PROVIDING A SEAMLESS ACCESS TO MOBILE USERS

by

María Ciminieri

An Engineering Thesis submitted to:

Facultad de Ingeniería
Universidad de Buenos Aires

Fulfillment of the requirement of the degree of

Ingeniero en Informática

Departamento de Computación

Facultad de Ingeniería- Universidad de Buenos Aires

December 2013

Co-directed engineering thesis between the Universidad de Buenos Aires and TELECOM BRETAGNE within the framework of an internship agreement:
Ing. Alberto Dams, Universidad de Buenos Aires, Argentina.
Dr. Germán Castignani, Dr. Nicolas Montavont and Dr. Andrés Arcia-Moret,
Institut TELECOM / TELECOM Bretagne Rennes, France.
PROVIDING A SEAMLESS ACCESS TO MOBILE USERS

María Ciminieri

FACULTAD DE INGENIERIA - UNIVERSIDAD DE BUENOS AIRES
Departamento de Computación

Internet wireless access has become ubiquitous and one of its widely spread technology corresponds to 802.11. However, this kind of access is mostly limited to stationary users when they are within the range of WiFi Access Point. Whenever a handover occurs, a TCP transfer is interrupted. After this interruption the TCP sender should adjust its transmission rate to the one provided by the new WiFi Access Point. The main objective of this recovery is then, to allow a mobile station running an application, experience a seamless continuity in the service.

Therefor the interest in studying the mechanisms involved and implement a solution that provides seamless user mobility on this kind of networks. In order to achieve this goal, an enhancement mechanism to default TCP called Divacks [3], is here introduced, implemented and evaluated.
DEDICATION

To my family.

My mother is the person I would like to thank the most. Her unconditional love never ceases to amaze me. Thank you mom, I love you!!.

To my grandmother without whom I wouldn’t have been able to accomplish this career, she is the goddess of wisdom (shh). To my grandparents that are right next to me every step along, each one in their own way. To my best friend, my sister who has always been by me side during so many studying hours and exam’s jitters, love you bestie!

To my dad, that taught me to go "Siempre adelante" and care about nature. To my aunts Moni and Valdi, that are a second pair of parents to me, they are so caring and generous.

To the rest of my family: From my great grandparents to Lucas.
ACKNOWLEDGMENTS

First I would like to thank Dr. Germán Castignani for allowing me to be part of the TELECOM Bretagne research team. Thanks to this internship I was able to learn a lot, both my professional and personal skills.

Following, I would like to recognize my TELECOM Bretagne’s tutors: Dr. Nicolas Montavont and Andrés Arcia-Moret for their time and suggestions that made my work each day better.

I would like to acknowledge Professor Alberto Dams for accepting to be my tutor at the Universidad de Buenos Aires.

As this thesis was developed under an internship program in France, I would like to thank EGIDE/Campus France and ARFITEC for their financial support.

A big thanks has to be made to professors, researchers, administrative authorities and students of both the Facultad de Ingeniería and TELECOM Bretagne, for teaching me and helping me.

Finally I do not want to forget very important people that walked by me side these years: Mariano Simone, Ariel George, Ali Bize, Alejandro Lampropulos, Renzo Navas, Belén Sapia and Pablo Berrilio.
# Contents

Table of Contents \hfill v

List of Figures \hfill ix

1 Introduction \hfill 1

2 802.11 Networks and handovers \hfill 3
   2.1 802.11 Networks \hfill 3
   2.2 The OSI model \hfill 4
   2.3 The CSMA/CA protocol \hfill 4
   2.4 Handover process \hfill 6
      2.4.1 Types of handovers \hfill 6
   2.5 Effects of handovers \hfill 9
      2.5.1 Handover delays \hfill 9
      2.5.2 General effects \hfill 9

3 The Transmission Control Protocol \hfill 15
   3.1 The sliding window system \hfill 15
      3.1.1 Sliding window pointers \hfill 17
   3.2 Congestion control strategies \hfill 19
      3.2.1 Slow start \hfill 19
      3.2.2 Congestion avoidance \hfill 21
      3.2.3 Fast retransmit \hfill 22
      3.2.4 Fast recovery \hfill 22

4 State of the art \hfill 23
   4.1 A classification of the TCP enhancement mechanisms \hfill 23
   4.2 Existing TCP enhancement mechanisms \hfill 24
## CONTENTS

4.2.1 Bandwidth estimation ........................................... 24  
4.2.2 End-to-end solutions ........................................... 25  
4.2.3 Awareness of wireless links .................................... 27  
4.3 Metrics for mechanisms evaluation ................................. 28  
4.4 Evaluation of the mechanisms ...................................... 29  

5 The Divacks mechanism .................................................. 31  
5.1 The Divacks mechanism’s operation ................................. 31  
5.2 The Ping Pong effect .................................................. 33  
5.3 Divacks variations .................................................... 34  
5.3.1 Brute Force Divacks mechanism ................................ 34  
5.3.2 Controlled Divacks mechanism ................................. 34  

6 Implementation of the Divacks mechanism ........................... 37  
6.1 Obtaining the source files ........................................... 37  
6.2 Configuring the kernel ............................................... 38  
6.3 Implementing the Divacks mechanism ............................... 39  
6.3.1 Modifications in the client’s side ............................... 39  
6.3.2 Modifications in the server’s side ............................... 46  
6.4 Compilation of the modified kernel ................................. 47  
6.5 Usage ............................................................... 48  

7 Experimentation and Evaluation ....................................... 49  
7.1 Testbed configuration ............................................... 49  
7.2 Tools ............................................................... 51  
7.3 Testing conditions ................................................... 52  
7.4 Testing configuration parameters .................................. 53  
7.4.1 Number of Divacks per data packet ............................ 53  
7.4.2 Size of the transfer .............................................. 56  
7.4.3 Round Trip Time ................................................ 59  
7.4.4 Divacks emission ................................................ 61  
7.4.5 Reception buffer’s size ......................................... 61  
7.4.6 Receiver’s announced window (rwnd) ........................ 64  
7.4.7 Evaluation of the testing configuration parameters ........ 65  
7.5 Evaluating the Divacks mechanism ................................ 72  
7.6 Evaluating the Divacks mechanism ................................. 75  

on a mobile environment ................................................ 75
CONTENTS

7.7 Discussion ................................................................. 78

8 Conclusions and Perspectives ........................................... 81

Bibliography ................................................................. 84
List of Figures

2.1 Hidden Node Problem. ................................................................. 5
2.2 Graphical representation of a layer 2 handover. ............................. 7
2.3 Graphical representation of a layer 3 handover. ............................. 8
2.4 Handover’s delays: Handover latency and handover recover latency. ..... 10
2.5 Campus AP deployment for the android handover test. ..................... 11
2.6 Handovers impact on the data transfer: RSSI and id number of the AP to which the MS was connected to. ........................................ 11
2.7 Handovers impact on the data transfer: goodput on the MS side during the test was performed. ................................................. 12
2.8 Handovers impact on the data transfer: the data transfer is paused while handovers occur. ..................................................... 12

3.1 TCP sliding window system from the sender’s point of view. Send and usable window. .................................................................... 16
3.2 TCP sliding window system from the receiver’s point of view. Receiver’s window. ....................................................................... 16
3.3 TCP sliding window system: Sender’s pointers. ............................... 18
3.4 TCP sliding window system: Receiver’s pointer. ............................... 18
3.5 TCP slow start representation....................................................... 21

5.1 TCP acknowledgement system: one acknowledgement sent per data packet received. ................................................................. 32
5.2 Divacks acknowledgement system: several acks sent, three in this example, per data packet received. ........................................... 33

6.1 Screenshot of the menuconfig. Selection of the congestion control strategy. .............................................................. 39

7.1 Testbed for divacks testing. ........................................................... 50
7.2 Test No. 1: Congestion window values for different values of divacks sent. ................................................................. 56
7.3 Test No. 1: Sequence number measured in the client’s side for different values of divacks sent. ................................................................. 57
7.4 Test No. 2: Sequence number measured in the clients side for different sizes of files exchanged: 500 KB, 1, 2, 4 and 8 MB. ................................. 58
7.5 Test No. 3.2: Sequence number measured in the clients side for different sizes of files exchanged: 500 KB, 1, 2, 4 and 8 MB, when RTT = 500 ms. .... 60
7.6 Test 2: Divacks and full-acks emission speed by 0.1 s intervals. .............. 61
7.7 Test No. 4.1: Sequence number measured in the client’s side for different reception buffers’ sizes. ................................................................. 63
7.10 Test No. 4.1: Receiver’s announced window (rwnd) measured in the client’s side for different reception buffers’ sizes. ........................................ 67
7.11 Variation of divacks Mechanism: Brute Force and Controlled. Sequence number measured in the clients side. ................................................... 73
7.12 Mobility test: Sequence number measured in the clients side for a mobile test with a default reception buffer size. ............................................. 76
7.13 Mobility test: Sequence number measured in the client’s side for a mobile test with 4x default buffer size. ..................................................... 77
Chapter 1

Introduction

Internet wireless access has become ubiquitous and its main technology nowadays corresponds to 802.11 [11]. However, this kind of access is limited to static users when they are within the range of a Wi-Fi Access Point. And thus, the interest on studying the mechanisms engaged as to implement solutions that provide user mobility in this kind of networks. So far, it has been observed that the transport layer (see section 2.3) suffers from frequent connection interruptions due to the so-called handovers.

Whenever a MS (mobile station), moves from an AP (access point) to another one, a process called handover takes place. This process is the transfer of an ongoing call or data session from the AP the mobile station is currently connected to, to another one. Whenever a handover occurs, the TCP (Transmission Control Protocol) transfer is interrupted. After this interruption, the TCP sender should adjust its transmission rate to discover the one provided by the new AP. Therefor, the main objective of this recovery is to allow the mobile station, which most surely is running an application that involves a TCP connection, experience a seamless continuity in the service. This implies that whenever a mobile station performs a handover, the running applications should be the least affected by it: the delay of connecting to a new access point and the time needed to discover the available bandwidth must be as small as possible.
Chapter 1 Introduction

The first and most general goal, was to study the TCP mechanisms that take place when the available bandwidth needs to be discovered and how handovers have an impact on the data transfer. The second goal was to implement a fast-ramp up mechanism, the Divacks mechanism introduced in [3] by Arcia-Moret, on a Linux platform where its performance was thoughtfully tested. The third goal was to implement an enhancement to the existing mechanism to avoid the Ping Pong effect (introduced in section 5.2), the Controlled Divacks mechanism (see section 5.3.2). A set of factors that have an impact on the performance of the mechanism have been tackled and tested. These parameters are either related to the way in which the mechanism works or to the network and TCP configuration. Finally, the mechanism was evaluated for the first time in a mobile environment where handovers took place.

This document is organized as follows: In chapter 2, 802.11 networks are described and the different types of handover are explained, making a strong emphasis in layer 2 and layer 3 handovers. The TCP default mechanisms that take care of the bandwidth discovery and the impact handovers have on the TCP layer are introduced in chapter 3. In chapter 4 the existing solutions that handle with the discovery of the available bandwidth and that deal with handovers are analyzed. In chapter 5 the Divacks mechanism and it’s two variations are introduced and their functionality, explained. The goal of chapter 6 is to indicate all the necessary steps to deploy the divacks mechanism in a Linux kernel. In chapter 7, the results obtained by the evaluation of the behaviour of the mechanism and it’s interaction with the different parameters of the testbed, are displayed. Finally, conclusions are arisen in chapter 8.
Chapter 2

802.11 Networks and handovers

2.1 802.11 Networks

The IEEE 802.11 is a standard for implementing wireless local area network (WLAN) computer communication. These standards provide the basis for wireless network products using the Wi-Fi brand. It was first released in 1997.

The 802.11 family consist of a series of half-duplex over-the-air modulation techniques that use the same basic protocol. There are several amendments, they append a unique letter to the end of the name. A brief description of each one follows.

- **802.11** Provides 1 or 2 Mbps transmission in the 2.4 GHz band.
- **802.11a** Provides up to 54-Mbps in the 5GHz band.
- **802.11b** (also referred to as 802.11 High Rate or Wi-Fi) Provides 11 Mbps transmission (with a fallback to 5.5, 2 and 1-Mbps) in the 2.4 GHz band.
- **802.11e** Adds QoS features and multimedia support to the existing IEEE 802.11b and IEEE 802.11a wireless standards, while maintaining full backward compatibility with these standards.
- **802.11g** Is used for transmission over short distances at up to 54-Mbps in the 2.4 GHz bands.
- **802.11n** Adds adding multiple-input multiple-output (MIMO). The additional transmitter and receiver antennas allow for increased data throughput through spatial multi-
plexing and increased range by exploiting the spatial diversity through coding schemes like Alamouti coding. The theoretical speed that can reach is 128 Mbit/s.

- **802.11ac** It is actually under development. It operates only in the 5 GHz frequency range and features support for wider channels (80MHz and 160MHz), more streams (up to 8), and high-density modulation (up to 256 QAM) to reach a throughput of 1 Gbps.

- **802.11ad** (also referred to as WiGig) It operates in the 60 GHz frequency band and can achieve a theoretical maximum throughput of up to 7 Gbits.

- **802.11r** (also referred to as Fast Basic Service Set (BSS) Transition) It supports VoWi-Fi handover between APs to enable VoIP roaming on a Wi-Fi network with 802.1X authentication.

- **802.11X** Is a standard that allows network administrators to restrict the use of IEEE 802 LAN service APs to secure communication between authenticated and authorized devices.

