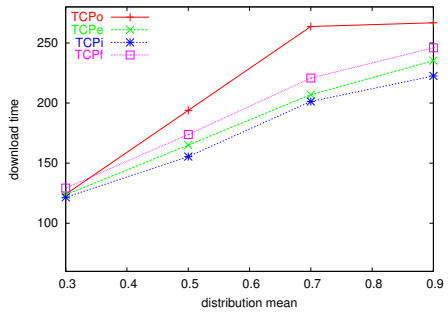
As the bulk data transferred is the same, there is little change in the number of transmitted packets. As expected, the download time should be doubled or quadrupled as the number of connections do. However, comparing the corresponding time values across cases with different connections, we find that the rate of increase in download time is less than that in the number of connections. As more connections are introduced, the throughput of the connection between S_1 and M_1 (the one we are monitoring) decreases, so the average RTT collected on the connection increases. The delay jitters due to LLR are less significant than the handover delays in the previous section. So all TCP senders become more robust against such less-variable delays as RTTs increase in multiple-connection cases. Compared with the single connection scenario, the connection being traced may avoid initiating some spurious timeouts when facing the same delay variations, and thus its performance can slightly improve. That is why the download time is not a direct double or quradruple increase here. The handover delays are more significant to the connection's average RTTs, so the slightly-increased RTTs have no effect in the multiple connection cases in the previous section.



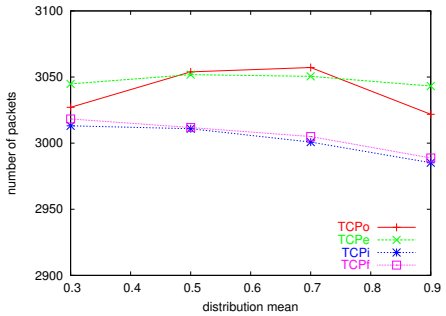
(a) Time for Reno with bugfix



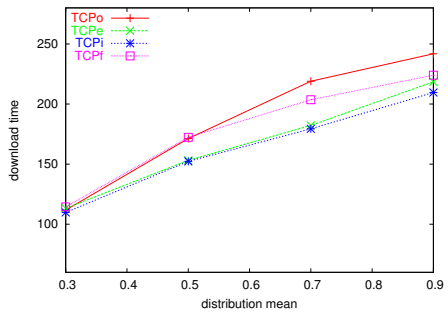
(b) Packets for Reno with bugfix



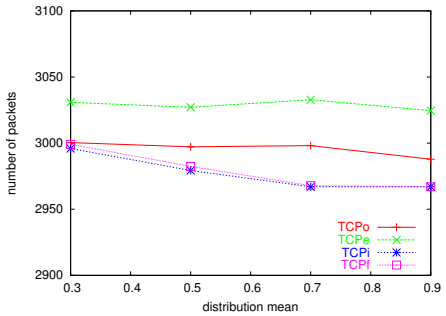
(c) Time for Reno without bugfix



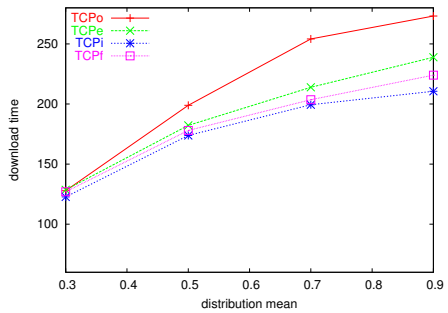
(d) Packets for Reno without bugfix



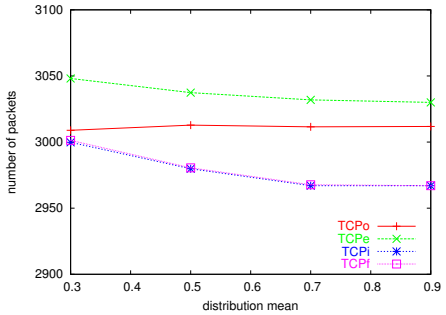
(e) Time for Newreno with bugfix



(f) Packets for Newreno with bugfix

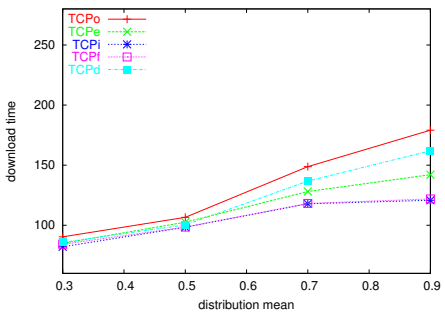


(g) Time for Newreno without bugfix

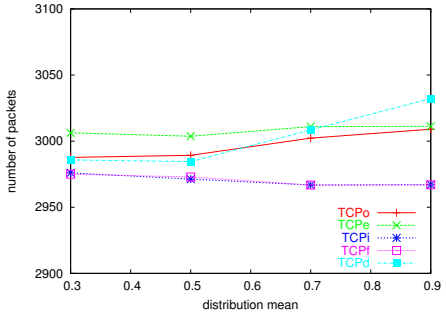


(h) Packets for Newreno without bugfix

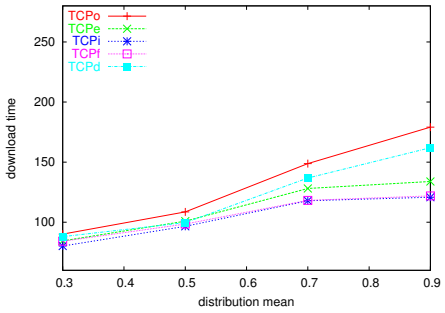
Figure 6.12: TCP Reno and Newreno in UMTS with LLR



