PROVIDING A SEAMLESS ACCESS TO MOBILE USERS

by

María Ciminieri

Soutenance Report

Advisors

Dr. Andrés Arcia-Moret, International Center for Theoretical Physics, Trieste, Italy.

Dr. Nicolas Montavont, Institut TELECOM / TELECOM Bretagne - Network Security and Multimedia Department -

Rennes, France.

April 2013

ABSTRACT

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. Our interest is studying the mechanisms involved to understand and be able to implement solutions to provide seamless user mobility on this kind of networks.

In order to achieve this goal, an enhancement TCP mechanism [2], Divacks, was implemented and evaluated, especially in mobile environments.

Contents

List of Figures

Chapter 1

Introduction

Internet wireless access has become ubiquitous and its main technology nowadays corresponds to 802.11. However, this kind of access is limited to static users when they are within the range of the Wi-Fi Access Point. And thus, our interest on studying the mechanisms to understand and implement solutions to provide user mobility on this kind of networks. So far, we have observed that transport layer 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 an ongoing call or data session from the AP the mobile device is currently connected to, to another one. Whenever a handover occurs, a TCP transfer is interrupted. After this interruption the TCP sender should adjust its transmission rate to discover the rate provided by the new AP. The main objective of this recovery is then, to allow a mobile terminal running an application, experience a seamless continuity in the service. This implies that whenever a mobile station performs a handover, the running applications should have the smaller impact as possible: the delay of connecting to a new access point and the time needed to discover the available bandwidth must be as small as possible.

The first, and most general goal, is to study of the TCP mechanisms that take place when the available bandwidth needs to be discovered and how handovers impact on TCP. The second goal was to implement a fast-ramp up mechanism, the **Divacks mechanism** presented in [2], on a Linux platform. Two variations to the method were proposed and evaluated in a deployed testbed that had been adjusted as to better understand the TCP legacy mechanisms and the way the Divacks mechanism works. Afterwards, the impact of handovers on TCP while having the mechanism enabled, was measured.

This document is organized as follows: In chapter 2, 802.11 networks are described and the different types of handover will be presented, 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 we will analyse the existing solutions that handle with the discovery of the available bandwidth and that deal with handovers. In chapter 5 we will present the Divacks mechanism and it's two variations, explaining they work. A working implementation in a Linux kernel is presented in chapter 6. 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 is presented.

- **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. It is a protocol that 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

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

Table 2.1 The OSI model.

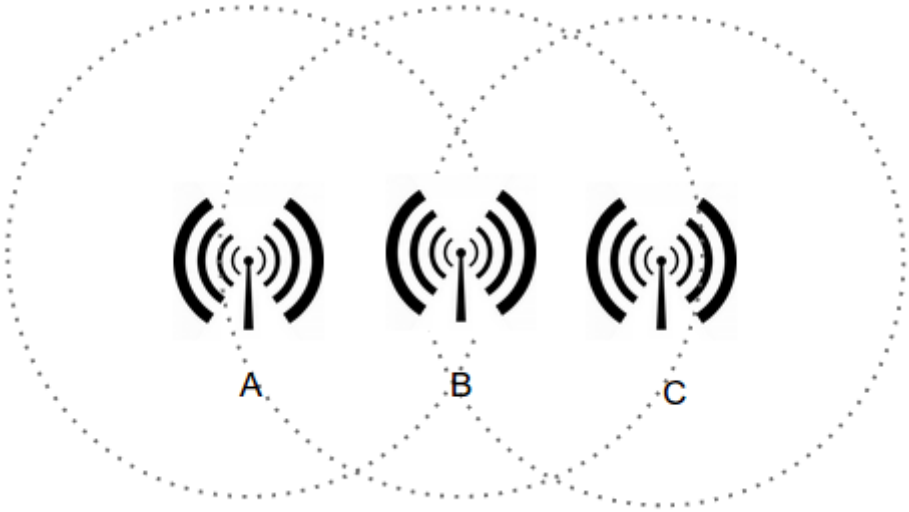

Figure 2.1 Hidden Node Problem.

other because they cannot hear one another's broadcast. Since they think that the link is available, the will start to transmit 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 mobile terminal 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, presented in [14], is when an AP is overloaded with clients, it can force the MS to change the AP it is connected to, as to alleviate the congestion.