### 2.2 The OSI model

The Open Systems Interconnection (OSI) model is standardization made to characterize the functions of a communications system in terms of abstraction layers. A layer serves the layer above it and is served by the layer below it. The seven layer model is the most well know, where layer are numbered from 1 to 7. The seven layers and their functionality are listed in Table 2.1.

### 2.3 The CSMA/CA protocol

The carrier sense multiple access with collision avoidance (CSMA/CA) is a network multiple access method in which carrier sensing is used. Here, nodes attempt to avoid collisions by transmitting only when the channel is sensed to be idle. This protocol operates in the Data Link layer (Layer 2) of the OSI model.

In wireless networks, this mechanism is of great importance because it solves the hidden node problem (present when using CSMA/CD) which can be seen in Fig. 2.1. Two nodes, A and C can be located in a way in which they are not aware of the existence of each other.
2.3 The CSMA/CA protocol

<table>
<thead>
<tr>
<th>Layer</th>
<th>Function</th>
<th>Data unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>7. Application</td>
<td>Supporting application and end-user processes</td>
<td>Data</td>
</tr>
<tr>
<td>6. Presentation</td>
<td>Data representation, encryption and decryption, convert machine dependent data to machine independent data</td>
<td>Data</td>
</tr>
<tr>
<td>5. Session</td>
<td>Establishing, managing and terminating connections between applications.</td>
<td>Data</td>
</tr>
<tr>
<td>4. Transport</td>
<td>Providing transparent transfer of data between end systems.</td>
<td>Segments</td>
</tr>
<tr>
<td>3. Network</td>
<td>Routing, forwarding, addressing, error handling, congestion control and packet sequencing.</td>
<td>Packets / Datagrams</td>
</tr>
<tr>
<td>2. Data link</td>
<td>Data packets encoding and decoding into bits</td>
<td>Frames</td>
</tr>
<tr>
<td>1. Physical</td>
<td>Media, signal and binary transmission</td>
<td>Bits</td>
</tr>
</tbody>
</table>

**Table 2.1** The OSI model.

Figure 2.1 Hidden Node Problem.

Because they cannot hear one another’s broadcast. Since they think that the link is available, the will start transmitting simultaneously. If there is a third node, B, in the middle that can hear both, a collision occurs. The Collision Avoidance method, avoids the hidden node problem by attempting to divide the channel equally among all transmitting nodes within
the collision domain. Now, the nodes will have to share the medium and transmit in a ping-pong fashion, by taking turns.

2.4 Handover process

A handover process takes place when a MS moves from an AP to another one. This process is the transfer an ongoing call or data session from the old AP to the new one. Since the TCP transfer is interrupted, the TCP sender needs to adjust its transmission rate to one provided the new AP. The main objective of this recovery is then, to allow a MS running an application, experience a seamless continuity in the service.

There are several scenarios in which a handover occurs. The first and most common one happens when the MS is getting out of range of the current AP, making the signal strength decrease and therefore degrading the connection. Another reason of a handover taking place is whenever an AP is overloaded with clients, it could force the MS to change of AP, alleviating the congestion [17].

2.4.1 Types of handovers

Handovers can be classified in different categories:

- **Vertical and horizontal handovers**: Vertical handovers happen when the internet connection changes from from one technology to another one. This is different from a horizontal handover, which happens between different devices that use the same technology. A vertical handover involves changing the data link layer technology used to access the network.

- **Hard and soft handovers**: They are also called Break Before Make and Make Before Break respectively. In the first case, the MS cannot maintain simultaneous communication with the new and the current device, while in the second case the connection to the new AP is established before the connection of the current one is broken.

- **Layer 2 and layer 3 handovers**: A handover might be categorized as layer 2 or 3 depending on whether the current and the new device the MS is attached to, are on the same IP subnet or not. This is introduced with more detail in the following section because of its direct impact on the application level.
2.4 Handover process

2.4.1.1 Layer 2 Handovers

A layer 2 handover takes place when both APs, the current and the new one, belong to the same network, which means that they have the same SSID (Service Set IDentifier) number as Fig. 2.2 shows. When a layer 2 handover occurs, the IP address to which the MS is connected to, remains the same. This does not mean that there is not an interruption in the data transmission, the two handover delays mentioned in section 2.5.1 take place.

The handover recovery delay, in layer 2 handovers might happen because, (1) the available bandwidth in the new access point is lower than the one of the previous AP. In this case, the sender’s TCP cwnd (Congestion window) should be reduced and adapted faster to the

**Figure 2.2** Graphical representation of a layer 2 handover.
correct bandwidth sharing and (2), if the available bandwidth at the new AP is larger, a faster TCP slow start is desirable to improve the transmission performance (see section 3.2).

2.4.1.2 Layer 3 Handovers

On the other hand, a layer 3 handover occurs when the old access point and the new one, belong to the different IP subnets as shown in Fig. 2.3. When a layer 3 handover occurs, the new roaming domain provides the MS a new IP address and consequently, a new port number. This changes have to be managed in order to reduce the impact that they might have on the applications running in the MS. Mechanisms like Mobile IP are used as to keep the same socket (and therefore TCP connection) that was established before the handover occurred.
2.5 Effects of handovers

In this section the effects that a handover has on the ongoing data transmission and the different handover delays, are introduced. A test in which handovers took place and their effects are visible, is going to be displayed.

2.5.1 Handover delays

Whenever a handover occurs two different delays can take place. The (1) handover latency and the (2) handover recover latency. Their length depends on the nature of the handover and the network characteristics. Fig. 2.4 illustrates a data transfer and its throughput. It shows that after a handover there is a period of time in which the new available bandwidth needs to be discovered.

**Handover latency**   This latency is the time introduced by a handover in which there is no data exchanged. This latency must be reduced because some of the applications running in the MS are going to feel a disconnection or their performance might degrade.

**Handover recover latency**   This latency is the time that it takes, after a handover occurs, to discover the available bandwidth provided by the new AP. This latency should be reduced as to reach as fast as possible the available data rate. To do so, the TCP connection triggers a legacy slow start algorithm which increases the cwnd in an exponential mode. A fast ramp-up growth of the cwnd reduces the impact that the handover has on the running applications.

2.5.2 General effects

When a handover occurs there are several behaviours related to the data transfer behavior and the TCP protocol, that might take place. These are:

- **The TCP transfer is interrupted:** The data transmission has a gap in which there is no data downloaded to the MS since there is no connection to an AP to do so (see section 2.4.1, Hard handovers). This effect is going to be quantified by the *Handover latency* length.

- **The TCP connection can be interrupted:** This happens when a handover involves changing of IP subnet. If the handover is managed to avoid so (like Mobile IP [17]), this interruption can be avoided.
The available bandwidth needs to be discovered: The TCP sender should adjust its transmission rate to the one provided by the new AP (see section 3.2). It is here quantified by both the throughput and the Handover recover latency.

2.5.2.1 Experimentation: Handover effects

Tests were performed in which an Android ICS 4.0.3 system working on a Samsung Nexus S (GT-I9023) smart phone, downloaded data while executing layer 2 handovers between different APs. The deployed testbed is shown in Fig. 2.5: nine AP (Linksys Wireless-G Broadband Router, WRT54GL model (2.4 GHz, 64 Mbps)) were deployed in the TELECOM Bretagne campus and the MS moved between them. Meanwhile, the TCP connection was never interrupted.

The RSSI (Received signal strength) is the signal emitted by the AP that the MS measures. This value is obtained periodically (every 500 ms) by beacons in which access points announce to all MS listening, certain information about the network while announcing the presence of a Wireless LAN. The beacon emitted also provides all the needed information that allows mobile stations to establish a connection with the access point.
The results obtained are now explained. In Fig. 2.6 the RSSI of the AP the MS was connected to is plotted in green, its scale is represented in the left y axis. In this graph, the right y axis represents the APs identification numbers and the red flat lines show the connection between the MS and a certain AP. Therefore is easy to see that before a handover occurs the RSSI decreases, displaying that the MS is going out of range of the AP. After the handover, when the MS is connected to the new AP, the RSSI is large again. In Fig. 2.7 the downloaded data (measured with the sequence number) is plotted. If figures 2.6 and 2.7 are compared, the interruption that a handover introduces in the data transmission, is noticeable: in Fig. 2.8 flat periods on the data transmission (meaning that no data was exchanged) are highlighted.
Figure 2.7 Handovers impact on the data transfer: goodput on the MS side while the test was performed.

Figure 2.8 Handovers impact on the data transfer: the data transfer is paused while handovers occur.
What is the importance of these effects?

The answer to this question strongly depends on the type of applications the end user is running. In some applications, just like sending, reading an email or browsing the Web, the MS functions as a client in a client-server architecture, which is not strongly affected by small disconnections. On the other hand, real time applications are more sensitive to changes in the throughput and interruptions in the data transmission.

When any kind of handover occurs, an interruption in the transmitted data flow takes place. The problem is not only the delay that this interruption introduces, but also, how the handover affects the transmission rate. This is not only due to the probable fact that the available bandwidth of both APs is different, but also because of the TCP algorithms that are going to be executed in order to discover this new bandwidth. In the following chapter, these algorithms are thoughtfully explained.
Chapter 3

The Transmission Control Protocol

The TCP, Transmission Control protocol, is a transport layer protocol. This protocol is characterized for providing a stream oriented connection, which means that the data is broken into segments (without any pre established size and structure) that have no sense for the application layer. Here, the sliding window system, which allows this behaviour to take place, is explained, together with the two TCP Congestion Control strategies: slow start algorithm and congestion avoidance, that manage the way in which the available bandwidth of the network is discovered.

3.1 The sliding window system

In TCP, a sequence number is used to keep track of the stream data that is being exchanged. It uses a sliding window system to indicate the number of data an endpoint can receive in a time moment. This system provides:

- reliability, by detecting and re-sending the lost segments and
- a data flow control, by controlling the rate in which data is sent as not to overload the receiver (which would make him discard segments).

The sliding window acknowledgement system makes use of an enhanced-PAR (Positive Acknowledgement with re-transmission) algorithm. The regular PAR algorithm needs the acknowledgement of a message to be received before sending the next message or a timer to be expired (indicating that a message has been lost) before re-transmitting the lost message. Enhanced-PAR gets rid of the timer and does not need to receive an acknowledgement before sending the next packet [16]. TCP uses a sliding window acknowledge system in which bytes are divided into four different categories, see Fig. 3.1. These categories are:
• **Sent and acknowledged:** these are the older bytes in the transmission, the ones that were sent and already acknowledged by the receiver.

• **Sent and not acknowledged:** bytes that have been sent but their acknowledgement has not been received by the sender yet.

• **Not sent but the receiver is ready to receive:** the receiver could receive this data in a burst without having any congestion problems.

• **Not sent data that the receiver is not ready to receive:** this is the data that the server could sent but the receiver could not be able to handle.

![Figure 3.1](image1.png) **Figure 3.1** TCP sliding window system from the sender’s point of view. Send and usable window.

Both the receiver and the sender keep track of these numbers for both streams of data: the data they are sending and receiving. When they are keeping track of the data they are receiving, categories 1 and 2 are collapsed into one: bytes received and acknowledged, see Fig.3.2.

Some definitions arise from the categories in which bytes, by the sender’s point of view,

![Figure 3.2](image2.png) **Figure 3.2** TCP sliding window system from the receiver’s point of view. Receiver’s window.

are group together. The TCP protocol defines a send and a usable window:
3.1 The sliding window system

• **Send window / Sender’s window / Window:** its the number of bytes that the receiver tells the sender that is willing to receive at one time.

   \[ \text{Send window} = (\text{Sent and not acknowledged} + \text{Not sent but receiver is ready to receive}) \text{ bytes} \]  

   (3.1)

• **Usable window:** its the number of bytes that could be sent by the sender in a time moment.

   \[ \text{Usable window} = (\text{Not sent but the receiver is ready to receive}) \text{ bytes} \]  

   (3.2)

A third window is defined, from the receiver’s point of view.

• **Receiver’s window:** its the number of bytes that the receiver is ready to receive in a period of time.

The sizes of the windows vary depending on the available bandwidth in the connection and the TCP algorithms that are been executed. Whenever an acknowledgement with a sequence number is received, it means that all the bytes before that sequence number have been correctly received by the receiver (*cumulative acknowledgement system*). When this happens, some bytes from category two (sender’s side) are transferred to category one. Since the window size did not change, it "slides" to the right, allowing bytes from category four to now be category three, making the usable window change its size depending on the number of acknowledged bytes.

A characterization of the sender’s window can be made by saying how it is evolving. The window *closes* whenever an acknowledgement is received. It *grows*, which means that the right edge moves to the right, when there is more data that the receiver can receive in a time moment. If the window *shrinks*, it means that the receiver can handle a smaller amount of data than before (the right edge is moving to the left).

The sender’s window and the receiver’s window are complementary: the send window of one device is the receive window of the other, and vice-versa.

3.1.1 Sliding window pointers

To keep track of the bytes that should be sent and those that the receiver should acknowledge, the sender uses some pointers. They are shown in Fig. 3.3 and defined here below:
• **SND.UNA**: the sequence number of the first sent but not yet acknowledged byte.

• **SND.NXT**: sequence number of the next byte of data to be sent.

• **SND.WND**: indicates the number of bytes that the receiver is willing to receive at one time.

![Figure 3.3 TCP sliding window system: Sender's pointers.](image)

Therefore, the number of bytes that the sender can send in a time moment (Equation 3.2) can be redefined by using the TCP sender’s pointers, see Equation 3.3.

\[
	\text{Usable window} = (\text{SND.UNA} + \text{SND.WND} - \text{SND.NXT}) \text{ bytes}
\]  

(3.3)

The receiver also uses pointers to keep track of the data it should receive from the sender.