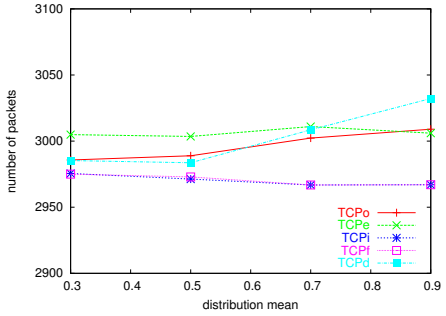
(a) Time for Sack with bugfix



(b) Packets for Sack with bugfix

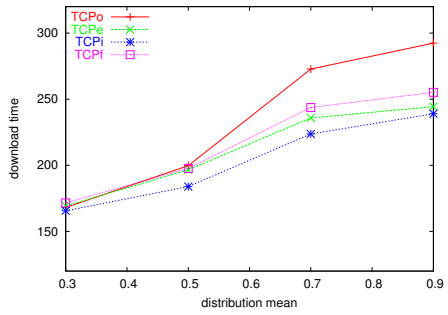


(c) Time for Sack without bugfix

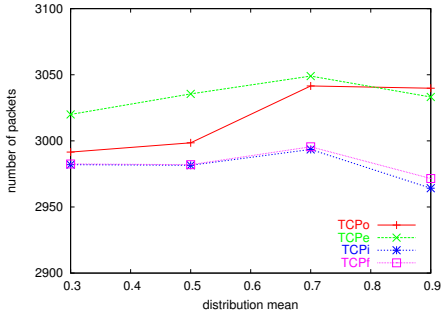


(d) Packets for Sack without bugfix

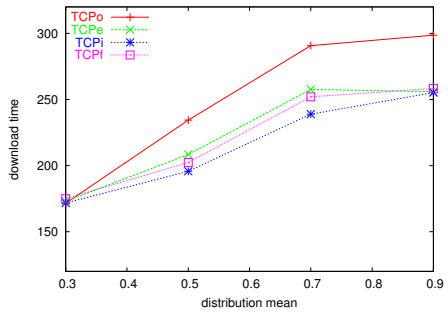
Figure 6.13: TCP Sack in UMTS with LLR



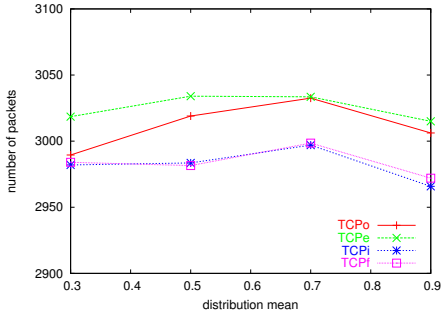
(a) Time for Reno with bugfix



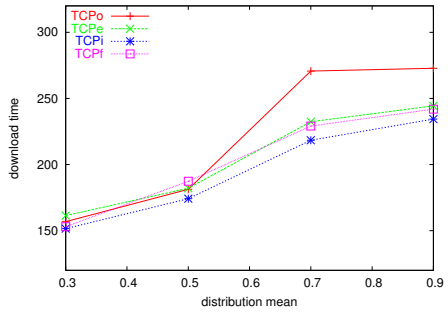
(b) Packets for Reno with bugfix



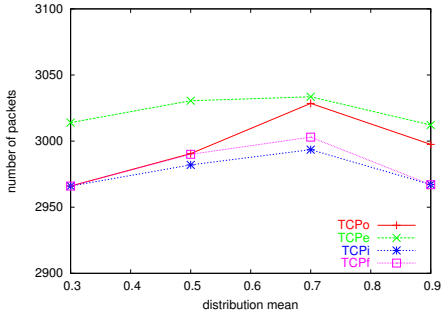
(c) Time for Reno without bugfix



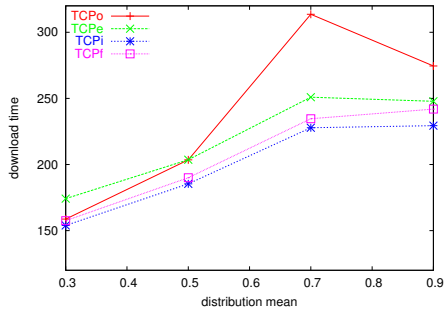
(d) Packets for Reno without bugfix



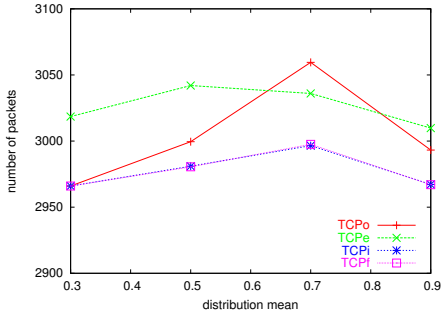
(e) Time for Newreno with bugfix



(f) Packets for Newreno with bugfix

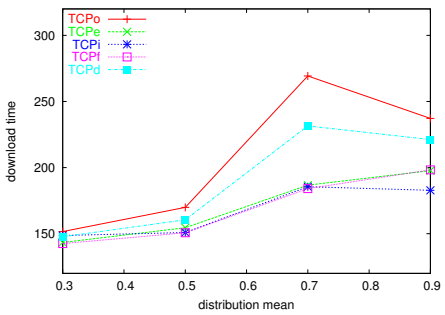


(g) Time for Newreno without bugfix

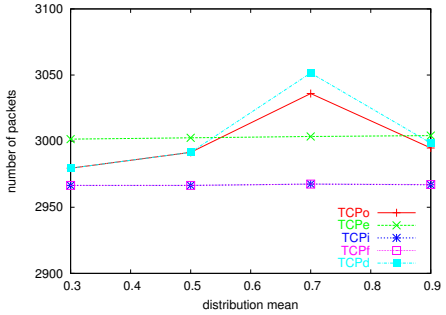


(h) Packets for Newreno without bugfix

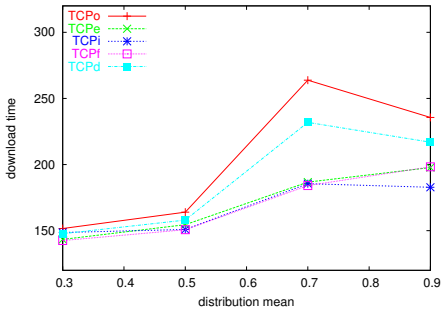
Figure 6.14: TCP Reno and Newreno in UMTS with LLR – two connections



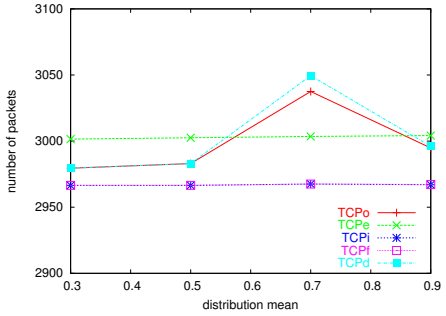
(a) Time for Sack with bugfix



(b) Packets for Sack with bugfix

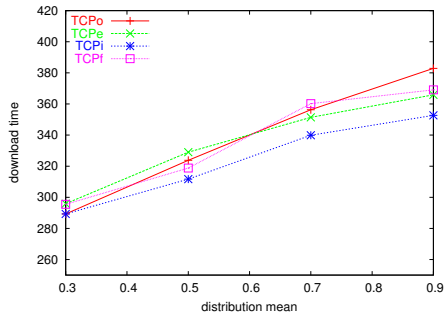


(c) Time for Sack without bugfix

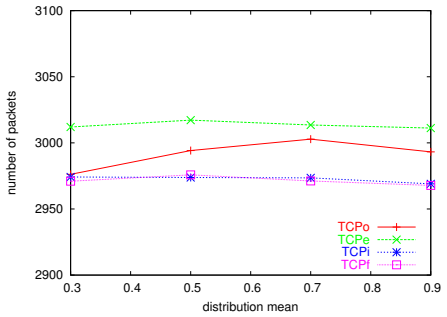


(d) Packets for Sack without bugfix

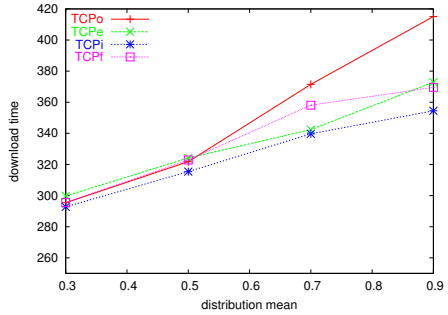
Figure 6.15: TCP Sack in UMTS with LLR – two connections