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 will be introduced with more detail in the following section because of its direct impact on the application level.

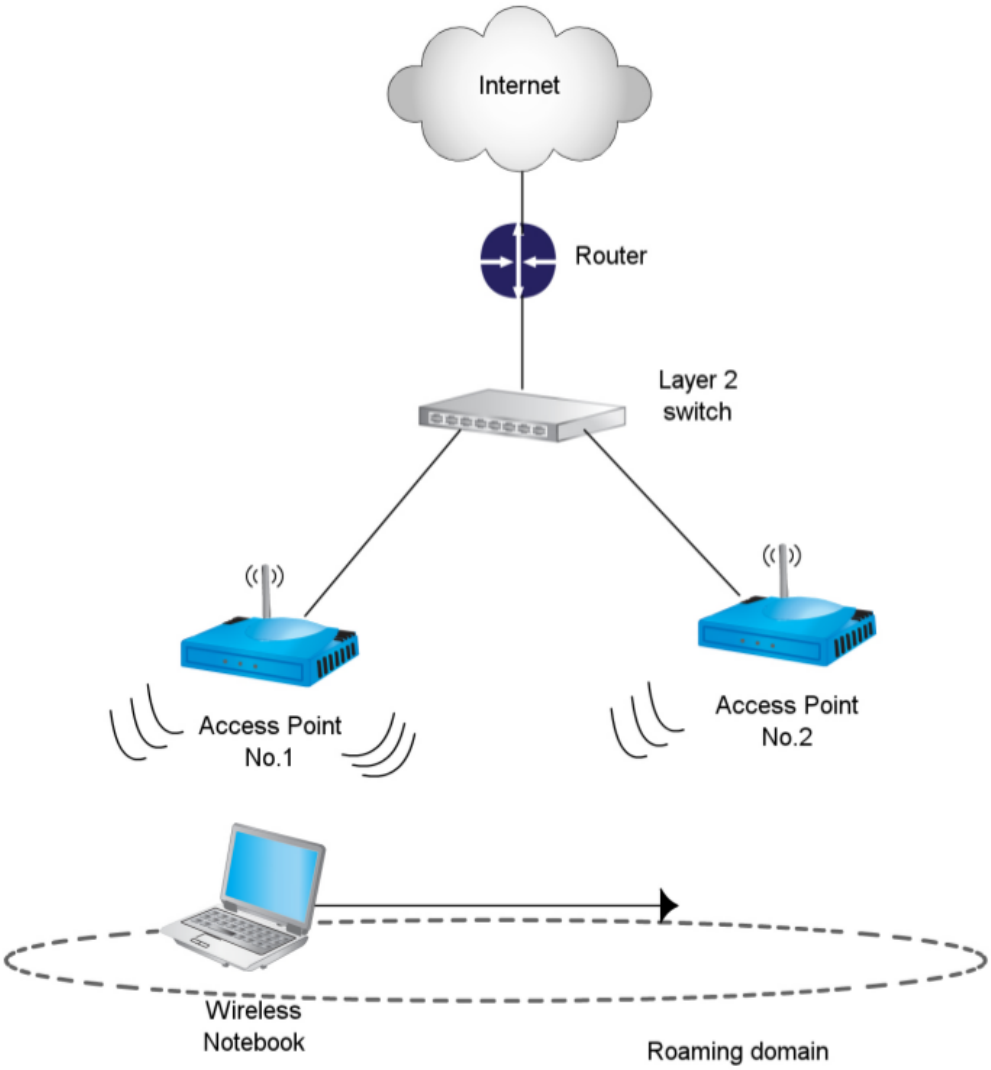

Figure 2.2 Graphical representation of a layer 2 handover

We will we working with horizontal hard layer 2 handovers.

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 will take place because of the interruption.