• **RCV.NXT**: sequence number of the next byte of data expected to be received.

• **RCV.WND**: indicates the number of bytes that it is willing to receive at one time from the sender.

![Figure 3.4 TCP sliding window system: Receiver’s pointer.](image)
3.2 Congestion control strategies

When the receiver sends an ack with the sequence number stored in RCV.NXT it indicates the sender that all the bytes before the one actually being acknowledged, have been received correctly. The RCV.NXT variable should store the same value that the sender registered in it’s SND.NXT variable. If both values match, it indicates that no packet was lost; otherwise it means that packets are lost or some in-flight-packets have not been taken into account by one of them.

3.2 Congestion control strategies

There are some TCP mechanisms that manage how these windows size evolve depending on the network and connection characteristics. There are four congestion control strategies [2]: (1) slow start, (2) congestion avoidance, (3) fast retransmit, and (4) fast recovery. They try to achieve the maximum available bandwidth while avoiding congestion in the receiver’s side. These strategies redefine new windows to estimate the existing congestion between the two sides. They are:

- **Congestion window (cwnd):** Is the number of outstanding bytes at a given time, which corresponds to the already defined *Sender’s Window*. It is defined and increased/decreased in the sender’s side.

- **Receiver’s announced window (rwnd):** Is the amount of data that the receiver is ready to receive at a given time, which corresponds to the already defined *Receiver’s Window*.

The sender always sends data with a sequence number higher than the sum of the highest acknowledged sequence number and the minimum of cwnd and rwnd [2], as Equation 3.4 shows the smaller one governs the data transmission.

\[
\text{Effective window size} = \min\{rwnd, cwnd\} \tag{3.4}
\]

3.2.1 Slow start

When a connection is recently established segments need to be transferred in a moderate fashion as not to saturate the receiver and the existing connections in the network. Therefore, TCP must slowly probe the network to determine the available capacity. The slow start algorithm is used for this purpose at the beginning of a transfer, or after repairing loss
detected by the retransmission timer.

The SMSS (Sender maximum segment size) is the size of the largest segment that the sender can transmit. This value can be based on the maximum transmission unit of the network, the path MTU (Maximum Transmission Unit) discovery algorithm, RMSS (receiver’s maximum segment size), or other factors. The size does not include the TCP/IP headers and options. The RTT (Round Trip Time) is the length of time it takes for a signal to be sent plus the length of time it takes for an acknowledgement of that signal to be received. At the beginning of a TCP connection, the value of the cwnd is equal to one, two, three or four SMSS; depending on it’s value. Therefore, the sending rate is equal to SMSS/RTT. There is another variable, the slow start threshold (ssthresh), which is used to determine whether the slow start or congestion avoidance algorithm is used to control data transmission. The initial value of ssthresh SHOULD be set arbitrarily high, this allows the network conditions, rather than some arbitrary host limit, to dictate the sending rate.

To fast discover the available bandwidth, whenever an acknowledgement is received, the cwnd size is increased by a SMSS (see Fig. 3.5), which makes it grow in an exponential way: the host A sends one segment, when it is acknowledged by host B, the value of cwnd grows to two SMSS so two segments are sent, when they are acknowledged, cwnd grows to 4 SMSS, and so on. Therefore the sending rate gets doubled every RTT.

This will happen until one of three things happens:

- A packet is lost (indicating that the link is congested), which is indicated by a time out.
- The ssthresh threshold is reached.
- Three duplicate ack are received.

The first time a time out occurs, the value of $1/2 \times cwnd$ (half the size the window reached before the time out) is going to be stored in the ssthresh variable and slow start begins again with $cwnd = 1 \text{ SMSS}$ (or the initial value it had). Whenever the threshold is reached, the size of the sender’s window is not doubled (which would probably end up in congestion), but TCP enters to the congestion avoidance mode (explained in the following section). Finally, when three duplicate acks are received, TCP enters the Fast Retransmit state (see section 3.2.3).
3.2 Congestion control strategies

3.2.2 Congestion avoidance

This algorithm takes place when cwnd is equal to half the value when the last congestion occurred. Therefore, the size of cwnd is not going to be doubled, but linear growth of one SMSS per RTT takes place.

This increase is ceased when: (1) a new time out happens, case in which slow start state takes place or (2) three duplicate acks are received; the value of ssthresh is recorded as half the value of the Flight-size (the amount of outstanding data in the network) and the fast recovery state is entered.
3.2.3 Fast retransmit

When the receiver receives a segment with a sequence number higher than the expected one, it assumes that there is another segment missing. To tackle this, the receiver sends an acknowledgement specifying the first byte number of the expected segment. Eventually, the sender concludes that a segment was lost, even if its retransmission timer has not expired.

The Fast Retransmit algorithm detects and repairs loss based on the reception of three duplicated acknowledgements. When this happens, TCP assumes that the segment starting with byte the acknowledgements are announcing, has been lost usually because it was dropped due to congestion. In this case, the device immediately retransmits the missing segment without going through the normal retransmission queue process. This improves performance by eliminating delays that would suspend effective data flow on the link. In the fast retransmit state the value of the cwnd is increased 1 SMSS by duplicate ack received for the segment that caused TCP to enter to Fast Recovery state. When the segment is received, TCP goes back to congestion avoidance with a reduced cwnd. On the other hand, if a time out happens, slow start is triggered with congestion window equal to one SMSS and ssthresh value is set to half the value of cwnd when the loss occurred.

3.2.4 Fast recovery

When Fast Retransmit is used to re-send a lost segment, the device using it performs Congestion Avoidance, but does not use slow start to increase the transmission rate back up again. This happens because since multiple acknowledgements were received by the sender all indicating receipt of out-of-order segments, this would indicate that several segments have already been removed from the flow of segments between the two devices. For efficiency reasons, then, the transmission rate can be increased more quickly than when congestion occurs in other ways. This improves performance compared to using the regular Congestion Avoidance algorithm after Fast Retransmit. The TCP Reno version of TCP includes Fast Recovery, which is not included in the TCP Tahoe version.
Chapter 4

State of the art

In this chapter, several existing enhancements to the TCP protocol are addressed. The existing solutions have a different nature depending on how the discovery of the new available bandwidth is done and if whether it only takes place after a handover or not.

4.1 A classification of the TCP enhancement mechanisms

The enhancements can be characterized by the way they work. Here three nonexclusive groups are defined, which means that a solution may be be part of more than one group at a time.

- **Bandwidth estimation.** Some schemes try to estimate the available bandwidth before establishing an actual sending rate. The estimation can be done by the end host himself or with information provided by other connections or intermediate nodes in the path.

- **End-to-end solutions.** These kind of mechanisms are based on having only the end hosts modified rather than the intermediary nodes. There are solutions that rely on the local buffering of either frames or acknowledgements, choosing the sending rate by themselves (by directly modifying the value of rwnd or the amount of data to send by RTT). In this kind of approaches, a balance needs to be found because if acks are buffered for a long time, retransmissions take place, making the mechanisms useless.

- **Awareness of the wireless link.** The essence of these kind of mechanisms is that both the server and the client are aware of the wireless link. Different solutions can take place, some of them inhibit the congestion control strategies in the wireless link.
Others, called split-connection approaches, divide the connection into two transport-layer connections: one between the client and the wireless access point, and another one between the wireless AP and the server. By doing so, different behaviours of TCP are handled in each part of the connection.

4.2 Existing TCP enhancement mechanisms

There are diverse TCP enhancement solutions that handle the problem introduced by the discovery of the available bandwidth. Here, they are going to be introduced by using the classification explained in the previous section.

4.2.1 Bandwidth estimation

Some schemes listen to the network traffic and force some actions in order to have an estimation of the available bandwidth.

To start with, the Swift Start by Partridge [22] mechanism is a solution that combines packet pair, to estimate the initial capacity, with pacing to spread out the initial burst until self-clocking is established. While transmission packets get separated in time at the bottleneck. The receiver acknowledges the data packets and when the sender receives them, he can measure the time between the arrivals of the acknowledgements. With this information and knowing the size of the data packets sent, the sender can now estimate the available capacity. Therefore the mechanism proposes an estimation of the size of the cwnd by Equation 4.1.

\[
\text{Estimated cwnd} = \text{bottleneck capacity} \times \text{RTT}
\]

On the other side, packet pacing is used to send the TCP segments paced so they do not arrive all together at the start of the round-trip cycle.

A second kind of approach, takes profit of the already established connections to inform the new established one the appropriate size of cwnd that should be used for the path concerned. The mechanism called Congestion Manager [8] by Balakrishnan is based on the lack of probing the network and triggering of slow start algorithm by the new TCP connection. This is an end-to-end solution. The congestion manager maintains congestion parameters and networks statistics information which then is exposed to an API that enables applications to learn about the network characteristics, send information to the congestion
manager, and schedule data transmissions. By doing so, the initial discovery phase is reduced to a request to the congestion manager.

A similar mechanism involves the collaboration between the end hosts and path routers. The Quick-start [23] mechanism by Sarolahti, allows TCP to explicitly request permission (from routers along the network path) to send at a higher rate than the normally allowed by TCP’s congestion control strategies. There is a high restriction in the implementation of this solution because it requires support from all the routers in the path.

4.2.2 End-to-end solutions

Some mechanisms base their functionality on the fact that they are end-to-end solutions, in which collaboration between the network and its elements is not needed for the mechanism to work.

The Jump Start technique introduced by Lui in [18] removes the traditional start up phase present in the slow start TCP algorithm. It does so by beginning the transmission at whatever rate the algorithm finds appropriate. The chosen rate will be the minimum between rwnd and the amount of data queued locally for transmission. When using this mechanism, retransmissions are probable but is a neglectful cost versus the gain obtained by almost transferring all data of the transfer in the first RTT (which tests have shown possible). This is a sender-side change to the TCP’s congestion control strategies. Simulations of the Jump Start mechanism have shown that the individual connection’s performance can be degraded and also that the overall congestion level on the network might increase.

On the same direction Goff proposes Freeze-TCP from [13], a mechanism that modifies (by freezing) the receiver’s window before a predicted disconnection happens. With this behaviour, loosing packets is avoided. When the connection is re-established, the saved value of rwnd is re-announced to the server, forcing the transmission to restart at the previous rate. This mechanism only requires changes to be made in the client side.

4.2.2.1 Acknowledgement mechanisms

There is a group of mechanisms that try to increase as fast a possible the throughput by modifying the way in which the receiver sends the acknowledgements. Some of these mechanisms are going to be introduced in this section.
The first one, by Caceres [10], is a solution only valid in mobile environments were handovers take place. After a handover, a time out timer is started after reducing the size of the congestion window. Three copies of the same ack (that acknowledges the last data packet received before the disconnection) are sent. This forces a new packet to be sent and therefore discovering the new available bandwidth faster.

A second mechanism introduced by Miten in [20], called Delayed Duplicate Acknowledgements, proposes to delay the emission of duplicate acknowledgements during for some time, during which the packet may reach the receiver due to local link level retransmission. Therefore, this mechanism avoids unnecessary invocation of congestion control mechanisms at the sender’s side.

### 4.2.2.2 Divided acknowledgement mechanisms

Some mechanisms base it’s functionality on the division of each acknowledgement of data packets into several ones. By doing so, the congestion window grows allowing more data packets to be transferred.

The Spack mechanism by Jin in [15], sends several acknowledgements from the Base Station to sender when a time out ends (similar to the TCP retransmission) of a detected loss. The mechanism will one ack per byte to acknowledge, but it wont overwhelm the link since it will only be for the lost packet. This mechanism has an important disadvantage: it works as long as the IP traffic is not encrypted since the Base Station needs to look inside the TCP header. On the other side, it does maintain end-to-end semantics. The Spack mechanism has only been tested by simulation.

A second mechanism proposed in [19] by Matsushita, also sends divided acknowledgements but only after a vertical handover takes place. When this happens the BDP (bandwidth delay product) changes from the connection before the handover occurred with the one discovered in the new link. If the new BDP is smaller then duplicated acknowledgements are sent, and if is bigger, since the available bandwidth needs to be discovered, divided acknowledgements are going to be sent. The advantages of this solution are that changes are only made in the receiver’s side and the divacks rate is varied depending on the delay-bandwidth products. The results of the mechanisms where obtained by performing simulations.

The Ack splitting mechanism by Hasegawa [14], tries to distinguish between congestion
in the wired link and bit errors in the wireless link as to only take action in the last scenario. To do so, a local estimation of the cwnd in the receiver’s side is made to calculate the number of divided acknowledgements needed to go back to the original lost rate. This mechanism also adapts the divacks rate to the available uplink bandwidth as not to saturate the link. A drawback is that the mechanism has only been tested in a simulator.

The Divacks mechanism by Arcia Moret [3], divides the acknowledgements as to gain a larger throughput by forcing the growth of the cwnd. This mechanism does not need any packet signaling and since changes are only made in the receiver’s side (it maintains end-to-end semantics). It also works while IP encryption is used since there is no need in snooping the TCP headers like the Spack mechanism.