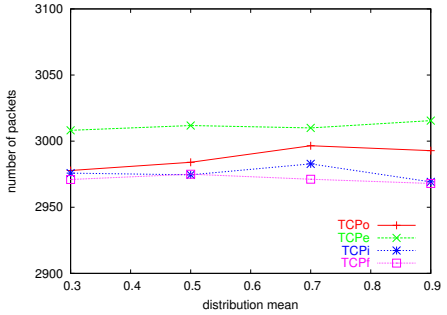
(a) Time for Reno with bugfix



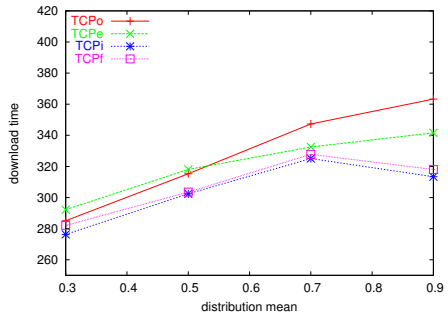
(b) Packets for Reno with bugfix



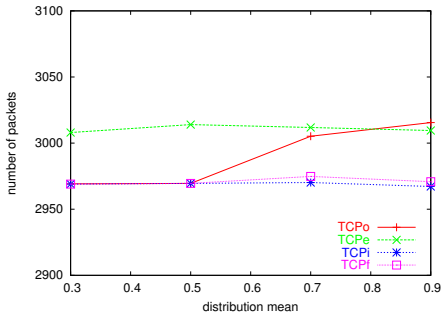
(c) Time for Reno without bugfix



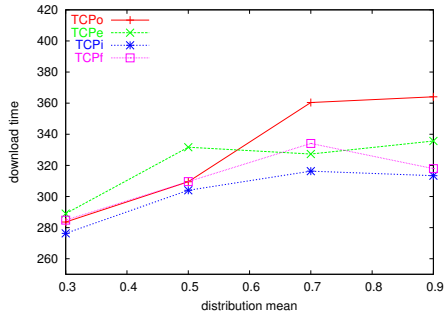
(d) Packets for Reno without bugfix



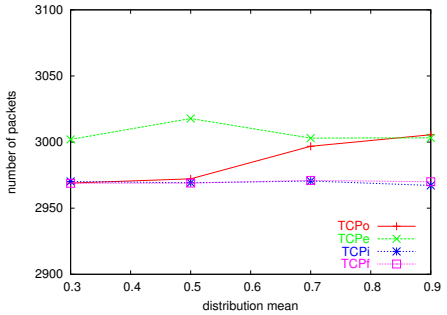
(e) Time for Newreno with bugfix



(f) Packets for Newreno with bugfix

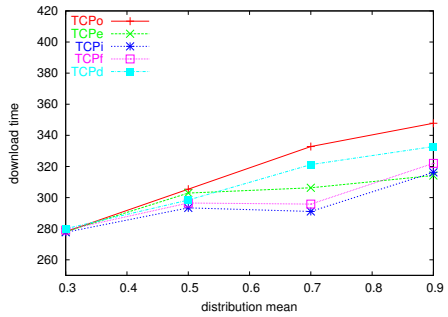


(g) Time for Newreno without bugfix

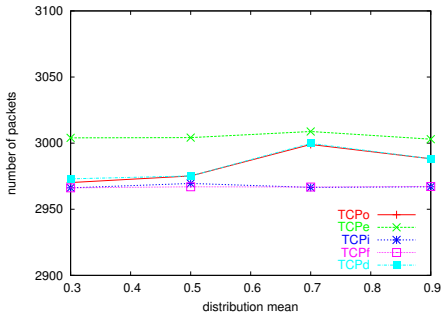


(h) Packets for Newreno without bugfix

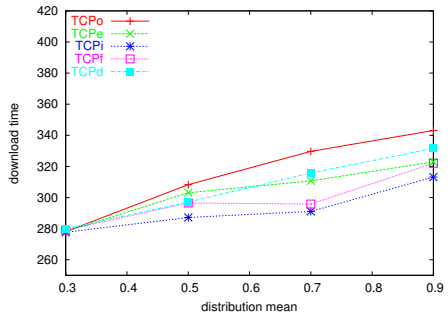
Figure 6.16: TCP Reno and Newreno in UMTS with LLR – four connections



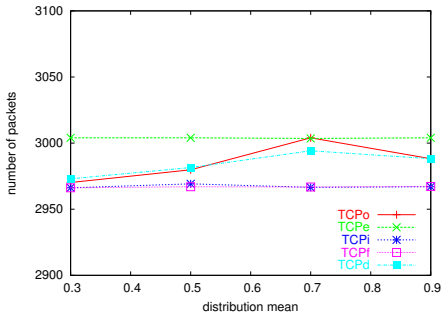
(a) Time for Sack with bugfix



(b) Packets for Sack with bugfix



(c) Time for Sack without bugfix



(d) Packets for Sack without bugfix

Figure 6.17: TCP Sack in UMTS with LLR – four connections

6.4 Summary of Results

In this chapter, we have evaluated Eifel-I and other approaches in the presence of delay spikes due to handovers. As we have used a large maximum window size, for all TCP senders, their slow-start transmission gets terminated by multiple packet losses in a single window due to congestion. This is because the congestion window overloads the bottleneck buffer after a certain amount of time. After that point, the senders enter the stationary congestion avoidance state with slow increase and periodic decrease of *cwnd* caused by a single congestion loss. Approaches like Eifel and F-RTO are only aimed at improving TCP's robustness against spurious retransmission, so when encountering such multiple losses they usually go through a long waiting period for a timeout retransmission and the degradation caused by this often exceeds the improvement gained from avoiding spurious timeouts. Eifel's aggressive timer adaptation can trigger retransmission timeout prematurely. When packet losses also occur in the same transmission window, this can lead to multiple timeouts. Eifel-I is able to recover multiple packet losses efficiently by using fast retransmit and recovery. So it performs consistently better than the others, including TCP with/without Eifel, F-RTO and DSACK. With a TCP flavor containing more advanced loss recovery, the inefficiency in Eifel and F-RTO's loss recovery may lessen. But Eifel-I can still perform better than the others because its stable RTT sampling from both original packets and retransmits. During the stationary congestion avoidance phrase, *cwnd* is halved periodically by a single congestion loss. When bugfix is enabled, TCP with/without Eifel, F-RTO or DSACK can only wait for a lengthy timeout to retransmit the lost packet. Again, Eifel-I can recover the loss much earlier through fast retransmit and fast recovery in such cases. Moreover, the performance degradation introduced by a fast retransmit is much less than the degradation that follows a timeout. With its enhanced loss recovery, Eifel-I also allows TCP more efficient congestion control restoration after a spurious timeout. Compared with the others, Eifel-I delivers robust performance improvement across different buffer sizes. This property is very important in a varying delay wireless environment, since it is difficult to size the system with an optimal buffer size, given that the capacity of the link also varies

over time.

We have then evaluated the different approaches with varying levels of link layer retransmission. For reasons similar to those that we have discussed, Eifel-I performs consistently better than or at least the same as the other approaches in all cases.

For the two scenarios of handover and link layer retransmission, we have also experimented the cases with multiple concurrent connections. These cases show a similar changing pattern and relative improvement among the different approaches. An interesting phenomenon is that after a timeout, F-RTO responds to the first non-duplicate ACK with two new packets. When multiple connections are competing for the wireless link concurrently, this aggressive transmission may multiply and lead to congestion losses.

In summary, we conclude that in situations like wireless networks where packet losses and variable delays frequently occur or co-occur, Eifel-I delivers consistently good performance because it is capable of efficiently coping with both variable delays and packet losses. In all the scenarios we have experimented in (regardless of the TCP flavor used, or the number of concurrent connections, etc.), TCPi is always better than or at least the same as the other approaches. In certain cases, it achieves up to 40% improvement over the original TCP, and more than 20% improvement over the approaches like DSACK, Eifel and F-RTO.

Chapter 7

Conclusion and Future Work

7.1 Summary

Spurious retransmissions, especially spurious timeouts, have been identified as one of the main causes for the performance degradation of TCP over wireless links. As wireless networks become increasingly popular, improving TCP's ability in solving the problems caused by spurious retransmissions is also becoming a more urgent task. Several algorithms have been proposed for making TCP more robust against spurious retransmissions, such as DSACK, Eifel and F-RTO. They all have taken the way of first detecting a spurious retransmission and then recovering by undoing the unnecessary transmission rate reduction. Some may even be able to actively adapt the TCP sender after a spurious retransmission so that future retransmissions can be avoided. All of them can improve TCP performance to some extent. However, each of them also suffers from some weaknesses, preventing them from getting more improvement in their performance. Another main cause of TCP's bad performance over wireless links is packet losses due to handovers or mobile management.