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 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 chapter 3: The Transmission Control Protocol, 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 presented 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 has to be managed as to reduce the impact that the applications running in the MS, suffer as small as possible. 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 transmission and the different delays that are introduced are presented. 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. Their duration will depend on the nature of the handover and the network characteristics. They are illustrated in Fig. 2.4. In this figure a transfer and its throughput is represented, in it we can see that after a handover, there is a period of rime in which the new available bandwidth needs to be discovered.

Handover latency This latency is the time introduced by a handover in which the is no data exchanged. We must reduce this latency because, since there is no data exchanged in this period of time, some of the applications running in the MS are going to feel a disconnection.

Handover recover latency This latency is the time that it takes, after a handover

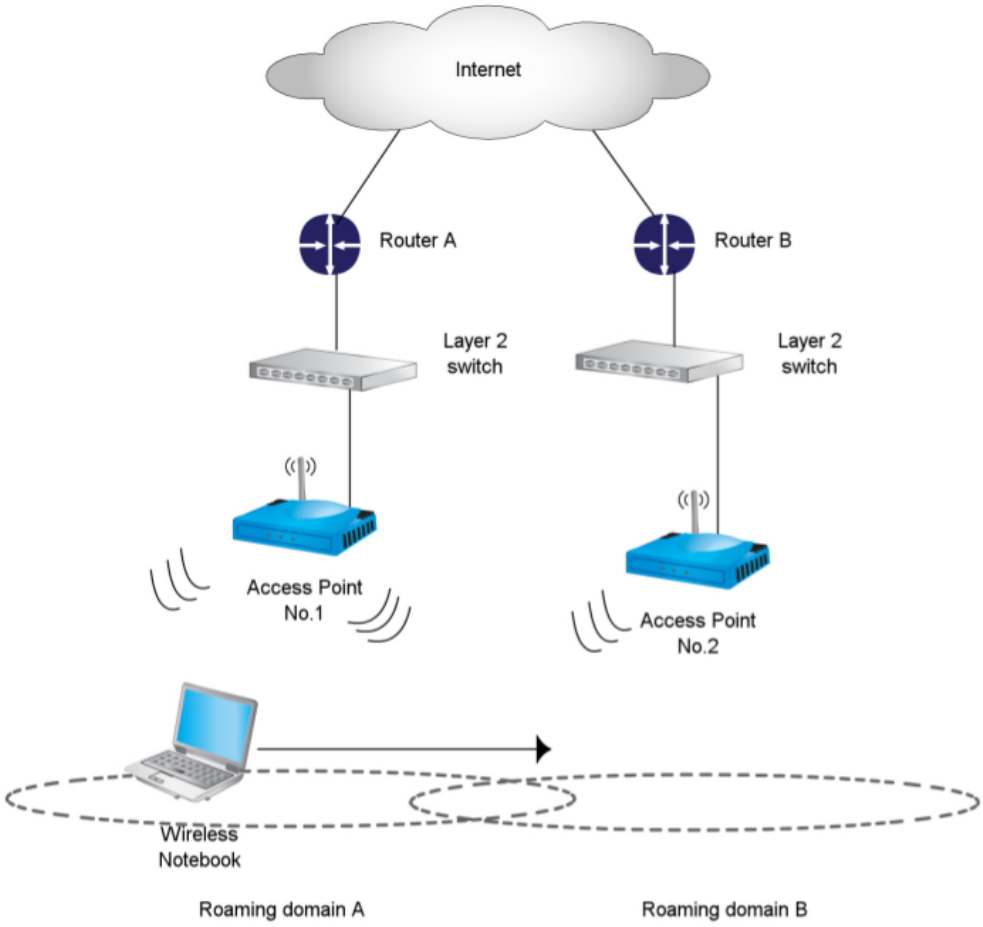

Figure 2.3 Graphical representation of a layer 3 handover

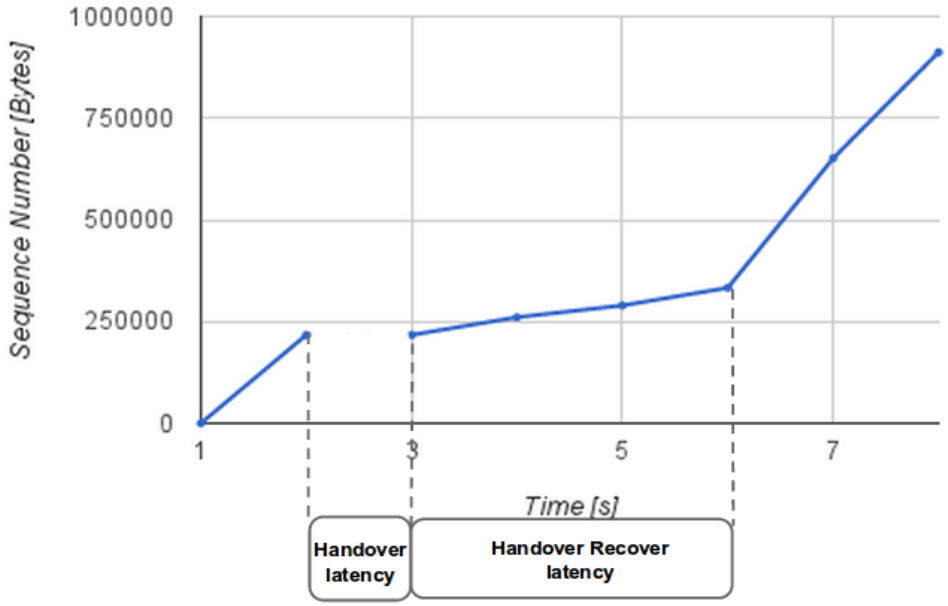