4.2.3 Awareness of wireless links

There are two main mechanisms that split the connection in two parts. I-TCP, Indirect TCP proposed by Bakre in [7], proposes to remove TCP from the wireless link and assign the responsibility of the emission of acks to the AP. The second one is more complex. M-TCP, proposed by Brown in [9], also splits the connection in two: one partition is MS to AP and the other is from AP to sender. This mechanism combines its functionality with the way in which Freeze TCP works. When a disconnection or a packet loss is detected, the AP forces the sender into persist mode, where it forces the cwnd size not to be dropped by holding back the ack to the last byte.
4.3 Metrics for mechanisms evaluation

In [13], evaluation factors are introduced as to highlight the strengths and drawbacks of the existing mechanisms. This list was taken and extended, as to characterize the mechanisms so far introduced.

- **Encrypted traffic:** In cases where the whole payload is encrypted, which is the case of IPSEC in IPv6, mechanisms that require intermediaries or snooping are not feasible.

- **Interoperation with the existing infrastructure:** To assure that interoperation is maintained, no changes should be required at intermediate routers or the sender’s side, which are generally unavailable for modifications. Mechanisms that split the connection require modifications and processing at intermediate nodes.

- **Scalability:** Access points have to buffer data of all the MS that are connected to it (and process this data, to some extent). When the MS moves, all of it’s data and the connection information, has to be transferred to the new AP. This creates overhead and makes the sender drop the cwnd, defeating the original purpose of these mechanisms.

- **Frequent disconnections:** As frequent disconnections are prompt to happen in wireless environments, it’s important that mechanisms handle this kind of disconnections. There are some approaches that need a certain period of time as to be able to react to handovers.
4.4 Evaluation of the mechanisms

In this chapter, several enhancements to the TCP protocol were introduced. But some of them are best suited than others if they are evaluated by the metrics introduced in the previous section.

The encryption of the IP payload makes the Indirect TCP and M-TCP mechanisms fail to work since they are based on the AP mediating and snooping the traffic. Mechanisms like Congestion Manager and Quick Start depend on the network to perform the estimation of the bandwidth, they are not scalable. End-to-end solutions are preferred since only the hosts need to be modified to make the mechanism work.

The delayed duplicate acknowledgement solution has shown good results but it can degrade the performance in presence of occasional transmission losses. Testing on the Jump start mechanism has shown that it can degrade individual connection’s performance while increasing the overall congestion level on the network. Also, because the Swift Start algorithm is based on network estimation, it is unstable due to the network dynamics shown by [1].

The Freeze TCP and the Divacks [4] mechanisms have a lot in common. They both are end-to-end schemes that not require intermediates to participate in the flow control, they simply exploit the way in which the TCP protocol works. The inter-operability with the existing infrastructure is guaranteed. The drawback of the Freeze TCP is that it needs to know that mobility is taking place as to trigger the mechanism, therefore the NIC vendors need to provide details on their roaming and handover algorithms.

The Divacks mechanism was chosen to be implemented and tested in a real environment. The manner in which Divacks works is addressed in detail in the next chapter: The Divacks Mechanism, followed by the details on an implementation in a Linux kernel in chapter 6. Finally, the results obtained are shown in chapter 7: Experimentation and Evaluation.
Chapter 5

The Divacks mechanism

So far, the different kinds of handovers and their influence on an ongoing TCP connection have been characterized. The mechanism proposed to reduce the impact that this interruption has on the running applications, is the Divacks mechanism. The Divacks mechanism is based on using the TCP data acknowledgements not only to indicate that a packet was correctly received but also to force the congestion window, while slow start is running, to grow in a more aggressive fashion. This growth increases the speed in which the available bandwidth, of the new AP, is discovered. Therefore the needed time to download all the data from the sender is reduced. How the divacks mechanism operates in order to gain speed in TCP transfers is going to be explained in this chapter. Also, an enhancement to the Divacks mechanism is introduced: the Controlled Divacks mechanism that is able to maintain the gain that the mechanism provides while not introducing unnecessary traffic that might lead to the Ping Pong effect.

5.1 The Divacks mechanism’s operation

The basic idea behind the Divacks mechanism is that whenever a data packet is received, the receiver will not send one acknowledgement for each TCP data segment received but several ones (each one acknowledging a portion of its data), called divacks for divided acknowledgements. The Divacks mechanism takes advantage of the relationship between the acknowledgement emission, performed by the receiver, and the sender’s TCP cwnd size. As explained in section 3.2.1 the value of the sender’s congestion window limits the number of unacknowledged bytes that can be sent to the client in a given time. Because this value is increased/decreased by the reception/loss of acknowledgements, if more acknowledgements are sent by the receiver (without any losses) in a time moment; the sender’s cwnd will be
increased faster, allowing the sender to increase the throughput.

![Diagram of TCP acknowledgement system: one ack per data packet.](image)

**Figure 5.1** TCP acknowledgement system: one ack per data packet.

In the TCP protocol, the receiver sends one acknowledgement (with the expected acknowledgement sequence number) to the sender, for each data segment received. In Fig. 5.1 a data segment is sent with 1448 bytes, numbered from y to y+1448. The receiver will then acknowledge the data segment by sending an acknowledgement with acknowledgement sequence number equal to y+1449.

The basic idea behind the Divacks mechanism is that whenever a data packet is received, the divacks client will divide the original acknowledgement in n acknowledgements, each one of them will acknowledge a smaller amount of data than the original one. The number of divided acknowledgements, here called "n", is a quantity defined by the user, and can be changed on runtime. The first divack will almost acknowledge all the data segment. It’s size is determined by Equation 5.1. The following divided acknowledgements will only acknowledge one byte, until the last one, which also has its acknowledgement sequence number matching the original one. In Fig. 5.2 it can be seen that the data packet received has a length of k bytes. Since the chosen value for n is three, there are going to be three acknowledgements sent corresponding to this data segment. The first divack acknowledges k-(3-2) bytes (with acknowledgement sequence number equal to y+k-1. The next n-1 divacks, two in this example, only acknowledge one byte. The first one has its acknowledgement sequence number equal to y+k. The last divack, will have its acknowledgement sequence number equal
5.2 The Ping Pong effect

The Ping Pong effect introduced in [5] is a reaction that takes place during a TCP connection in which there is a large amount of packets to be transmitted via a wireless link. Since the

\[ FirstDivack_{BytesToAcknowledge} = (segment, size - (n - 2)) \text{ bytes} \]

The reader might be driven to think that the higher the number of divacks sent, the faster the cwnd grows and therefore the data rate increases. This is not the case because there is a threshold between the number of divacks sent per data segment and the actual gain obtained in the total throughput. This limit is characterized by the *Ping Pong effect* explained in the following section.

**Figure 5.2** Divacks acknowledgement system: several acks sent, three in this example, per data packet received.

to $y+k+1$, just as if default TCP was operating (this is why this last divack is sometimes called full-acknowledgement/full-ack).

\[ FirstDivack_{BytesToAcknowledge} = (segment, size - (n - 2)) \text{ bytes} \]

The reader might be driven to think that the higher the number of divacks sent, the faster the cwnd grows and therefore the data rate increases. This is not the case because there is a threshold between the number of divacks sent per data segment and the actual gain obtained in the total throughput. This limit is characterized by the *Ping Pong effect* explained in the following section.
Divacks mechanism profits by dividing one original acknowledgement into several ones, the traffic in the uplink (client to server) increases, and so does the downlink traffic (in response to the Divacks mechanism, the server sends more data packets per time moment). Therefore, by increasing the number of divacks sent, the amount of packets sent in both links also increases.

The CSMA/CA protocol (section 2.3) forces the AP and the client to either send packets or receive them, but not both at the same time. Packets are going to be sent in a ping pong fashion: the client and the AP have to take turns to send packets to each other. If the traffic increases because of the Divack mechanism, it is not certain that the performance improves since it is a shared medium. There is a limit to be found: the maximum number of divacks send per data segment without falling into the ping pong effect, which saturates the wireless link.

5.3 Divacks variations

Having the consequence of the Ping Pong effect in mind is that it was decided to built an enhancement to the existing Divacks mechanism. The original mechanism sends divacks as long as the TCP slow start algorithm is taking place, this is why it is called Brute Force Divacks mechanism from now on. The upgrade to this mechanism, called Controlled Divacks mechanism, only allows a defined number of divacks to be sent, after this limit is reached, the Divacks mechanism is inhibited and only full-acks (regular TCP acknowledgements) are sent.

5.3.1 Brute Force Divacks mechanism

When the Brute Force variation in enabled, \( n \) divacks are going to be sent per IP packet received when TCP is in the slow start state. This mechanism is an aggressive one, which does not imply that throughput is be larger in correspondence.

5.3.2 Controlled Divacks mechanism

The Controlled Divacks mechanism takes advantage of the limitations that the wireless link introduces and reduces the unnecessary uplink traffic. This is achieved by setting a limit on the number of divacks to be sent during a TCP connection.
5.3 Divacks variations

To achieve this behaviour, two extra parameters are needed: one to keep track of the number of divacks already sent: `tcp_divack_count` and another one that sets the limit until divided acknowledgements are sent: `tcp_divack_max_count`. When the connection is established, the value of `tcp_divack_count` is equal to zero. Every time a data packet is received `n` divacks are sent and `tcp_divack_count` is incremented by `n`, until `tcp_divack_count` reaches the limit set by `tcp_divack_max_count`. When this happens, only one acknowledgement per IP packet (full-acks) is going to be sent.

Sending divacks only at the beginning of the TCP transfer forces the growth of the cwnd, which makes the discovery of the available bandwidth at take-off much faster than default TCP; but it also frees the wireless link, allowing the data packets to be transferred without congestion (due to unnecessary divacks).
Chapter 6

Implementation of the Divacks mechanism

The goal of this chapter is to indicate all the necessary steps to deploy the Divacks mechanism in a Linux kernel. The changes here introduced are based in [12] with their authors authorization. This operating system was chosen because of its open source software development and distribution characteristics. The Linux kernel version 2.6.35.7, also called Yokohama, was chosen because it is a stable version (released August 1st, 2010).

To implement the mechanism, both the divacks server and the divacks client need to be modified as to allow the emission of divacks in the client side and the reception of them, in the servers side. In this section all the modifications done on both kernels can be found.

Initial Requirements
The computers where divacks is going to be implemented and deployed must be running Ubuntu 10.04. This can be checked by typing:

```
$ lsb_release -a
```

6.1 Obtaining the source files
The v2.6.35.7 Linux kernel source files can be obtained from the official site:

```
$ wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.35.7.tar.bz2
```
Move the zipped kernel from the download directory to /usr/src and unzip it:

```
mmmary@mmmary-laptop:~ $ sudo mv ./Downloads/linux-2.6.35.7.tar.bz2 /usr/src/
mmmary@mmmary-laptop:~ $ cd /usr/src
mmmary@mmmary-laptop:/usr/src $ sudo tar -xvf linux-2.6.35.7.tar.bz2
```

A symbolic link should be created as to be allow to create scripts and (if necessary) only change the kernel.

```
mmmary@mmmary-laptop:/usr/src $ sudo ln -s linux-2.6.35.7/ linux
```

### 6.2 Configuring the kernel

Before configuring and compiling the kernel, it is necessary to have some packets installed.

```
mmmary@mmmary-laptop:~ $ sudo aptitude install build-essential libncurses5-dev
```

If some of the packets cannot be found, first update your system and retry to install them. Have your system up to date by typing:

```
mmmary@mmmary-laptop:~ $ sudo aptitude update
```

A makefile is used to compile the kernel. The first time it takes some time, but afterwards only the the changed files and their dependencies have to be compiled.

```
mmmary@mmmary-laptop:/usr/src $ cd linux
mmmary@mmmary-laptop:/usr/src/linux $ sudo make menuconfig
```

The congestion default algorithm is going to be changed from Cubic to Reno. Cubic is a derivation of Binary increase Congestion Control (BIC) which tries to find the maximum of the cwnd by using a binary search algorithm. Here, the cwnd is a cubic function of time since the last congestion event. Cubic is used by default in Linux kernels since version 2.6.19. In Reno (in this case New Reno), during fast recovery, for every duplicate ack that is returned to TCP a new unsent segments from the end of the congestion window is sent, to keep the transmit window full. To perform the change, go to: Networking support -> Networking options -> TCP: advanced congestion control -> Default TCP congestion control, change the option and save. See Fig. 6.1.
6.3 Implementing the Divacks mechanism

6.3.1 Modifications in the client’s side

The following modifications only concern the kernel that is deployed in the client computer. The client will be downloading data from the server.

6.3.1.1 Divacks parameters

Four system parameters need to be implemented as to modify in runtime the behaviour of the Divacks mechanism. These are:

- **tcp_divack**: This is the number of divacks sent per data packet, also defined as n in the previous chapter. This parameters allow only natural numbers, from 0 to SMSS - 1:
  - 0: the divacks technique is deactivated. One full-ack is sent.
  - 1: the acknowledgement is divided in two parts: one divack and the full-ack.
  - 2: the acknowledgement is divided in three parts: two divacks and the full-ack.
  - … until SMSS - 1.

![Screenshot of the menuconfig. Selection of the congestion control strategy.](image)
• **tcp_divack_controlled**: This variable indicates which variation is used. There are only two possible values:

  - 0: the Brute Force Divacks mechanism is used
  - 1: the Controlled Divacks mechanism is used. In this case the following two variables need to be set.

• **tcp_divack_count**: This variable is increased once every time a divack is sent during a TCP connection.

• **tcp_divack_max_count**: This indicated the maximum number of divacks that can be sent per TCP connection. Therefore, when tcp_divack_count = tcp_divack_max_count the Divacks mechanism is inhibited and only full acks are sent.