After carefully studying the weaknesses of the existing approaches, we have proposed a new approach, Eifel-I, for handling both spurious timeouts and congestion losses. In Chapter 1, the objectives in designing this new approach were laid down as:

- First, Eifel currently suffers heavily from the use of timestamps as the extra information, so we need to find a piece of extra information that would introduce as little overhead as possible.

- Second, the new approach should retain the strengths of the current approach, such as its early detection and its robustness against ACK losses.

- Third, the new approach should enable the use of the current header compression schemes which have proved to be useful over low-speed links.

- Fourth, [3] pointed out that the current standard TCP retransmission timer defined in RFC2988 [39] adapts fairly slow to changes in network conditions. This is because retransmits are not allowed by Karn's algorithm [30] to be used in RTT sampling. With the use of timestamps, the current Eifel approach solves this slow adaptation problem, and provides the possibility for a better RTO estimator for avoiding future spurious timeouts. Our new approach should also try to retain this property.

- Fifth, if possible, our approach should cover the packet loss problem as well.

Eifel-I is based on the same idea as Eifel. It preserves the nice properties of the current Eifel approach: it eliminates the retransmission ambiguity; it detects spurious timeouts upon the first acceptable ACK that arrives during loss recovery, so avoiding the go-back-N retransmission and restoring the congestion control state; and it is also quite robust against ACK losses.

As it has its own way of the implementation, Eifel-I also introduces some new properties of its own, allowing it better performance:

First, Eifel-I's selective use of timestamps avoids the 12-byte overhead most of the time as retransmits only form a relatively small part of the total transmitted packets. The saved space can be used by other TCP options. For example, if the space is used by SACK [35], the TCP sender may avoid unnecessarily retransmitting a series of packets.

Removing timestamps in most of the packets also enables the use of current TCP/IP header compression schemes [14] [28]. As these compression schemes can compress

a 40-byte TCP/IP header to just 3 to 5 bytes, they can greatly reduce the total protocol overhead incurred by each packet. This is especially useful for slow wireless links. Over such links, the improvement of the Eifel algorithm comes mainly from avoiding some unnecessary data delivery.

In adapting the TCP retransmission timer, Eifel-I can respond in time to changing network conditions and maintain the adapted RTO value at a reasonable level for an amount of time. It is therefore more superior than other existing approaches like DSACK, Eifel and F-RTO in avoiding future spurious timeouts.

With Eifel-I, a TCP sender can easily distinguish ACKs for original transmits from ACKs for retransmits. In conjunction with this capability, we have also worked out a new method that can greatly improve the ability of non-Sack TCP (e.g., TCP Reno, NewReno, etc.) in recovering from multiple packet losses. It enables the TCP sender to avoid unnecessary fast retransmits if the DUPACKs are triggered by duplicate packets, and to efficiently recover lost packets through fast retransmit and fast recovery instead of waiting for a timeout.

In Chapter 6, we have used simulation to evaluate Eifel-I against the original TCP and other approaches, such as DSACK, Eifel and F-RTO. We have provided extensive experiment results and discussed in detail Eifel-I's improvements in various circumstances. From the results, we have found that in environments like wireless networks where packet losses and variable delays frequently occur or co-occur, Eifel-I can deliver consistent performance improvement over other approaches because it is capable of efficiently handling both variable delays and packet losses. In all the scenarios that we have experimented in (regardless of the TCP flavor used, or the number of concurrent connections, etc.), Eifel-I always performs better than or at least the same as the other approaches. In certain cases, it achieves up to 40% improvement over the original TCP, and more than 20% improvement over approaches like DSACK, Eifel and F-RTO.

7.2 Future Work

Some other problems of the current TCP retransmission timer have been identified in [34]. We plan to evaluate the possible ways to fix them by further extending our timer adaptation.

In this thesis, we have mainly focused on dealing with spurious timeouts. In fact, Eifel-I is also able to detect spurious fast retransmits. As packet reordering is disallowed in current 2.5G/3G wireless networks, spurious fast retransmit is not a main concern now. However, allowing out-of-order delivery can reap benefits like low delay time for non-reliable real-time traffic. With more measurement data to guide the modeling of reordering in wireless links, we can devote more effort in verifying Eifel-I's ability in handling spurious fast retransmits over wireless links.

In addition to spurious timeouts, wireless characteristics like bandwidth oscillation and on-demand resource allocation may have some other effects on TCP. We plan to study these characteristics, and verify Eifel-I's behavior in the presence of them.

So far, we have verified Eifel-I by implementing it in the NS-2 network simulator and running experiments by simulating the behavior of the wireless network. For more realistic testings, we intend to implement Eifel-I in real TCP implementations in the future.

Bibliography

[1] M. Allman, H. Balakrishnan, and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit", RFC3042, January 2001.

[2] M. Allman, "A Web Server's View of the Transport Layer", *ACM Computer Communication Review*, 30(5), October 2000.

[3] M. Allman, V. Paxson, "On Estimating End-to-End Network Path Properties", In *Proceedings of ACM SIGCOMM 99*, September 1999.

[4] M. Allman, V. Paxson, and W. R. Stevens, "TCP Congestion Control", RFC2581, April 1999.

[5] H. Balakrishnan, et al, "TCP Performance Implications of Network Asymmetry", RFC3449, December 2002.

[6] H. Balakrishnan, et al, "Improving TCP/IP Performance over Wireless Networks", In *Proceedings of ACM Mobicom*, November 1995.

[7] E. Blanton, M. Allman, "On Making TCP More Robust to Packet Reordering", ACK Computer Communication Review, 32(1), January 2002.

[8] E. Blanton, M. Allman, "Using TCP DSACKs and SCTP Duplicate TSNs to Detect Spurious Retransmissions", Internet Draft, draft-ietf-tsvwg-dsack-use-00.txt, work in progress, June, 2003.

[9] A. Bakre, B.R. Badrinath "Handoff and System Support for Indirect TCP/IP", In *Proceedings of Second Usenix Symposium on Mobile and Location-Independent Computing*, April 1995.

[10] N. Brownlee, K.C. Claffy, "Understanding Internet Traffic Streams: Dragonflies and Tortoises", IEEE Communications Magazine, October 2002.

[11] C. Bettstetter, "GSM Phase 2+ General Packet Radio Service GPRS: Architecture, Protocols, and Air Interface, IEEE Communication Surveys, vol. 2, n.3, 1999.

[12] J. Bennett, C. Partridge, N. Shectman, "Packet Reordering is Not Pathological Network Behavior", IEEE/ACM Transactions on Networking, Vol. 7, No. 6, December 1999, p789

[13] M.C. Chan, R. Ramjee "TCP/IP Performance over 3G Wireless Links with Rate and Delay Variation", In *Proceedings of ACM Mobicome'02*, September 2002.

[14] M. Degermark, B. Nordgren, and S. Pink, "IP Header Compression", RFC2507, February 1999.

[15] S. Dixit, R. Prasad, "Wireless IP and Building the Mobile Internet", Artech House, 2003.

[16] W. Eddy, et al, "New Techniques for Making Transport Protocols Robust to Corruption-based Loss", Under Submission, January 2004.

[17] S. Floyd, T. Henderson, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC2582, April 1999.

[18] S. Floyd, et al, "An Extension to the Selective Acknowledgment (SACK) Option for TCP", RFC2883, July 2000.

[19] S. Floyd, "TCP and successive fast retransmits", technical report, Oct 1994.