Figure 2.4 Handover's delays: Handover latency and handover recover latency.

to discover the available bandwidth. This latency should be reduced as to reach as fast as possible the new bandwidth. The TCP connection has to do so, by triggering a legacy slow start algorithm which increases the cwnd in an exponential mode. A fast ramp-up on the cwnd should reduce the impact of the typical reduction on the cwnd.

2.5.2 General effects

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

- **The TCP transferrer will be interrupted:** The data transmission will have 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* duration.

- **The TCP connection can be interrupted:** This happens when a handover involves changing of IP subnet. If the handover is managed to avoid so (ref to Mobile IP), 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 chapter: The Transmission

Control Protocol, section 3.2). It will be 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) smartphone, downloading data and performing layer 2 handovers between different APs while walking between them. The deployment is presented in Fig. 2.5. In this case the TCP connection was not interrupted.

The RSSI (Received signal strength) is the signal strength that the MS receives from the

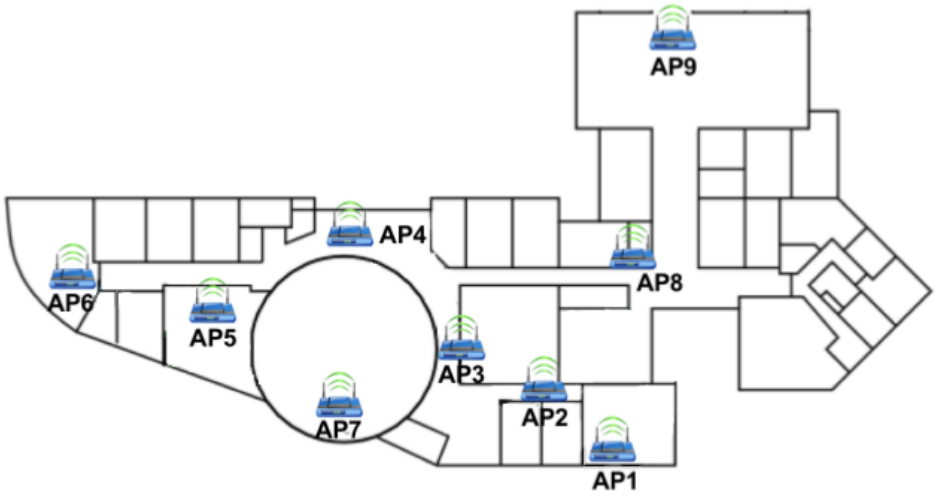

Figure 2.5 Campus AP deployment for the android handover test.