These variables need to be declared before using them. They are going to be declared in the net/ipv4/tcp_input.c file. The sysctl_ prefix is used as to indicate that is a system control parameter.

```
mem@mem-laptop:/usr/src/linux$ sudo gedit net/ipv4/tcp_input.c
```

Add the following lines (98):

```c
int sysctl_tcp_divack__read_mostly;
int sysctl_tcp_divack_controlled__read_mostly;
int sysctl_tcp_divack_count__read_mostly;
int sysctl_tcp_divack_max_count__read_mostly;
```

The next step is to modify the ctrl_table ipv4_table[ ] defined in the net/ipv4/sysctl_net_ipv4.c file. In this table the name of the variable in the system control tree (procname), the address of the variable that contains the data (data), the size of the variable (maxlen), the permissions (mode) and the function that handles it (proc_handler), are defined. In this case, the procname is the name of the tcp_divack variable in the /proc/sys/net/ipv4 directory and the variable that contains the data is sysctl_tcp_divack which has the size of an integer, the owner of the file is root (with writing and reading permissions and only reading permission for the group and other users (0644)) and the proc_handler will be assigned to proc_doitvec which writes integer numbers from and to the user’s buffer. The four variables are added in the same fashion. Open the file to edit:

```
mem@mem-laptop:/usr/src/linux$ sudo gedit net/ipv4/sysctl_net_ipv4.c
```

Add the following lines in the ipv4_table[ ] (line 493): (624)
6.3 Implementing the Divacks mechanism

In the include/linux/sysctl.h file, the system control interfaces are defined. The sysctl variables are added via an enumeration. Open the file:

```
mmmary@mmmary-laptop:/usr/src/linux $ sudo gedit include/linux/sysctl.h
```

Add the following lines (428):

```
1 NET_TCP_divack=126,
2 NET_TCP_divack_count=127,
3 NET_TCP_divack_max_count=128,
4 NET_TCP_divack_controlled=129,
```

The variables need to be included to the bin_net_ipv4_table[] struct of the kernel/sysctl_binary.c file where the modification function (CTL_INT for int variables), the sysctl just defined and the procname are indicated.
Add the following lines (421):

```c
{CTL_INT, NET_TCP_divack, "tcp_divack" },
{CTL_INT, NET_TCP_divack_count, "tcp_divack_count" },
{CTL_INT, NET_TCP_divack_max_count, "tcp_divack_max_count" },
{CTL_INT, NET_TCP_divack_controlled, "tcp_divack_controlled" },
```

Finally the sysctl variables must be added (as external) to the rest of the sysctl TCP variables.

Add the following lines (250):

```c
extern int sysctl_tcp_divack;
extern int sysctl_tcp_divack_count;
extern int sysctl_tcp_divack_max_count;
extern int sysctl_tcp_divack_controlled;
```

Now that the required variables to execute the Divacks mechanism are defined, the necessary changes to send the divacks can be included.

### 6.3.1.2 Sending Divided acknowledgements

The Divacks mechanism is implemented in the `__tcp_ack_snd_check()` function, where the decision of sending a new ack (and divacks if enabled) or waiting for a delayed ack, is made. The ack sequence number is saved in the `rcv_nxt_original` variable and it won’t send it right away, `sysctl_tcp_divack` will be subtracted and it enters a loop where if the acknowledgement sequence number is smaller than `rcv_nxt_original` the acknowledge is sent and the sequence number is incremented in one. Equation 6.1 indicates the number of bytes that are acknowledged by the first divack.

\[
FirstDivack_{BytesToAcknowledge} = (packets\ size - sysctl\tcp\_divack)\ bytes
\]  

(6.1)

The successive divacks are only going to acknowledge one byte from the packet until sending the full-ack that acknowledges the last byte of the packet. If the `sysctl_tcp_divack` parameter is zero (0) the ack sequence number won’t be changed, the loop won’t take place and only the full-ack is sent (recognizing the whole packet). Once the loop is finished there would have been sent as many divacks as indicated in the `sysctl_tcp_divack` parameter. In Code 6.1, a pseudo code of the Divacks mechanism is shown.
6.3 Implementing the Divacks mechanism

/* tcp_ack_snd_check() function sending divacks in the tcp clients side*/

rcv_nxt_original = ack_sequence_number;
ack_sequence_number = -sysctl_tcp_divack;

while (ack_sequence_number < rcv_nxt_original) {
    if ((sysctl_tcp_controlled &&
        (sysctl_tcp_divack_count < sysctl_tcp_divack_max_count)) || (!sysctl_tcp_controlled)) {
        sendDivack(ack_sequence_number);
        sysctl_tcp_divack_count++;
    }
    ack_sequence_number++;
}
sendFullAck(ack_sequence_number);

Code 6.1 Pseudocode: Sending divacks in the clients side
6.3.1.3 Example: sending divacks iteration

A detailed example of an iteration on the code is explained in this section. The number of divacks send per packet is set to three \((tcp\_divack = 3)\) and the mechanism chosen is Brute Force \((tcp\_controlled = 0)\). A packet arrives and its sequence number is 1900 (value to acknowledge). These are the steps that correspond to executing the code presented in Code 6.1:

1. \(\text{ack\_sequence\_number} = 2000\)
2. \(\text{rcv\_nxt\_original} = 2000\)
3. \(\text{ack\_sequence\_number} = 2000 - 3 = 1997\)
4. As \((\text{ack\_sequence\_number} < \text{rcv\_nxt\_original})\)
   (a) As Brute Force mechanism is enabled
      i. \(\text{sendDivack}(1997)\)
   (b) \(\text{ack\_sequence\_number} = 1998\)
5. As \((\text{ack\_sequence\_number} < \text{rcv\_nxt\_original})\)
   (a) As Brute Force mechanism is enabled
      i. \(\text{sendDivack}(1998)\)
   (b) \(\text{ack\_sequence\_number} = 1999\)
6. As \((\text{ack\_sequence\_number} < \text{rcv\_nxt\_original})\)
   (a) As Brute Force mechanism is enabled
      i. \(\text{sendDivack}(1999)\)
   (b) \(\text{ack\_sequence\_number} = 2000\)
7. \(\text{sendFullAck}(2000)\)
6.3 Implementing the Divacks mechanism

Back to the implementation, the \_\_tcp\_ack\_snd\_check() function needs to be modified. Open the file:

```
mmmary@mmmary-laptop:/usr/src/linux $ sudo gedit net/ipv4/tcp_input.c
```

Add the following changes to implement the Divacks mechanism:

```c
static void \_\_tcp\_ack\_snd\_check(struct sock \*sk, int ofo_possible)
{
    struct tcp_sock \*tp = tcp_sk(sk);
    \/* Divacks mechanism */
    u32 rcv_nxt_original = tp->rcv_nxt;

    \/* More than one full frame received... */
    if (((tp->rcv_nxt - tp->rcv_wup) > inet_csk(sk)->icsk_ack.rcv_mss &&
        \/* ... and right edge of window advances far enough. */
        \/* tcp_recvmsg() will send ACK otherwise. Or... */
        \/*
        \_\_tcp_select_window(sk) >= tp->rcvWnd) ||
        \/* Each frame is ACK or... */
        tcp_in_quickack_mode(sk) ||
        \/* We have out of order data. */
        (ofo_possible && skb_peek(tp->out_of_order_queue)))
    {
        tp->rcv_nxt = sysctl_tcp_divack;
        \/* sending divacks loop */
        while(tp->rcv_nxt < rcv_nxt_original){
            \/*If controlled, check the limit has not been reached before sending divacks. If not controlled (= Brute Force), send divacks. */
            if((sysctl_tcp_divack_controlled && (sysctl_tcp_divack_count <
                sysctl_tcp_divack_max_count)) || (!sysctl_tcp_divack_controlled)){
                tcp_send_ack(sk);
                sysctl_tcp_divack_count++;
            }
            tp->rcv_nxt++;
        }
        \/* Send the full ack*/
        tcp_send_ack(sk);
    }else{
        \/* Else, send delayed ack. */
        tcp_send_delayed_ack(sk);
    }
}
```
Chapter 6  Implementation of the Divacks mechanism

37
38

Code 6.2 Modifications on the kernel to send divacks

Another modification has to be done. It enables sending in the announced window (rwnd) the size of the available space of the reception buffer. To do so, the limitation that restricts the available space that the client can announce, must be removed from the \_tcp\_select\_window() function. This allows the Divacks mechanism to work. Open the file:

```
mmmary@mmmary-laptop:/usr/src/linux $ sudo gedit net/ipv4/tcp_output.c
```

And comment the following line (1910):

```
/* if (free_space > tp->rcv_ssthresh)
   free_space = tp->rcv_ssthresh; */
```

6.3.2 Modifications in the server’s side

There are some modifications that need to be done only in the kernel of the computer that will be working as the divacks server. The divacks server will receive the divacks from the client, and when this happens the cwnd is going to be updated. Since the Linux kernel is protected against the Divacks mechanism, the size of the congestion window must be enabled to be changed by the mechanism. Consequently, this behaviour must be enabled. The tcp\_ack() function form the net/ipv4/tcp_input.c file has to be changed.

```
mmmary@mmmary-laptop:/usr/src/linux $ sudo gedit net/ipv4/tcp_input.c
```

Comment the following validations:

```
if (tcp_ack_is_dubious(sk, flag)) {
    /* Advance CWND, if state allows this. */
    /* Divacks comment if ((flag & FLAG_DATA_ACKED) && !frto_cwnd) */
    if (frto_cwnd &&
        tcp_may_raise_cwnd(sk, flag))
        tcp_cong_avoid(sk, ack, prior_in_flight);
    tcp_fastretrans_alert(sk, prior_packets - tp->packets_out, 
        flag);
} else {
    /* Divacks comment if ((flag & FLAG_DATA_ACKED) && !frto_cwnd) */
    if (!frto_cwnd)
        tcp_cong_avoid(sk, ack, prior_in_flight);
}
```
Another validation that restricts the growth of the cwnd needs to be commented, as to allow it’s growth with each divack received. Modify the tcp_reno_cong_avoid() function from the net/ipv4/tcp_cong.c file.

```
void tcp_reno_cong_avoid(struct sock *sk, u32 ack, u32 in_flight)
{
    struct tcp_sock *tp = tcp_sk(sk);
    /* Divacks comment */
    if (!tcp_is_cwnd_limited(sk, in_flight))
        return; /*
```

6.4 Compilation of the modified kernel

The following steps should be executed in both computers, the client and server. The first time the kernel is compiled, it takes a lot of time. To take advantage of the number of processors the computer has, the make command should include the -jn option, where n is the number of processors. If the modified kernel has a mistake, the system might not be able to boot. If so, you just have to boot from the last working version available in the booting list.

```
mmmary@mmmary-laptop:/usr/src/linux $ sudo make -jn
```

After this step is done, the kernel modules must be installed.

```
mmmary@mmmary-laptop:/usr/src/linux $ sudo make modules_install -jn
```

And the kernel is installed.

```
mmmary@mmmary-laptop:/usr/src/linux $ sudo make install
```

The boot files need to be created, and since a Linux kernel v 2.6.35.7 is being installed, and therefore this must be indicated. If this command is run a second time, instead of -c (create) it should be -u (update).

```
mmmary@mmmary-laptop:/usr/src/linux $ cd ..
mmmary@mmmary-laptop:/usr/src/ $ sudo update-initramfs -c -k 2.6.35.7
```

The modules are installed in the /lib/modules/2.6.35.7 directory; the kernel and the initrd file, in the /boot/vmlinuz-2.6.35.7 and /boot/initrd.img-2.6.35.7 respectively. The boot manager needs to be updated as to allow to choose the modified kernel in the boot list.
Chapter 6 Implementation of the Divacks mechanism

Now that the divacks client and server are working, tests can be performed to evaluate the behaviour of the mechanism. To do so, the sysctl kernel parameters (defined in section 6.3.1.1) can be modified in runtime. To read the value of a variable, `tcp_divack_max_count` in this example:

```
mmmary@mmmary-laptop:~ $ sudo sysctl net.ipv4.tcp_divack_max_count
```

And to modify this parameter, the following command can be used (it sets its value to 1500):

```
mmmary@mmmary-laptop:~ $ sudo sysctl -w net.ipv4.tcp_divack_max_count="1500"
```
Chapter 7

Experimentation and Evaluation

Tests were performed in Labo4g at TELECOM Bretagne campus, to measure the Divacks mechanism effectiveness. The results obtained are published in this chapter. In the first place, the testbed, tools and configuration used, are introduced. The influence of the different variables that play a role in the mechanism’s output have been measured and quantified in section 7.4. By taking them into account, the values for which the mechanism presents a gain regarding the default TCP mechanism were found. The different variations of the Divacks mechanism and their behaviour on mobile environments are introduced in sections 7.5 and 7.6 respectively. Conclusions are given in section 7.7.

7.1 Testbed configuration

Firstly the Divacks mechanism was implemented in two computers: one which is the divacks client and a second one, the divacks server. The divacks client is a Asus-W5fe Sideshow notebook (2.00 GHz Intel Core 2 Duo T7400 processor and 1 GB of RAM). The divacks server is a Dell Latitude D410 notebook (1.86 GHz Intel Pentium M processor and 489 MB of RAM). Both computers have Ubuntu 10.04 as operating system and gcc version 4.4.3 installed. A 2.6.35.7 Linux kernel was downloaded, modified (each one with its respective modifications), compiled and deployed to both computers (see chapter 6 for more details). A third computer, here called netem computer (3 GHz Intel Pentium 4 CPU processor and 992MB of RAM) was used to emulate network conditions, in this case the RTT (Round Trip Time). An access point was also used in the deployment to provide a wireless link where handovers took place. This AP is a Linksys Wireless-G Broadband Router, WRT54GL model (2.4 GHz, 64 Mbps).
The network topology, shown in Fig. 7.1, consists on the two divacks computers: the divacks server "A" and the divacks client "C", the netem one: "B" and the access point. The server computer is connected to the netem computer via an Ethernet cable, and this computer is also connected to the access point via a second Ethernet cable. The client connects via a wireless link to the AP. In this testbed, the divacks client "C" will be downloading data from the divacks server "A". When enabled, the client will acknowledge every IP packet received with several divacks, speeding up the transfer.