[20] A. Gurtov, S. Floyd, "Resolving Acknowledgment Ambiguity in non-SACK TCP", In *Proceedings of the Next Generation Teletraffic and Wired/Wireless Advanced Networking* (NEW2AN'04), St.Petersburg, Russia, February 2004.

[21] A. Gurtov, S. Floyd, "Modeling Wireless Links for Transport Protocols", to appear in ACM Computer Communication Review (ACM CCR), November 2003.

[22] A. Gurtov, R. Ludwig, "Evaluating the Eifel Algorithm for TCP in a GPRS Network", In *Proceedings of European Wireless*, Florence, Italy, February 2002.

[23] A. Gurtov, R. Ludwig, "Responding to Spurious Timeouts in TCP", *IEEE Infocom'03*, California, U.S.A., March 2003.

[24] A. Gurtov, et al, "Multi-Layer Protocol Tracing in a GPRS Network", In *Proceedings of IEEE Vehicular Technology Conference (VTC'02)*, Birmingham, Alabama, September 2002.

[25] A. Gurtov, "The Eifel Implementation in NS-2", http://www.cs.helsinki.fi/u/gurtov/ns/.

[26] International Engineering Consortium (IEC): Universal Mobile Telecommunications System (UMTS) Protocols and Protocol Testing, http://www.iec.org.

[27] H. Inamura et al, "TCP over Second (2.5G) and Third (3G) Generation Wireless Networks", RFC3481, February 2003.

[28] V. Jacobson, "Compressing TCP/IP Headers for Low-Speed Serial Links", RFC1144, February 1990.

[29] V. Jacobson, R. Braden, and D. Borman, "TCP Extensions for High Performance", RFC1323, May 1992.

[30] P. Karn, C. Partridge, "Improving Round-Trip Time Estimates in Reliable Transport Protocols", In *Proceedings of ACM SIGCOMM 87*, August 1987.

[31] R. Ludwig, A. Gurtov, "The Eifel Response Algorithm for TCP", Internet draft, draft-ietf-tsvwg-tcp-eifel-response-03.txt, work in progress, March 2003.

[32] R. Ludwig, R. H. Katz, "The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions", *ACM Computer Communication Review*, 30(1), January 2000.

[33] R. Ludwig, M. Meyer, "The Eifel Detection Algorithm for TCP", RFC3522, April 2003.

[34] R. Ludwig, K. Sklower, "The Eifel Retransmission Timer", *ACM Computer Communication Review*, 30(3), July 2000.

[35] M. Mathis, et al, "TCP Selective Acknowledgment Options", RFC2018, October 1996.

[36] M. Methfessel, et al., "Vertical Optimization of Data Transmission for Mobile Wireless Terminals", IEEE Wireless Communication Magazine, December 2002.

[37] The Network Simulator - NS-2, available at: http://www.isi.edu/nsnam/ns/

[38] B. Patil, et al, "IP in Wireless Networks", Prentice Hall, 2003.

[39] V. Paxson, M. Allman, "Computing TCP's Retransmission Timer", RFC2988, November 2000.

[40] V. Paxson, "End-to-End Internet Packet Dynamics", Proceedings of the ACM SIGCOMM '97, September 1997.

[41] J. Postel, Transmission Control Protocol, RFC793, September 1981.

[42] M. Schlager, "The *Hiccup* Implementation in NS-2", http://www-tkn.ee.tu-berlin.de/ morten/eifel/ns-eifel.html

[43] K. N. Srijith, L. Jacob, and A. L. Ananda, "Worst-case Performance Limitation of TCP SACK and a Feasible Solution", In *Proceedings of 8th IEEE International Conference on Communications Systems (ICCS)*, Singapore, 25-28 November 2002, pp. 1157-116.

[44] P. Sarolahti, M. Kojo, "F-RTO: An Algorithm for Detecting Spurious Retransmission Timeouts with TCP and SCTP", Internet Draft, draft-sarolahti-tsvwg-tcp-frto-04.txt, work in progress, June 2003.

[45] P. Sarolahti, M. Kojo, K. Raatikainen, "F-RTO: A New Recovery Algorithm for TCP Retransmission Timeouts", University of Helsinki, Department of Computer Science, Series of Publications C, No. C-2002-07, February 2002.

[46] W. R. Stevens, "TCP/IP Illustrated, Volume 1 (The Protocols)", Addison Wesley, November 1994.

[47] M. Taferner and E. Bonek, "Wireless Internet Access over GSM and UMTS", Springer, 2002.

[48] F. Vacirca, et al,"Investigating Interactios between ARQ Mechanisms and TCP over Wireless Environments", In *Proceedings of IEEE Global Telecommunications Conference*, (Globecom'03), San Francisco USA, 2003.

[49] B. Walke, "Mobile Radio Networks: Networking, Protocols and Traffic Performance", 2nd Ed, Wiley & Sons, 2001.

[50] M. Yavuz, F. Khafizov "TCP over Wireless Links with Variable Bandwidth", In *Proceedings of IEEE Vehicular Technology Conference* (VTC'02 Fall), September 2002.

[51] M. Zhang, et al, "Improving TCP's Performance under Reordering With DSACK", ICSI Technical Report TR-02-006, July 2002.

Appendix A. Cellular Wireless Systems

A.1 Cellular Wireless Fundamentals

In this section, we present some background information on mechanisms or techniques that have commonly been deployed by various cellular wireless links.

A.1.1 Multiple Access

"Any scarce resource that is to be used simultaneously by more than one user needs to be divided into subportions in order to prevent interference in each user's usage of that resource" [38].

In cellular mobile radio systems, that resource is the *radio transmission media*. Lack of radio resource to support all users with the required bandwidths has for years been the main problem of wireless links. In order to allow users to access the same radio transmission medium simultaneously, the radio spectrum is divided into channels. This simultaneous use of channels is called *multiple access*. The physical medium, i.e., the radio spectrum, can be divided into individual channels based on a set of criteria. These criteria depend on the technology used to make the distinction between channels.

The three primary technologies used in cellular wireless systems are:

- **Frequency Division Multiple Access (FDMA).** In FDMA, the radio spectrum is divided into several frequency bands with a guard channel in between to prevent interference. Each channel is a specific frequency, and each user is assigned a channel for the duration of a call. However, since the channel may be in use during the whole duration, this exclusive allocation may result in poor resource

utilization.

- **Time Division Multiple Access (TDMA).** In TDMA, the channel is a time slot on a specific frequency. Periodically, a frequency channel is allocated alternately to different logical channnels, each of which is occupied by a different user.

- **Code Division Multiple Access (CDMA).** In CDMA, the channel is a unique code, and each user is assigned a different code. The code is typically pseudo-random in nature, which processes favourable correlation properties to ensure that different physical channels are not confused with one another. Data packets intended for a specific channel is modulated with that channel's code.

A.1.2 Error Protection in Radio Channels

Due to reasons such as distance, speed and the radio signal shadowing of mobile stations communicating with each other, environmental conditions, electromagnetic interference, etc., the bit error ratio (BER) [1] in mobile radio communication can fluctuate considerably over time. To help cope with the various error causes and provide a more reliable communication medium, a number of error protection and error control methods have been utilized for quite some time to detect and correct errors during the transmission of a data signal.

Generally speaking, there are two categories of error control: *error detection* and *error correction*. Error correction again comprises two categories: *backward error correction* (BEC) and *forward error correction* (FEC) [49]. In any case, an error control method involves, unavoidably, the use of the available bandwidth to carry the extra redundant bits for detecting and correcting errors.

Error Detection

Error detection focuses on the detection of errors only. Through error detection, it is possible to establish whether a received packet is valid or not. However, data packets

[1] BER is the ratio of error bits to the total number of bits transmitted

that are recognized as being incorrect cannot be corrected. This is due to the fact that error detection uses a much fewer number of redundant bits than error correction codes.