AP. This value is obtained periodically (every 500 ms), by beacons in which access points announce to all MS listening certain information about the network (and also announce the presence of a Wireless LAN). The information emitted not only provides the RSSI but also all the needed information as to allow mobile stations to associate to an access point.

In Fig. 2.6 the RSSI of the AP the Ms was connected to is plotted. In this graph there is also plotted the number of AP the MS was associated to: the right y axis represents the AP number and the red flat lines represent the connection between the MS and a certain AP. Therefore is easy to see that before a handover occurs, the RSSI decreases, indicating 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 presented. If figures 2.6 and 2.7 are compared, the interruption that a handover introduces in the data transmission, is noticeable. This comparison, can be appreciated in Fig. 2.8 where the flat periods on the data transmission (meaning that no data was exchanged) are highlighted.

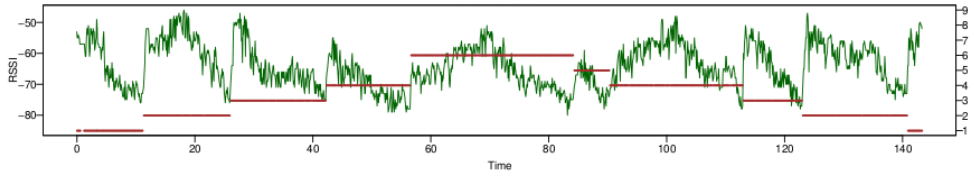

Figure 2.6 Handovers impact on the data transfer: RSSI and number of the APs to which the MS was connected to.

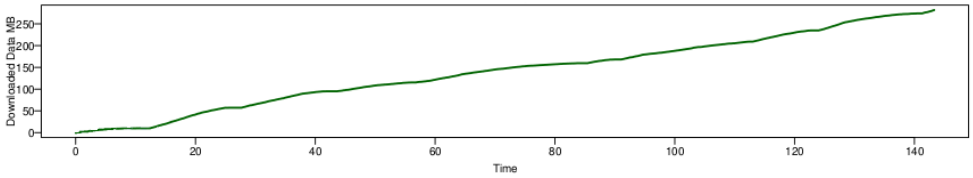

Figure 2.7 Handovers impact on the data transfer: goodput on the MS side during the test was performed.