As to measure the speed of the different interfaces, UDP traffic was emulated with **iperf**

![Testbed for divacks testing.](image)

(see section 7.2). The results shown in Table 7.1, reveal that the bottleneck of the transmission is in the wireless link of the topology, which can reach a maximum speed of 16.2 Mbps.

<table>
<thead>
<tr>
<th>Link</th>
<th>Measured bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server-Netem</td>
<td>100 Mbps</td>
</tr>
<tr>
<td>Netem-Client</td>
<td>19.5 Mbps</td>
</tr>
<tr>
<td>Server-Client</td>
<td>16.2 Mbps</td>
</tr>
</tbody>
</table>

**Table 7.1** Testbed interfaces speed, measured with iperf.
7.2 Tools

Several tools were used in order to measure and configure the testbed; execute, measure and obtain the results. The tools used are now listed.

- **GNU wget** is a computer program that allows the exchange of files between computers. To do so, the server computer needs to have Apache HTTP Server installed as to work as a web server. It was used to exchange files between the *divacks server* and the *divacks client*. To download the testFile.txt file from the server, the following command needs to be run in the client’s side:

  ```
  mmary@labo4g-desktop:~ $ sudo wget -0 file.txt http://192.168.11.10/testFile.txt
  ```

- **iperf** is a network testing tool that can create TCP and UDP data streams while measuring the throughput of the network that is carrying them. It was used to measure the topology interfaces speeds.

  Usage on the client’s side:

  ```
  mmary@labo4g-desktop:~ $ iperf -c <<servers address>> -u
  ```

  Usage on the server’s side:

  ```
  mmary@labo4g-desktop:~ $ iperf -s -u
  ```

- **netem** is a Network Emulation functionality for testing protocols, which emulates the properties of wide area networks. Netem was used in the *netem computer* as to introduce a delay in the network. For instance, the command used to introduce a delay of 250 ms in the eth0 interface is:

  ```
  mmary@labo4g-desktop:~ $ tc qdisc add dev eth0 root netem delay 250ms
  ```

- **R** is a free software programming language and a software environment for statistical computing and graphics. It was used to process and graph the results displayed in the following sections.

- **TCP probe** is a module that records the state of a TCP connection in response to incoming packets. It was used in the server’s side to capture the congestion window size during the transmissions. Before the transmission is established, in the server’s side, these commands need to be executed:
Once the transfer is finished, to save and kill the capture, the following command need to be executed (on the sever’s side):

```
mmmary@labo4g - desktop:~\$ sudo kill \$TCPCAP
```

- **Wireshark** is a free and open-source network protocol analyzer for Unix and Windows. It was used to sniff network traffic and obtain detailed information on the exchanged packets between the divacks client and server.

### 7.3 Testing conditions

To perform the following tests, some assumptions must be arisen:

- TCP is used as the transport layer protocol during the transfer.
- The client will be downloading data from the server.
- The handover is a layer 2 hard horizontal handover.
- The TCP connection is not interrupted by the handover.
7.4 Testing configuration parameters

In this section the factors and parameters that have a measurable impact on the performance of the Divacks mechanism are going to be explained and the results of the tests while varying these values are shown. Even if the nature of these factors is different (some are related to the network configuration, others to the TCP configuration and others to the Divacks mechanism) they all have an influence on the TCP transfer. These parameters are listed in Table 7.2 which also contains the values used to evaluate them.

<table>
<thead>
<tr>
<th>Testing Configuration parameters</th>
<th>Used Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of divacks sent per received data packet</td>
<td>0, 3, 6, 9, 12, 15</td>
</tr>
<tr>
<td>File size</td>
<td>500, 990 KB; 1, 2, 4 and 8 MB</td>
</tr>
<tr>
<td>Round Trip Time</td>
<td>10, 70, 125, 250 and 500 ms</td>
</tr>
<tr>
<td>Buffer’s Size</td>
<td>default, 4x default, 8x default (see Table 7.6)</td>
</tr>
<tr>
<td>Divacks mechanism</td>
<td>Brute Force, Controlled</td>
</tr>
<tr>
<td>Handover</td>
<td>FALSE, TRUE</td>
</tr>
</tbody>
</table>

Table 7.2 Parameters and its used values to test the Divacks mechanism’s performance.

7.4.1 Number of Divacks per data packet

The number of divacks sent per received packet of data is probably one of the most important, if not the most, parameter in the evaluation of the Divacks mechanism. This is due to its direct impact on the mechanism’s performance. It has also been called $n$ in this document (see chapter 5), and it is set via the tcp_divacks system variable (see chapter 6: Implementation, section 6.5). To test the impact that this parameter has, Test No. 1 was performed (see Table 7.3 for detailed information about the configuration) where only the number of divacks was changed while the other parameters remained constant. In this case, only the Brute Force Divacks mechanism was used. When the number of divacks sent per data packets is equal to zero, it means the the default TCP mechanism takes place, sending the full-ack solely.
<table>
<thead>
<tr>
<th>Testing configuration parameters</th>
<th>Test 1: Number of divacks per data packet</th>
<th>Test 2: Transfer’s size</th>
<th>Test 3.1 and 3.2: Round Trip Time</th>
<th>Test 4.1 and 4.2: Buffer’s size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of divacks per data packet</td>
<td>0, 3, 6, 9, 12, 15</td>
<td>0, 12</td>
<td>0, 12</td>
<td>0, 12</td>
</tr>
<tr>
<td>File size</td>
<td>990 KB</td>
<td>500 KB, 1, 2, 4 and 8 MB</td>
<td>3.1: 990 KB</td>
<td>4.1: 990 KB</td>
</tr>
<tr>
<td>Round Trip Time</td>
<td>250 ms</td>
<td>250 ms</td>
<td>3.1: 10, 70, 125, 250 and 500 ms</td>
<td>4.1: 990 KB, 1, 2, 4 and 8 MB</td>
</tr>
<tr>
<td>Buffer’s Size</td>
<td>default</td>
<td>default</td>
<td>default</td>
<td>default, 4xdefault, 8xdefault</td>
</tr>
<tr>
<td>Divacks mechanism</td>
<td>Brute Force</td>
<td>Brute Force, Controlled</td>
<td>3.1: Brute Force</td>
<td>4.1: Brute Force, Controlled</td>
</tr>
<tr>
<td>Handover</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

Table 7.3 Configuration used for the executed Tests 1-4:
The impact of the testing environment and divacks parameters on the mechanism’s performance.
When increasing the number of divacks sent per received IP packet, the server receives a larger number of acknowledgements, making the size of it’s cwnd to grow in a more aggressive fashion, as can be seen in Fig. 7.2, than if the default TCP mechanism was enabled \( (n = 0) \).

The growth of the cwnd, allows more packets to be transferred in a time moment. This has an impact on the speed in which the TCP segments are received by the receiver. In Fig. 7.3 the sequence number of the received data has been plotted for all the different numbers of divacks sent per data packet. It shows that increasing the divacks number makes the transfer to speed up since the available bandwidth is discovered faster (the transfer is finished earlier when increasing the value of \( n \)). But, performing several repetitions of this test has shown that there is a limit: while using \( n = 15 \) the average transfer length of time is larger than when \( n = 12 \), see Table 7.4 and also Fig. 7.3.

<table>
<thead>
<tr>
<th>Number of divacks per data packet</th>
<th>Average transfer time (s)</th>
<th>Std of the transfer time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3.12</td>
<td>0.19</td>
</tr>
<tr>
<td>3</td>
<td>2.82</td>
<td>0.35</td>
</tr>
<tr>
<td>6</td>
<td>2.60</td>
<td>0.16</td>
</tr>
<tr>
<td>9</td>
<td>2.62</td>
<td>0.14</td>
</tr>
<tr>
<td>12</td>
<td>2.42</td>
<td>0.20</td>
</tr>
<tr>
<td>15</td>
<td>3.05</td>
<td>0.13</td>
</tr>
</tbody>
</table>

**Table 7.4** Several runs of Test No. 1: Average transfer time per number of divacks sent per packet received.

Increasing the number of divacks does not assure that the Divacks mechanism performs better by making the transfer time shorter, since there are other TCP factors that need to be taken into account (see section 5.2). From the results shown in this subsection, it can be said that there is an optimal number of divacks to send per data packet, and this value is between 7 and 14. The number of divacks chosen to perform the following tests is going to be 12, since not only it has shown a shorter transfer time but also is more stable (smaller standard deviation).
Chapter 7 Experimentation and Evaluation

Figure 7.2 Test No. 1: Congestion window values for different values of divacks sent.

7.4.2 Size of the transfer

The amount of data packets that the server needs to send to the client, influences on the performance of the mechanism. Whenever the size of the file transferred is increased, more packets need to be sent (because of Ethernet’s MTU of 1500 bytes) and if the Divacks mechanism is enabled, specially the Brute Force variation, more divacks are going to be sent. When this occurs, it has been shown that the transfer is limited due to the ping pong effect.

To clarify this, Test No. 2 was performed, where the size of the file to be transferred was varied between 500 KB and 2 MB (see Table 7.3 for more details on the configuration). There is a constraint regarding the size of the transfer as Fig. 7.4 shows. In the presented topology, the transfers in which the performance of divacks overtakes the default mechanism
7.4 Testing configuration parameters

Figure 7.3 Test No. 1: Sequence number measured in the client; s side for different values of divacks sent.

are those whose file’s size is roughly smaller than 1 MB. In the other cases, the default TCP mechanism performs better by having a bigger throughput. For this reason a 990 KB file has been chosen to perform the majority of the following tests.

This test also shows that the Divacks mechanism needs to be controlled. In the case of the larger transfer (with a file of 8 MB), when using the Brute Force Divacks mechanism and sending 12 divacks per data packet, retransmissions can be observed. These retransmissions are due on the fact that sending divacks constantly, saturates the wireless link; which reduces the throughput, seen as smaller slope in the transfer.
Figure 7.4 Test No. 2: Sequence number measured in the clients side for different sizes of files exchanged: 500 KB, 1, 2, 4 and 8 MB.
7.4.3 Round Trip Time

The influence that the round trip time of the network has on the divacks performance has already been introduced in [3]. This delay of the network needs to be addressed in order to see how it might affect the Divacks mechanism performance. In Tests No. 3 the RTT of the network was modified while the throughput was evaluated. A first test, Test No. 3.1, the RTT was changed between 10, 70, 125, 250 and 500 ms, while a 990 KB file was transferred. The output shown in Table 7.5 shows that there is a crossing point under which the Divacks mechanism performs worse than default TCP does. When the RTT is 250 ms or larger (in this case, 500 ms was tested), the Brute Force Divacks mechanism performs the transfer faster (2.53 ms and 17.8 ms) than the default TCP mechanism (3.28 ms and 25.08 ms).

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Round Trip Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10 ms</td>
</tr>
<tr>
<td>Default TCP</td>
<td>1.4</td>
</tr>
<tr>
<td>Divacks Brute Force</td>
<td>3.33</td>
</tr>
</tbody>
</table>

Table 7.5 Test No. 3.1: Transfer time for a 990 KB file, while variating the RTT.

A second test was performed, Test No. 3.2, in which the RTT was set to 500 ms and the file’s size was variable. The results displayed in Fig. 7.5 indicate that, since there is a bigger delay (if compared with results of Fig. 7.4 where \( RTT = 250 \text{ ms} \)) in the network, the limit in which the Divacks mechanism starts loosing against the default TCP, is increased.

The RTT chosen to perform the majority of the tests is going to be 250 ms, because as [3] shows, when the value of RTT is higher than 100 ms, the Divacks mechanism has more stable results regarding its throughput. As has been said before, this value of RTT is valid as long as the file being transferred has a size smaller than 1 MB.
Figure 7.5 Test No. 3.2: Sequence number measured in the clients side for different sizes of files exchanged: 500 KB, 1, 2, 4 and 8 MB, when RTT = 500 ms.
7.4 Testing configuration parameters

7.4.4 Divacks emission

When the Divacks mechanism is enabled, divacks might be sent whenever a data packet is received depending on the chosen Divacks variation. As Fig. 7.6 shows, the Controlled Divacks mechanism only sends divacks in an aggressive way in the first two RTTs; after this, the Divacks mechanism is disabled and the acknowledgements are sent in the legacy TCP way (one acknowledgement per data IP packet). On the other hand, when the Brute Force mechanism is enabled, divacks are sent as long as the transfer lasts.

Results shows not only that the Controlled Divacks mechanism ends the transfer before the Brute Force one, but also the difference between the number of divacks emission speed. The Ping Pong effect is put in evidence showing that increasing the number of divacks sent could, and mostly will, degrade the performance by increasing the traffic between the links.