Forward Error Correction (FEC)

FEC coding can improve data reliability by introducing a redundant data pattern prior to the transmission of a data packet. As the sender adds enough redundancy to a packet, the receiver is able to detect and correct a certain number of errors based on the redundancy pattern. This method is more suitable for applications dealing with real time information interchange or time sensitive communications. In contrast to the ARQ methods described in the next sub-section, a reverse channel from the receiver to the sender is not needed.

There are two code families that are suitable for forward error correction [49]:

- The linear block codes which are systematic codes, i.e., a certain number of redundant bits are calculated from the packet to be protected and are transmitted with it. The coded packet can thus be divided into a redundant and a non-redundant part.
- The convolutional codes which are non-systematic codes. In comparision with block codes, convolutional codes have a memory, i.e., a bit in a packet which is not only dependent on the actual data bit but on several preceding data bits.

Convolutional codes are very suitable for the correction of uncorrelated errors, but are extremely sensitive when it comes to bursty distributed errors, which frequently occur in radio channels. Consequently, in mobile radio systems, convolutional codes are almost exclusively used in combination with *interleaving* [47], a technique that is able to let bursty errors appear as single-bit errors.

Error Handling By ARQ Protocols

Backward error correction is also known as *Automatic Repeat reQuest* (ARQ) [49]. Unlike FEC, ARQ methods do not attempt to correct the corrupted packets at the receiver and accept them. Instead, with the combination of an error detection method, they simply check for the correctness of an incoming packet, and if the packet is not recognized

as correct, the receiver discards the packet and requests the sender to transmit the packet again. This ARQ process is closely related to the flow control mechanism. Flow control limits the amount of information being sent so the receiver is not overwhelmed with data, to avoid the loss of frames due to an inability to process incoming data.

In order for the receiver to ask the sender for retransmission, a reverse channel between the two ends is needed. The result of the error evaluation of each packet transmitted back to the sender is call an *acknowledgment*. Data acknowledgment helps the flow control process and provides the mechanism for indicating the status of the received data. A positive acknowledgment (ACK) is sent when the packet received is correct; if it is defective, then a negative acknowledgment is sent along with a retransmission request for the defected data.

Three kinds of ARQ techniques can be used [49]:

- *Send-and-Wait ARQ Protocol* The sender transmits a packet and then waits for an ACK from the receiver. During this period, no other packets can be transmitted until a positive ACK is returned. If an error is encountered, the receiver will request retransmission of the packet by issuing a NACK. As the method can only deal with one packet at a time, the throughput is minimal, especially for links with long propagation delays and with short data packets. So, this method is not commonly used.

- *Go-Back-N ARQ Protocol* This ARQ method is also called the explicit reject (REJ) or cumulative ARQ. With this method, the sender can transmit a series of packets up to the window size controlled by flow control mechanisms. If an error is encountered at the receiver, it will explicitly reject (REJ) that packet and all successive ones, and then ask the sender to resend the packets again sequentially, starting from the packet that was damaged. This method requires sufficient buffer at the sender for storing all the unacknowledged packets, and it also wastes some bandwidth because of the retransmission of packets which were not originally damaged but yet discarded.

- *Selective-Reject ARQ Protocol* As with REJ, the objective of the SREJ method is to transmit packets between senders and receivers as continuously as possible. With this method, if an error occurs in one packet in a sent series, the technique can selectively point out which packet needs to be retransmitted by issuing an SREJ. At the same time, the receiver needs to store all the packets after the damaged one until the retransmit of that packet is correctly received. The drawback to this method is that it requires an even larger buffer in order to sort and retransmit packets out of order.

A.2 Some Details on GPRS

A.2.1 Logical Packet Data Channels

GPRS defines a number of packet data logical channels that can be mapped onto the GPRS physical channels.

Packet Common Control Channel (PCCCH) comprises logical channels for common control signalling used for data packet:

- *PRACH* is used by MS to initiate uplink transfer for sending data, or signaling information like paging responses.
- *PPCH* is used to page an MS prior to downlink packet transfer.
- *PAGCH* is used to inform an MS about the assignment of dedicated uplink or downlink resources.
- *PNCH* is used for point-to-multi-point multicast (PTM-M) service. It is used to send a PTM-M notification to a group of MSs prior to a PTM-M packet transfer.

Packet Broadcast Control Channel (PBCCH) broadcasts packet data specific system information about a cell to all the GPRS-enabled MSs that are currently in that cell.

Packet Data Traffic Channel (PDTCH) is the bearer channel allocated for packet data transfer. As illustrated in Fig. 2.2, this kind of channel is assigned separately

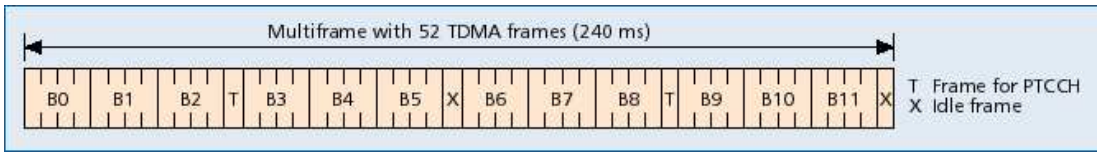


Figure A-1: Multiframe structure with 52 TDMA frames

for uplink and downlink directions. In multislot operations, one MS may use multiple PDTCHs in parallel for individual packet transfer.

Packet Dedicated Control Channel (PDCCH) is a control channel dedicated to a particular MS. There are two types of channels under PDCCH:

- *PACCH* is used to convey signalling information related to a given MS, such as ACK, power control information, etc. It can also carry resource assignment and reassignment messages. A PACCH is associated with one or more PDTCHs currently assigned to an MS and shares resources with these PDTCHs.
- *PTACH* is used to synchronize the timing advance of an MS for uplink data transmission.

A.2.2 Multiframe Structure

The GPRS multiframe structure is illustrated in (Fig. A-1 [11]), framing 52 TSs into one multiframe. Each 52-multiframe represents one physical GPRS channel consisting of 12 radio blocks and one idle block, each block comprising four radio bursts distributed on TSs with the same TS number in consecutive TDMA frames.

In Fig. A-1, each of the 52 vertical slices is one TDMA frame, so there exist altogether 8 multiframes for each of the 8 TSs in a TDMA frame. Hence, in each multiframe, the 52 timeslots are not consecutive in time but rather every n_{th} TS (n = 0, 1, ..., 7) in a TDMA frame.

The mapping of the various logical channels onto the radio blocks B0 – B11 of each multiframe (i.e., each physical channel) can vary from block to block and is controlled by parameters that are broadcasted on PBCCH.

Precedence level	Identifier	To be served
1	High priority	preferably before levels 2 and 3
2	Normal priority	preferably before level 3
3	Low priority	without preference

Table A-1: Precedence levels

Delay class	128 byte packet		1024 byte packet	
	Mean delay (s)	95% (s)	Mean delay (s)	95% (s)
1 (predictive)	0.5	1.5	2	7
2 (predictive)	5	25	15	75
3 (predictive)	50	250	75	375
4 (best effort)	unspecified			

Table A-2: GPRS delay classes

A.2.3 QoS Parameters

A QoS profile defines the QoS within the range of the following service parameters [49] [47]:

Precedence

Under normal circumstances the network should try to meet all profiles' QoS agreements. The precedence specifies the relative importance to keep the conditions even under critical circumstances, e.g., momentarily high network load. The various precedence levels are presented in Table A-1 [49].