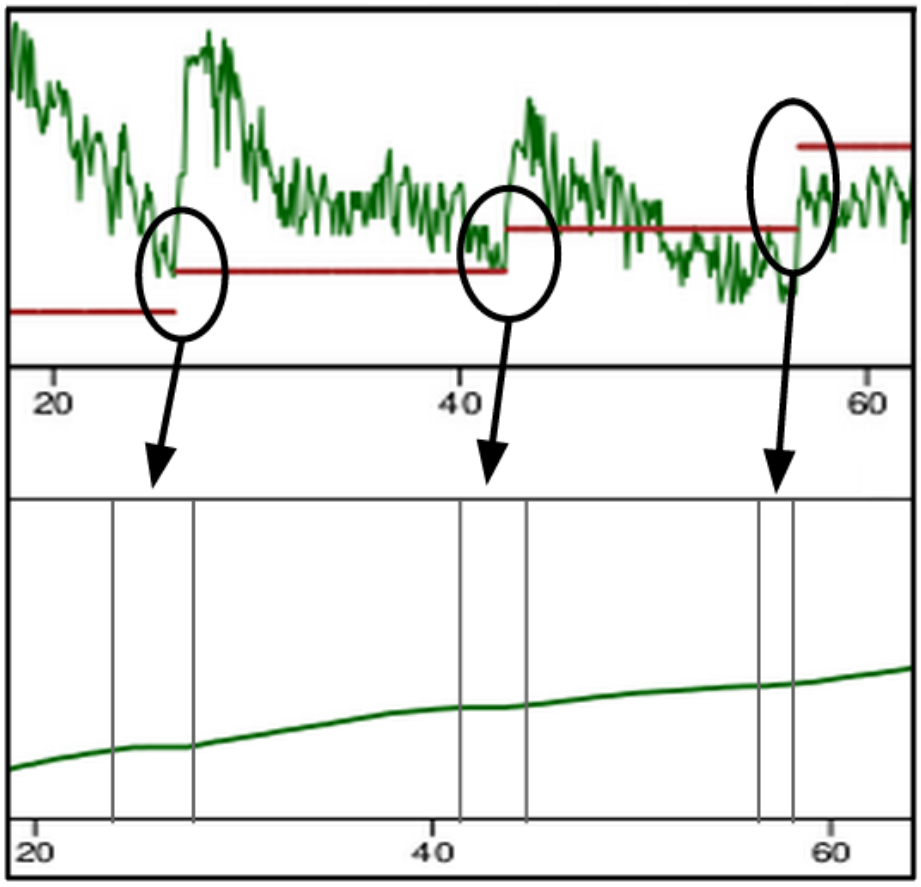

Figure 2.8 Handovers impact on the data transfer: the data transfer is paused while handovers occur.

What are the importance of these effects?

The answer to this question will strongly depend 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 is functioning as a client in a client-server application that is not strongly affected by small disconnections. On the other hand, a real time application would be more sensitive to a change in the throughput and an interruption in the data transmission.

When any kind of handover occurs, we have seen that 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 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 is allows this behaviour to take place, will be presented, together with the TCP Congestion Control strategies: slow start mechanism and congestion avoidance that manage the way the available bandwidth 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 moment of time. This system provides:

- reliability, by detecting and resending 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 retransmission) algorithm. The regular PAR algorithm needs the acknowledge 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 retransmitting the lost message. Enhanced-PAR gets rid of the timer and does not need an acknowledge to be received before sending the next packet [13]. TCP uses a sliding window acknowledge system in which bytes are divided into four different categories, see Fig. 3.1.

- **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 have 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 be received:** this is the data that the server could sent but the receiver could not be able to handle.

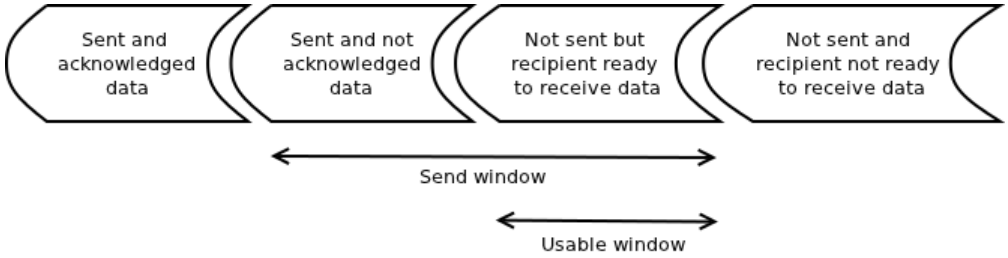

Figure 3.1 TCP sliding window system: Send and usable window.

Both the receiver and the sender keep track of this numbers for both streams of data: the data they are sending and the data they are 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 senders point of view,

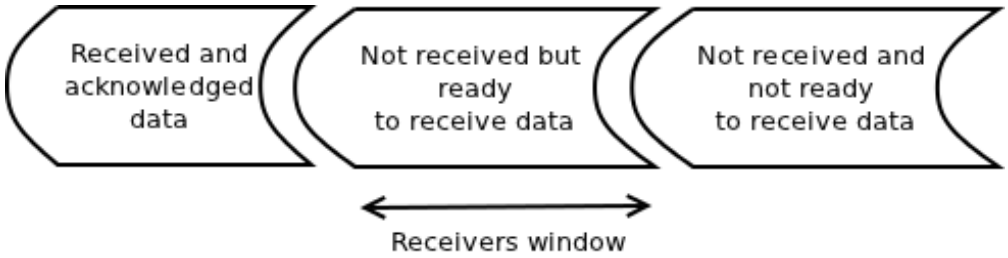

Figure 3.2 TCP sliding window system: Receiving packets,bytes categories.