![Figure 7.6 Test 2: Divacks and full-acks emission speed by 0.1 s intervals.](image)

7.4.5 Reception buffer’s size

The client’s reception buffer plays an important role in the performance of the Divacks mechanism. This buffer is opened for each TPC connection and it stores all incoming packets before delivering them to the client. Therefore, if the buffer’s size is increased the amount of packets that can be stored, and not discarded because of congestion, increases; increasing the throughput of the transmission.
Linux includes a system variable, called `net.ipv4.tcp_rmem` that allows to modify the size of the reception buffer. It has three values: minimum, default and maximum size. The first value tells the kernel the minimum receive buffer size for each TCP connection; this size is always going to be allocated to a TCP socket, even under high pressure on the system. The second value specified tells the kernel the default receive buffer allocated for each TCP socket. The third and last value specified in the maximum variable indicates the maximum reception buffer size that can be allocated for a TCP socket. The default values in the divacks client are shown in Table 7.6, which also includes the reception buffer’s sizes used for testing.

<table>
<thead>
<tr>
<th>Buffer’s size</th>
<th>min</th>
<th>default</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>default size</td>
<td>4096</td>
<td>87380</td>
<td>3244032</td>
</tr>
<tr>
<td>4x default size</td>
<td>16384</td>
<td>349520</td>
<td>12976128</td>
</tr>
<tr>
<td>8x default size</td>
<td>32768</td>
<td>699040</td>
<td>25952256</td>
</tr>
</tbody>
</table>

Table 7.6 Test No. 4: Buffers’ sizes (Bytes) used for testing.

goodput (it’s configuration is described in Table 7.3). In Test No. 4.1 a 990 KB file was downloaded by the divacks client. Observe in Fig. 7.7 the transfers performed, when the buffer size is increased four or eight times it’s original value (and the Divacks mechanism is enabled), the throughput grows. Another way of measuring the gain is by calculating the throughput for the interval [1, 2] s, see Table 7.7. For the case in which the buffer grows form the default value to 8x the default value, the Divacks mechanism jumps from 4.10 to 5.67 Mbps. It also shows that the throughput is similar for both the 4x and the 8x default buffer’s size, in this interval. Therefore is recommended that the default buffer’s size is incremented when running the Divacks mechanism.

Another test was performed, Test No. 4.2, in which the file size was modified while the receiver’s buffer size was set to four times it’s default value. The results, Fig. 7.8, show three things: on the first place, that the size of the file exchanged should be maintained under 1 MB so that the Divacks mechanism overtakes the default TCP mechanism. Secondly, that the existing retransmissions when the 8 MB file was transmitted (when the buffer size was the default one (see Test No. 2, Fig. 7.4) and Brute Force Divacks mechanism enabled), are not present because the buffer is now able to store the packets that before where lost.
Finally, it seems that for big files (4 and 8 MB) the Brute Force mechanism has a better performance than the controlled one.

Whenever the size of the reception buffer is increased, the size of the receiver’s announced window (rwnd) increases. A bigger rwnd also alters the distribution of the delayed acknowledgements. The TCP delayed acknowledgment is a technique used by some implementations of TCP in an effort to improve network performance. In essence, several ack responses may be combined together into a single response, reducing protocol overhead. Observe Fig. 7.9; when the receiver buffer’s size is increased 8x it’s default value, 75% of the acknowledgements sent are acknowledging one data packet. In the other cases, roughly 50% of the acks are acknowledging one data packet, and the other half is acknowledging more. This is due to the induced delay of acks in the receiver by the long backlog produced by divacks in the receiver.
Table 7.7 Test No. 4.1: Transfer throughput between 1 and 2 s for different reception buffer's sizes.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Bytes received by $t = 1$ s</th>
<th>Bytes received by $t = 2$ s</th>
<th>Throughput (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 0$, default buffer</td>
<td>4344</td>
<td>131768</td>
<td>0.97</td>
</tr>
<tr>
<td>$n = 12$, default buffer</td>
<td>63746</td>
<td>602402</td>
<td>4.10</td>
</tr>
<tr>
<td>$n = 12$, 4x default buffer</td>
<td>209994</td>
<td>945578</td>
<td>5.61</td>
</tr>
<tr>
<td>$n = 12$, 8x default buffer</td>
<td>171690</td>
<td>915474</td>
<td>5.67</td>
</tr>
</tbody>
</table>

interface. This backlog is making the receiver to trigger more frequently the delayed acks timer to generate the following acks faster.

7.4.6 Receiver’s announced window (rwnd)

As already explained in section 3.2.1, the amount of data packets that the server is going to send to the client during the slow start algorithm, is the minimum between $rwnd$ and $cwnd$ (Equation 3.4). Since the value of the receiver’s announced window affects the transmission output, it was decided to measure this impact. Since the size of the $rwnd$ is determined by the size of the reception buffer and the transfer speed, the value of this window was captured while changing the size of the reception buffer (Test No. 4.1).

The results in Fig. 7.10 show how narrow the relationship between the $rwnd$ and the throughput is: the growth of the reception buffer size, increments the size of the $rwnd$ and this allows the growth of the throughput (plotted in Fig. 7.7). The sooner the value of $rwnd$ increases, the faster the transfer at a given time moment is, which also makes the overall transfer faster. In particular, when the size of the reception buffer was set to 8 times the default value and Divack was enabled, between 1.5 and 2.3 seconds the increment of the $rwnd$ is noticeable and also the corresponding behaviour can be appreciated in the rate of transmission of Fig. 7.7.
Figure 7.8 Test No. 4.2: Sequence number measured in the client’s side for 4x default value of the reception buffers size and different files’ sizes.

7.4.7 Evaluation of the testing configuration parameters

A more clear view on this subject is going to be explained in this section. While performing Test No. 4.1 the values of both cwnd and rwnd, were registered at the end of each round trip cycle. Also, the number of data packets transferred by round trip cycle was calculated. Four tests were performed:

- Default buffer size and default TCP mechanism \((n = 0)\). Results are in Table 7.8,

- Default buffer size and Brute Force Divacks mechanism \((n = 12)\). Results are in Table 7.9,

- 4x buffer size and Brute Force Divacks mechanism \((n = 12)\). Results are in Table 7.10 and
There are several conclusions that can be arisen from these results. There are going to be explained next.

1. **Increasing the number of divacks sent, increases the goodput, until a limit is reached.** In Test No. 1, files were exchanged while changing the number of divacks sent per data packet. The values were varied between 3, 6, 9, 12 and 15. Results show that the cwnd increases while this number does and therefore so does the goodput. But there is a maximum in \( n = 12 \); if a larger number is used, the Divacks mechanism under-performs the default TCP mechanism.

2. **The RTT value and the size of the file transferred determines if whether the Divacks mechanism shows a gain over default TCP or not.** Tests No. 2, 3.1 and 3.2 show that the RTT need to be at least 250 ms and that the file size to be downloaded from the server by the divacks client, must be smaller than 1 MB; so as that the Divacks mechanism performs the transfer in a shorter amount of time.

3. **Increasing the receiver’s buffer size enhances the advantages of the Divacks mechanism.** When the receiver’s buffer size is increased and the Divacks mechanism is enabled, throughput (in a given interval) has jumped from 0.97 Mbps for default TCP to 5.67 Mbps. Not only is this increment is considerable, but also it has retransmissions (for 8 MB files) eliminated (by allowing more packets to be stored) and an increment on the rwnd (which commands the data rate).
4. **The size of the cwnd when using divacks could be considered as infinity.**

When the transfer was performed with the default TCP mechanism, the largest cwnd size obtained was $330 \times 1448$ bytes in round trip cycle eleven. In this case, the value of cwnd is of the same order of magnitude that the one of rwnd. When the Divacks mechanism is enabled, the size of cwnd reaches $5014 \times 1448$ bytes for the default reception buffer size, and almost $5500 \times 1448$ bytes for 4x and 8x buffer size.

5. **The throughput is mainly limited by the rwnd.** When the value of the cwnd overtakes the value of the rwnd, because of Equation 3.6, the number of packets sent by round trip cycle is kept under the value of the rwnd. This happened during the whole transfer (in all the four cases); except for the two first round trip cycles in which the cwnd is smaller ($3 \times 1448$ for $RT \ cycle = 1$ and $41 \times 1448$ for $RT \ cycle = 2$) and therefore limits the output.

**Figure 7.10** Test No. 4.1: Receiver’s announced window (rwnd) measured in the client’s side for different reception buffers’ sizes.
6. **The size of rwnd depends on the reception buffer size** Whenever the size of the reception buffer is increased, the size of the receiver’s announced window has a more abrupt expansion. When Divacks is enabled, for the default case, the rwnd grows from its initial value (45 * 1448 bytes) to its maximum (439 * 1448 bytes) in eight round trip cycles. Meanwhile, in the case when the buffer is increased four times, the rwnd grows from 181 * 1448 bytes to 373 * 1448 bytes in six round trip cycles. This behaviour is even more noticeable when the buffer is eight times the default size: the rwnd grows from 360 * 1448 bytes to 747 * 1448 bytes, the double size, in the same time (six round trip cycles). As a consequence, the increment of the reception buffer size, as it enables more data packet to be received (by enlarging rwnd), makes the throughput increase: more packets are sent in the firsts round trip cycles when the buffer size is bigger.

7. **There is a pattern in the emission of the data packets.** In all cases, the number of data packets received per round trip cycle increases in regard to the ones sent in the previous round trip cycle, until a maximum is reached (at RT cycle = 5 for the cases where Divacks is enabled), moment after which the number of data packets received, starts decreasing.

8. **The Ping Pong effect inhibits the growth of the throughput.** Table 7.10, shows that whenever the rwnd size increases, there is a noticeable increase of the throughput. In the following round trip cycles where the size of rwnd remains constant, the throughput eventually decreases. In RT cycle = 3 rwnd grows and the amount of packets sent per round trip cycle increases from 42 to 101, and for RT cycle = 6 (rwnd grows again), the amount of packets sent grows from 97 to 144. However, before experiencing a new rwnd increase, the throughput decreases abruptly (at RT cycle = 5 and RT cycle = 8). This is due to the ping-pong effect on the WLAN access. There are bursts of increasing throughput for which TCP does better depending on rwnd. Observing Table 7.10, at the beginning of an increment of the receiver’s announced window (rwnd) size there is an noticeable increase of the throughput. This corresponds to the recently liberated data packets by the sender, i.e., by passing from rwnd to 2 * rwnd (see rwnd = 180 to rwnd = 383 SMSS). However, at the end of the transmission period, i.e., before experiencing a new rwnd increase, the throughput decreases due to the high number of data packets and Divacks to transmit.

The effect of the Ping Pong effect can also be appreciated in Fig. 7.3 when several numbers of divacks per data segment was tested. In this test, it is noticeable the decrease in the goodput when n = 15 because the huge number of divacks is overwhelming the
shared link.
<table>
<thead>
<tr>
<th>Round Trip cycle</th>
<th>Time (s)</th>
<th>cwnd size (*1448 bytes)</th>
<th>rwnd size (*1448 bytes)</th>
<th>Packets sent per interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.25 to 0.5</td>
<td>3</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0.50 to 0.75</td>
<td>5</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>0.75 to 1</td>
<td>11</td>
<td>44</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>1 to 1.25</td>
<td>23</td>
<td>44</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>1.25 to 1.50</td>
<td>45</td>
<td>44</td>
<td>46</td>
</tr>
<tr>
<td>6</td>
<td>1.50 to 1.75</td>
<td>72</td>
<td>50</td>
<td>72</td>
</tr>
<tr>
<td>7</td>
<td>1.75 to 2</td>
<td>117</td>
<td>94</td>
<td>101</td>
</tr>
<tr>
<td>8</td>
<td>2 to 2.25</td>
<td>184</td>
<td>102</td>
<td>102</td>
</tr>
<tr>
<td>9</td>
<td>2.25 to 2.5</td>
<td>277</td>
<td>159</td>
<td>96</td>
</tr>
<tr>
<td>10</td>
<td>2.5 to 2.75</td>
<td>330</td>
<td>210</td>
<td>210</td>
</tr>
<tr>
<td>11</td>
<td>3 to end</td>
<td>330</td>
<td>289</td>
<td>28</td>
</tr>
</tbody>
</table>

**Table 7.8** Test No. 4.1: Cwnd, rwnd, and throughput for the transfer of a file while using default buffer and default TCP mechanism (n = 0).