Delay

The delay is defined as the end-to-end transfer time between two communicating MSs or between an MS and the Gi interface at the GGSN. This includes all delays within the GPRS network, e.g., the delay for request and assignment of radio resources and the transit delay in the GPRS backbone network. Transfer delays outside the GPRS network, e.g., in external transmit networks, are not taken into account.

Four delay classes are defined, each of which specifies the maximum allowed mean delay and 95% delay allowed by the transfer of data through the GPRS network. The 95% delay is the maximum delay allowed in 95% of all data transfers. The network operator has to provide for convenient resources on the air interface to be able to serve

Reliability class	Probability for packet loss	duplicate packet	out of sequence	corrupt data
1	10^{-9}	10^{-9}	10^{-9}	10^{-9}
2	10^{-4}	10^{-5}	10^{-5}	10^{-6}
3	10^{-2}	10^{-5}	10^{-5}	10^{-2}

Table A-3: GPRS reliability classes in terms of residual error rates

the number of participants with a certain delay class expected within each cell [49]. Although there is no need for all delay classes to be available, at least *best effort* has to be offered.

Reliability

Data services generally require a low residual bit error rate (BER). Erroneous data is usually useless, while incorrectly received speech only leads to a worse perception. Reliability of data transmission is defined within the scope of the following cases:

- probability of loss of data
- probability of out-of-sequence data delivery
- probability of multiple delivery of data
- probability of erroneous data

The reliability classes are listed in Table A-3 [47].

The negotiated reliability classes subsequently specifies the requirements for each layer's services. As presented in Table A-4, the combination of different modes of operation of the GPRS specific protocols GTP, LLC and RLC, explained in Section 2.1.3, support the reliability requirements of various applications, e.g., Real-Time (RT) or Non Real-Time (NRT).

Peak and Mean Throughput

User data throughput in a GPRS network is specified within the scope of a set of throughput classes that characterize the expected bandwidth for a requested PDP context. It is defined by the choice of *peak* and *mean* throughput classes. Both are measured at the Gi

Reliability class	GTP mode	LLC packet mode	LLC data mode	RLC block mode	Traffic type security
1	ACK	ACK	PR	ACK	NRT traffic, error sensitive, loss sensitive
2	UNACK	ACK	PR	ACK	NRT traffic, error sensitive, slightly loss sensitive
3	UNACK	UNACK	UNPR	ACK	NRT traffic, error sensitive, not loss sensitive
4	UNACK	UNACK	UNPR	UNACK	RT traffic, error sensitive not loss sensitive
5	UNACK	UNACK	UNPR	UNACK	RT traffic, not error sensitive, not loss sensitive

(UN)ACK (Un)acknowledged
(UN)PR Protected/Unprotected

Table A-4: GPRS reliability classes with the corresponding protocol mode combinations

and the radio interface. There is no guarantee that the peak throughput negotiated at the beginning is ever reached. The peak throughput classes are presented in Table A-5 [47]. Note that the peak throughput classes are defined up to 2Mbit/s, while the GPRS raido interface only supports a maximum of 171.2kbit/s user data rate. The higher classes are already defined for EDGE [2] [49] and the 3G UMTS system (see Section 2.2).

However, mean throughput specifies the average rate data transmitted for a connec-

[2] *Enhanced Data rates for GSM Evolution* (EDGE introduces a new modulation scheme at the GSM radio interface, which supports a maximum achievable bit rate of about 384kbit/s

Peak throughput classes	Peak throughput		
	[byte/s]		[kbit/s]
1	up to	1000	8
2	up to	2000	16
3	up to	4000	32
4	up to	8000	64
5	up to	16000	128
6	up to	32000	256
7	up to	64000	512
8	up to	128000	1024
9	up to	256000	2048

Table A-5: GPRS peak throughput classes

Mean throughput classes	Mean throughput	
	[byte/h]	≈ [bit/s]
1	100	0.22
2	200	0.44
3	500	1.11
4	1000	2.2
5	2000	4.4
6	5000	11.1
7	10000	22.
8	20000	44.
9	50000	111.
10	100000	220.
11	200000	440.
12	500000	1110.
13	1000000	2200.
14	2000000	4400.
15	5000000	11100.
16	10000000	2200.
17	20000000	4400.
18	50000000	11100.
31	Best effort	

Table A-6: GPRS mean throughput classes

tion, and it must be maintained during the lifetime of the connection. The mean throughput may be limited by the network even if more resources were available. if *best effort* has been agreed on as the throughput class, throughput is made available to an MS when ever there are resources needed and at disposal. Table A-6 summarizes the classes of mean throughput [49].

A.2.4 Mobility Management Scenarios

Before discussing the possible GPRS MM scenarios, we first consider an MS's state. Relating to GPRS MM, an MS can be in one of the following three states:

- **Ready.** When an MS is in this state, it can send or receive data, and it informs the SGSN every time it changes cells. A timer monitors Ready state and upon expiry, the MS is put in Standby state.

- **Standby.** A connected MS which is inactive is put in Standby state. In this state, the location of the MS is only known at the RA level.

- **Idle.** An MS is not traceable when it is in Idle state, e.g., the MS is on power-off mode. It needs to perform the attach procedure in order to be reachable.

Basically, there are three different types of MS location updates possible in GPRS MM. To illustrate the update procedures, here, we consider the scenario of an MS on the move while downloading a file from an Internet host. The network topology used in this section is shown is Fig. A-2. At this point, we assume that the MS has established an appropriate PDP Context and is currently in the cell covered by BTS1. So, as we can see, the current downlink transmission path is: starts from the Host (through Internet) to GGSN, then to SGSN1, BSC1 and BTS1, and finally arrives at the MS. As we will explain later, as the MS moves, the BTS, BSC and SGSN on the transmission path, which serve the MS, will also change dynamically to facilitate the location changes. However, the GGSN remains as an anchor point, which effectively hides the mobility of the MS.

Cell Change

As the name implies, this applies only to an MS which has moved to another cell in the same RA. As in Fig. A-2 [15], the MS is moving from BTS1 to BTS2 (arrow 1). At a certain time, it identifies that BTS2 can offer better communication quality, so it will camp on a channel controlled by BTS2. This procedure is known as a *mobile originated handover*, which happens as the MS suddenly switches frequency channels and camps on a new one. It is also known as *cell reselection*.

Referring Fig. A-2, since the new cell (BTS2) is still controlled by BSC1, both BSC1 and SGSN1 will not know the movement of the MS until the MS makes an uplink transmission. Then during the handover, all the downlink data packets will still be sent to BTS1. With unknowledged LLC, these packets will be lost; on the other hand, the packet transmitted in acknowledged LLC will be kept at SGSN1 as unknowledged and will be retransmitted when the MS informs SGSN1 about its location change.

After the handover, when the MS gets to know the new cell ID and RA identity from the broadcast control channel of the new cell, it will send out a special LLC frame in the uplink (arrow 2). Then the cell change of the MS is recorded by SGSN1 and all the

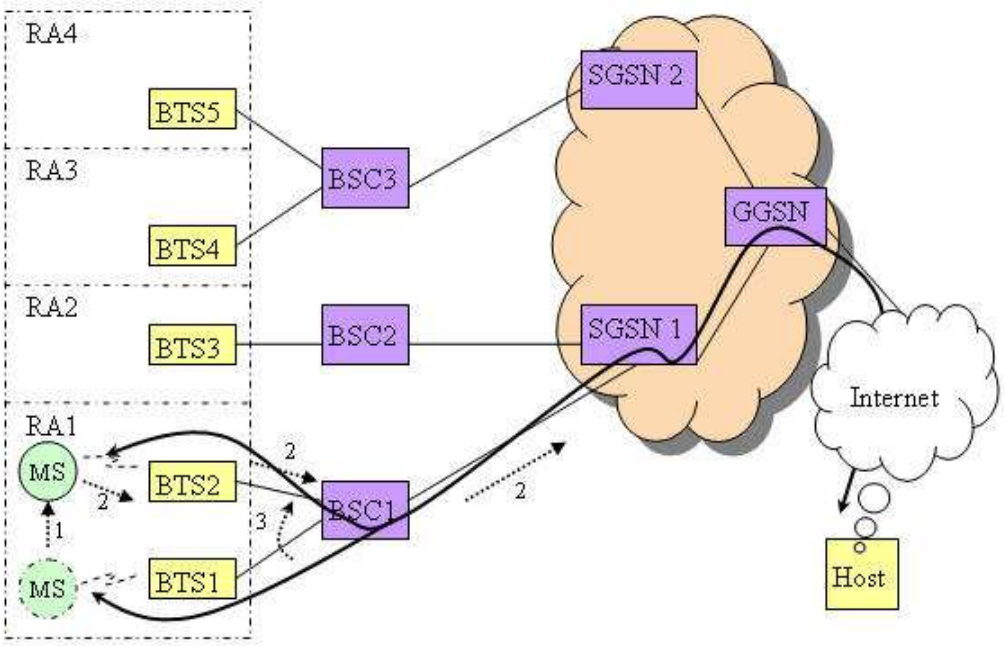


Figure A-2: Cell change – new cell in the same routing area

subsequent packets are forwarded to BTS2 (arrow 3).

Intra-SGSN RA Change

As the MS moves on (see Fig. A-3 [15]), it makes another handover, from BTS2 to BTS3 (arrow 1). In addition to a cell change, the routing area of the MS changes too. When noticing about the RA change, the MS will send an *RA Update (RAU) Request* message (arrow 2) to SGSN1. Since the new RA is still attached to the same SGSN (SGSN1), the request is treated merely as a cell change and all the subsequent packets will be routed to BSC2 and further to BTS3.

Inter-SGSN RA Change

When the MS further performs a handover from BTS3 to BTS4 (arrow 1) as in Fig. A-4 [15], it again sends out an *RAU Request* message (arrow 2). When SGSN2 receives the message, it informs SGSN1 about the movement of the MS (arrow 3-5) and SGSN1 stops transferring packets to the MS (3a). In acknowledged LLC, SGSN1 buffers all the unknowledged packets. So it will forward these packets to SGSN2 (arrow 5a). New LLC connections will be established between the MS and SGSN2. SGSN2 will also inform

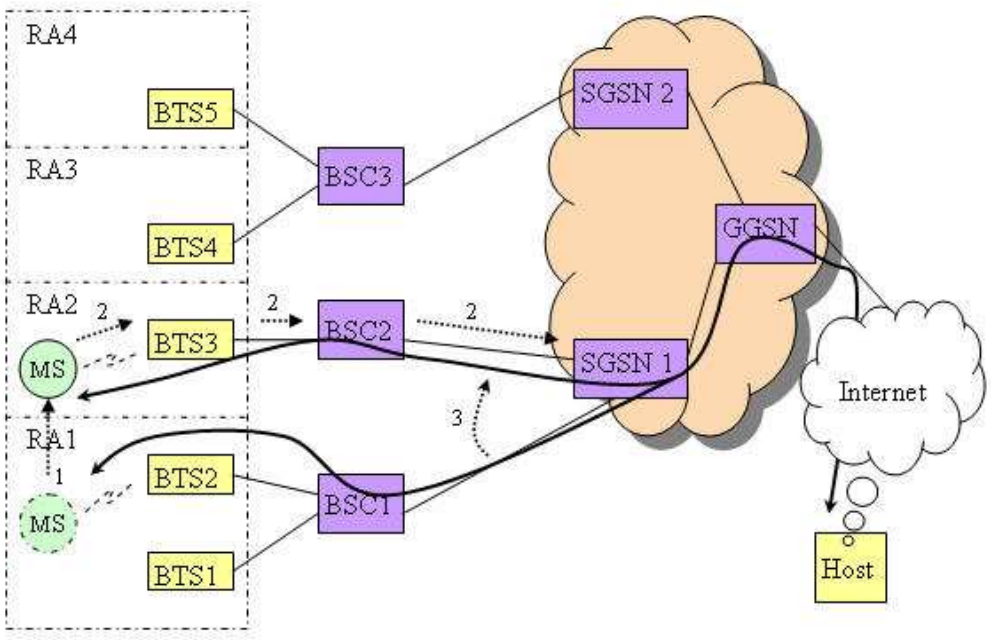


Figure A-3: Cell change – new cell in another RA handled by the same SGSN

GGSN (arrow 6 & 7) about receiving subsequent packets heading for the MS (arrow 8).

This series of communication among the affected BTS, BSC, SGSNs and GGSN will take some time before the data packets can be routed to the MS again. If the RAU Request from the MS is lost on the way to SGSN2 (e.g., due to bad radio conditions), typically, another RAU Request would be transmitted after 15 seconds. Then, it would take an even longer time for SGSN1 to identify that the MS has changed its RA.

Before SGSN1 gets to know that the MS has changed its RA, it would keep buffering any new packets sent by GGSN and would periodically retransmit the buffered packets that remain unacknowledged. And before the GGSN is informed of the change in the MS's RA, it would assume that the MS is reachable through SGSN1 and continue to send new packets to SGSN1. In this case, SGSN1 would probably run out of buffer space, and packet losses occur.

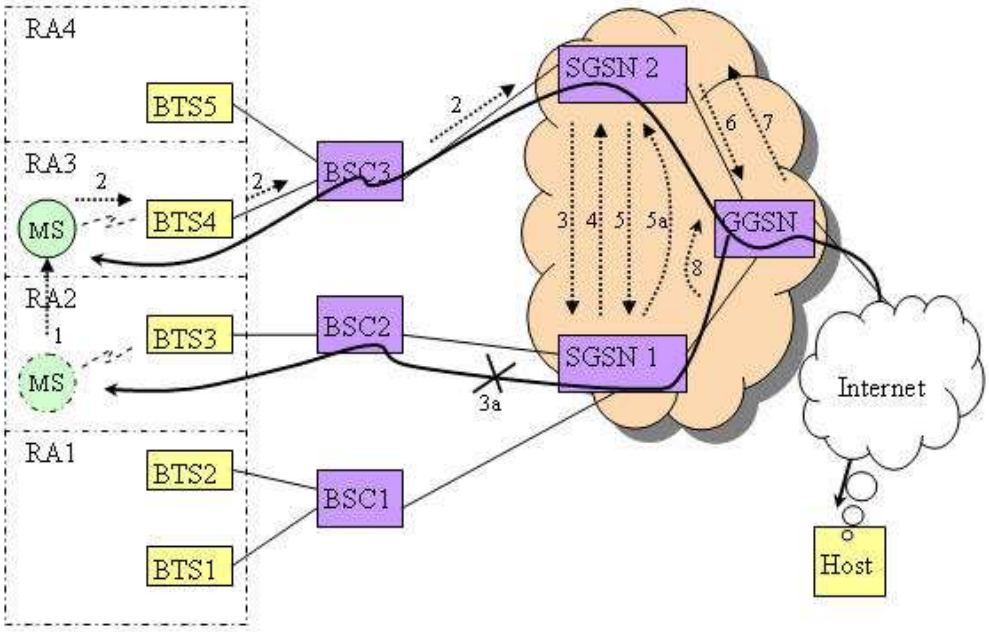


Figure A-4: Cell change – new cell in another RA handled by another SGSN