<table>
<thead>
<tr>
<th>Round Trip cycle</th>
<th>Time (s)</th>
<th>cwnd size (*1448 bytes)</th>
<th>rwnd size (*1448 bytes)</th>
<th>Packets sent per interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.25 to 0.5</td>
<td>3</td>
<td>45</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0.50 to 0.75</td>
<td>41</td>
<td>44</td>
<td>41</td>
</tr>
<tr>
<td>3</td>
<td>0.75 to 1</td>
<td>287</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>4</td>
<td>1 to 1.25</td>
<td>635</td>
<td>108</td>
<td>108</td>
</tr>
<tr>
<td>5</td>
<td>1.25 to 1.50</td>
<td>1376</td>
<td>108</td>
<td>91</td>
</tr>
<tr>
<td>6</td>
<td>1.50 to 1.75</td>
<td>2214</td>
<td>212</td>
<td>126</td>
</tr>
<tr>
<td>7</td>
<td>1.75 to 2</td>
<td>3178</td>
<td>218</td>
<td>142</td>
</tr>
<tr>
<td>8</td>
<td>2 to 2.25</td>
<td>3995</td>
<td>439</td>
<td>75</td>
</tr>
<tr>
<td>9</td>
<td>2.25 to end</td>
<td>5014</td>
<td>439</td>
<td>67</td>
</tr>
</tbody>
</table>

**Table 7.9** Test No. 4.1: Cwnd, rwnd, and throughput for the transfer of a file while using default buffer and Brute Force Divacks mechanism enabled (and n = 12).
### 7.4 Testing configuration parameters

<table>
<thead>
<tr>
<th>Round Trip cycle</th>
<th>Time (s)</th>
<th>cwnd size (*1448 bytes)</th>
<th>rwnd size (*1448 bytes)</th>
<th>Packets sent per interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.25 to 0.5</td>
<td>3</td>
<td>181</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0.50 to 0.75</td>
<td>41</td>
<td>178</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>0.75 to 1</td>
<td>587</td>
<td>180</td>
<td>101</td>
</tr>
<tr>
<td>4</td>
<td>1 to 1.25</td>
<td>1513</td>
<td>180</td>
<td>131</td>
</tr>
<tr>
<td>5</td>
<td>1.25 to 1.50</td>
<td>2422</td>
<td>180</td>
<td>97</td>
</tr>
<tr>
<td>6</td>
<td>1.50 to 1.75</td>
<td>3578</td>
<td>373</td>
<td>144</td>
</tr>
<tr>
<td>7</td>
<td>1.75 to 2</td>
<td>4540</td>
<td>373</td>
<td>136</td>
</tr>
<tr>
<td>8</td>
<td>2 to 2.25</td>
<td>5437</td>
<td>373</td>
<td>47</td>
</tr>
<tr>
<td>9</td>
<td>2.25 to end</td>
<td>5492</td>
<td>373</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 7.10** Test No. 4.1: Cwnd, rwnd, and throughput for the transfer of a file while using 4x default buffer and Brute Force Divacks mechanism enabled (and n = 12).

<table>
<thead>
<tr>
<th>Round Trip cycle</th>
<th>Time (s)</th>
<th>cwnd size (*1448 bytes)</th>
<th>rwnd size (*1448 bytes)</th>
<th>Packets sent per interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.25 to 0.5</td>
<td>3</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0.50 to 0.75</td>
<td>41</td>
<td>-</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>0.75 to 1</td>
<td>587</td>
<td>362</td>
<td>75</td>
</tr>
<tr>
<td>4</td>
<td>1 to 1.25</td>
<td>1484</td>
<td>361</td>
<td>135</td>
</tr>
<tr>
<td>5</td>
<td>1.25 to 1.50</td>
<td>2487</td>
<td>361</td>
<td>214</td>
</tr>
<tr>
<td>6</td>
<td>1.50 to 1.75</td>
<td>2988</td>
<td>361</td>
<td>82</td>
</tr>
<tr>
<td>7</td>
<td>1.75 to 2</td>
<td>4061</td>
<td>362</td>
<td>83</td>
</tr>
<tr>
<td>8</td>
<td>2 to 2.25</td>
<td>5137</td>
<td>747</td>
<td>68</td>
</tr>
<tr>
<td>9</td>
<td>2.25 to end</td>
<td>5453</td>
<td>748</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 7.11** Test No. 4.1: Cwnd, rwnd, and throughput for the transfer of a file while using 8x default buffer and Brute Force Divacks mechanism enabled (and n = 12).
7.5 Evaluating the Divacks mechanism

As told in section 5.3, two variants of the Divacks mechanism have been implemented: the **Brute Force** mechanism and the **Controlled** enhancement. In the first case, divacks are sent as long as the slow start algorithm (see section 3.2.1) is being executed. In the second case, divacks are sent while slow start but only until the number of divacks sent reaches a threshold set in the `tcp_divack_max_count` system control variable. Test were performed to define this maximum, which was set to 1500 divacks per TCP connection.

Previous tests (see Fig. 7.6) have already shown what is introduced thoughtfully in this section: the necessity of a controlled mechanism that preserves the gain obtained without congesting the shared link.

The configuration used to execute the tests displayed in this section, is shown in Table 7.12. Three tests were performed in which a file was exchanged:

- Default TCP mechanism,
- Brute Force Divacks mechanism and
- Controlled Divacks mechanism.

<table>
<thead>
<tr>
<th>Testing configuration parameters</th>
<th>Testing Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of divacks</td>
<td>0, 12</td>
</tr>
<tr>
<td>File size</td>
<td>990 KB</td>
</tr>
<tr>
<td>Round Trip Time</td>
<td>250 ms</td>
</tr>
<tr>
<td>Buffer’s Size</td>
<td>default</td>
</tr>
<tr>
<td>Mechanism</td>
<td>Brute Force, Controlled</td>
</tr>
<tr>
<td>Controlled threshold</td>
<td>1500 divacks</td>
</tr>
<tr>
<td>Handover</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

**Table 7.12** Values used to test the variants of the Divacks mechanism: Brute Force and Controlled divacks.

Once the three tests were executed, the sequence number of the data segment transferred was plotted in Fig. 7.11, which shows that the Controlled Divacks mechanism has a better performance. There are three reasons why. Firstly, even if both divacks variants overtake
7.5 Evaluating the Divacks mechanism

the throughput of the default TCP mechanism (the transfers are finished in a shorter period of time), Table 7.13 (where the lifespan of a transfer has been measured for several iterations of this test) shows not only that the length of the transfer for the Controlled mechanism is in average smaller but also it’s standard deviation is limited. The second reason why the Controlled variant is chosen over the Brute Force one, is that in the last case the Ping Pong effect degrades the result of the transmission. Since divacks are going to be sent all the time when using the Brute Force mechanism, the wireless link ends up congested and the throughput is reduced. The effect that it has on the transmission can be appreciated in Fig. 7.11, in which the transfer speed between 2.2 and 2.6 s is substantially reduced (the slope has a valley) when enabling the Brute Force mechanism. Finally, as Fig. 7.4 has shown, the nature of the Brute Force mechanism not only reduces the throughput but it also might introduce retransmissions; which in a long term, derive in a worse performance than if Divacks was not enabled at all.

Figure 7.11 Variation of divacks Mechanism: Brute Force and Controlled. Sequence number measured in the clients side.
### Table 7.13

Testing the Divacks mechanisms: Transmission lifespan for the Brute Force and the Controlled mechanism.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Average transmission time (s)</th>
<th>Std transmission time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default TCP</td>
<td>3.12</td>
<td>0.19</td>
</tr>
<tr>
<td>Divacks Brute Force</td>
<td>2.49</td>
<td>0.07</td>
</tr>
<tr>
<td>Divacks Controlled</td>
<td>2.35</td>
<td>0.05</td>
</tr>
</tbody>
</table>
7.6 Evaluating the Divacks mechanism on a mobile environment

Using the knowledge gained while performing the tests mentioned in the previous sections, the Divacks mechanism was tested in a mobile environment where layer 2 handovers were emulated. To do so, the same testbed was used and the mobility was emulated by connecting and disconnecting the wireless link between the Divacks client and the AP. The handover latency (see section 2.5.1) is not addressed in the evaluation of the mechanism (but Navas does in [21]). The handover recover latency is measured indirectly by the total transfer time.

The configuration chosen to perform the mobility tests is detailed in Table 7.14. The values chosen are the ones that have shown in previous tests that they optimize the performance of the Divacks mechanism. The transfer was performed three times, one for each mechanism:

- Default TCP mechanism,
- Brute Force Divacks mechanism and
- Controlled Divacks mechanism.

The results of exchanging a 990 KB file while a L2 handover was emulated, is shown in Fig. 7.12. Here it can be observed how both Divacks variations overtake the default TCP goodput before the handover takes place. After the handover occurs the new available rate has to be discovered. In the case of the Brute Force mechanism, after the handover occurs retransmissions take place, a behaviour which has been already introduced in the previous
The throughput obtained before and after the handover was computed and is shown in Table 7.15. As expected, when the Divacks mechanism is executed in a controlled fashion, it overtakes the default TCP behaviour not only before a handover occurs (default: 0.86 Mbps vs divacks: 1.38 Mbps) but also afterwards (default: 0.23 Mbps, divacks: 0.90 Mbps).

A second mobility test was performed with the reception buffer size incremented four times its default value. As Fig. 7.13 shows, results are similar for those obtained with the default size. The throughput before the handover is larger when Divacks is enabled but after the handover the results are somewhat different: (1) the retransmissions present when using the Brute Force variation are reduced since the buffer is now able to receive a larger amount
7.6 Evaluating the Divacks mechanism on a mobile environment

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Throughput (Mbps) before handover</th>
<th>Throughput (Mbps) after handover</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default TCP</td>
<td>0.86</td>
<td>0.23</td>
</tr>
<tr>
<td>Divacks Brute Force</td>
<td>1.34</td>
<td>0.18</td>
</tr>
<tr>
<td>Divacks Controlled</td>
<td>1.38</td>
<td>0.90</td>
</tr>
</tbody>
</table>

Table 7.15 Mobility test: Transfer throughput with default reception buffer size.

Figure 7.13 Mobility test: Sequence number measured in the client’s side for a mobile test with 4x default buffer size.

of packets at the same time and (2) the throughput of the Controlled mechanism and the TCP default mechanism is roughly different.
7.7 Discussion

Once all the factors that affect the Divacks mechanism’s performance were addressed and evaluated, the best combination of values to assign them was found. Only after this process was made, the mechanism was ready to be tested in a mobile environment.

The Divacks mechanism success over the default TCP mechanism is evident but this is only true if some cautions are taken. First, as sending divacks not only increases the data traffic but also the acknowledgement packets, there is a limit of the number of divacks that should be sent per received data packet. It was found that this variable should be set between 7 and 14. To perform the majority of the tests, $n = 12$ was chosen. If a larger number is chosen, the performance is degraded due to the Ping Pong effect. On the same direction, the size of the file to be transferred and the RTT of the network set a threshold under which the Divacks mechanism under performs. The transfer should be smaller than 1 MB when the RTT is set between 250 and 500 ms; if these conditions are not met, there is no gain over the default TCP mechanism.

The impact that the receiver’s buffer size has on the throughput is important due to two main reasons:

- It has been shown that whenever this buffer’s size is incremented, the rwnd grows allowing more data packets to be sent per round trip cycle. Therefore the transfer is finished faster, in opposition to using the default buffer’s size.

- Since the buffer allows packets to be stored before delivering them, the increase of it’s size reduces the Ping Pong effect.
Tests performed in a mobile environment were introduced in section 7.6. In this case, results are similar to those when handovers did not take place; except here it is more evident the necessity of limiting the number of divacks sent per data packet towards to avoid congestion (which is generated not only by a larger number of divacks but also a higher number of data packets).

The need of a controlled mechanism is essential for the Divacks mechanism to work. Without a controlled emission of the divacks, the gain obtained with the mechanism is lost because the data packets have to share the available bandwidth in the wireless link. Having unnecessary traffic generated by the emission of too many divacks might lead to the Ping Pong effect. On the other hand, the Controlled mechanism not only has a better performance but it also remains stable while avoiding retransmissions, which are present in the Brute Force variation.
Chapter 8

Conclusions and Perspectives

The discovery of the available bandwidth in a new connection with an AP is handled by the TCP slow start algorithm. Even though this process allows the congestion window to grow in an exponential fashion, it is not enough when recovering from a handover. In a mobile environment, the interruption in the data transfer needs to be as unnoticeable as possible so that mobility seems seamless to the user.

First, tests were performed to study how the default TCP mechanisms work and how handovers impact on the ongoing transfer. In section 2.5.2.1, tests performed show how the data transmission is interrupted when a handover takes place.

The second goal was to implement a fast-ramp up mechanism that takes advantage on how TCP works. To accomplish this, the Divacks mechanism introduced in [3] was chosen and implemented in a Linux kernel. The mechanism divides a data acknowledgement in several smaller ones. By doing so, it forces the congestion window at the server’s side to increase, action that allows more data packets to be sent.

Afterwards, tests were performed to evaluate how the different factors (some related to the testbed and others to the mechanism itself) affect the performance of the mechanism. Files were exchanged between the divacks server and the divacks client, through a wired-wireless network. Limits to the possible values of these factors are given by the restrictions that the network introduces to the performance of the mechanism. Also, the goal was to find the configuration in which the Ping Pong effect was reduced.
Since the aggressiveness of the Divacks methods degrades its own performance, an enhancement had to be done. An improved mechanism, the Controlled Divacks mechanism takes profit of the gain obtained by the emission of divacks while not overwhelming the wireless link. To do, divacks are sent in a controlled fashion: while on slow start but restricting the total amount of divacks sent. Results have shown that when the Divacks mechanism is enabled, a file transfer could reach a throughput of 4.1 Mbps, while the default TCP mechanism remained under 1 Mbps.

Tests were executed in a mobile environment where layer 2 handovers were emulated. The results show that the Controlled Divacks mechanism not only overtakes the TCP default mechanism before the handover takes place but also after the transmission is reestablished, with a throughput four times larger. The results make visible the necessity of limiting the number of divacks sent per data packet so as to achieve the avoidance of congestion (and therefore retransmissions), present when divacks are sent with out restriction.
Improvements can still be made to the mechanism. First, triggers should be defined in order to choose the period in which divacks should be emitted. The first one should indicate that a handover has occurred in order to start the emission of divacks (and not before). A second trigger should be used to cease the emission. An option is to measure the speed in which data packets are received and if this speed decreases, then full-acks should be sent instead. Secondly, it would be valuable if the mechanism was implemented in a mobile operating system like Android for example.

The need of control is essential to the Divacks mechanism. Without a controlled emission of the divided acknowledgements, the gain obtained by forcing the congestion window to grow is lost due to shearing nature of the wireless link. On the other hand, the controlled mechanism not only has a better performance but it also remains stable while avoiding retransmissions.
Bibliography