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

- **Send window/ window:** its the number of bytes that the receiver allows to the sender to send unacknowledged in a period of time.

$$Send\,window\ =\ (Sent\,and\,not\,acknowledged + Not\,sent\,but\,the\,receiver\,is\,ready\,to\,receive)\,bytes \tag{3.1}$$

- **Usable window:** its the number of bytes that could be sent.

$$Usable\,window\ =\ (Not\,sent\,but\,the\,receiver\,is\,ready\,to\,receive)\,bytes \tag{3.2}$$

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 will vary depending on the available bandwidth in the connection and the TCP algorithms that are been executed. Whenever an acknowledge 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 are transferred to category one. Since the window size did not change, it will "*slide*" to the right, allowing bytes from category four to pass to category three, making the usable window change its size (depending on the number of acknowledged bytes).

We can then characterize the window by saying how it is evolving. The window *closes* whenever an acknowledgement is received. When we say that the send window is *growing*, it means that the right edge moves to the right, because there is more data that the receiver can receive in a moment of time. If the send window is *shrinking*, it means that the receiver can handle a smaller amount of data than before (the right edge is moving to the left).

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 keeps track of them by using 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.

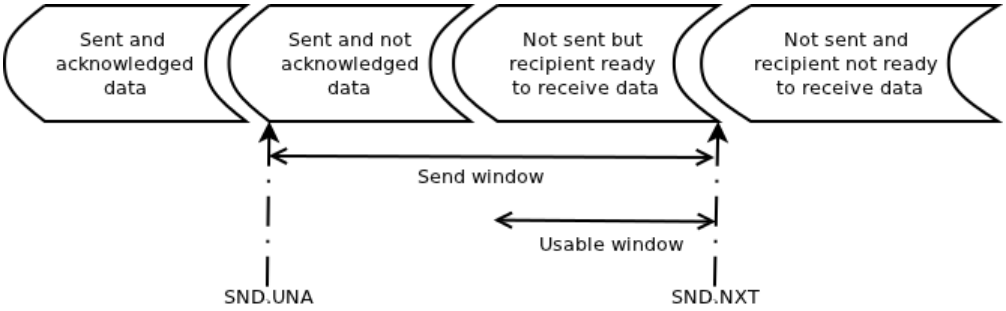

Figure 3.3 TCP sliding window system: Sender's pointers.

- **SND.WND:** indicates the number of bytes the send window is.

Therefore, the number of bytes that can be sent together in a given moment of time, defined in Equation 3.2 can be redefined by using the TCP sender's pointers, in Equation 3.3.

$$Usable\,window = (SND.UNA + SND.WND - SND.NXT)\,bytes \tag{3.3}$$

The receiver also takes track of the data it should receive from the sender. It uses two pointers:

- **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.

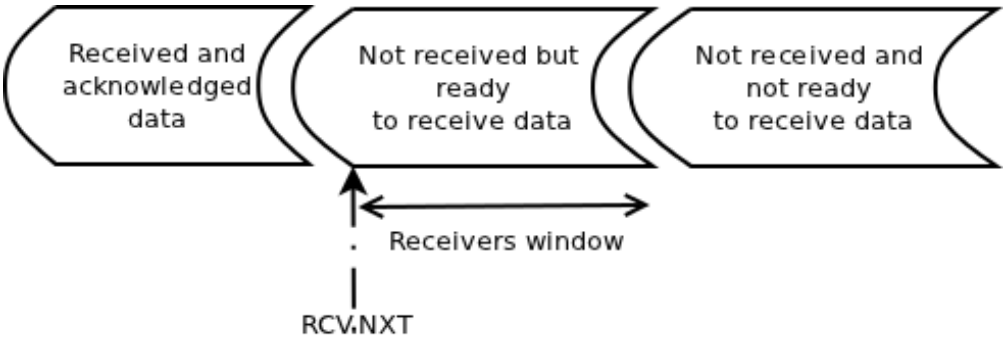

Figure 3.4 TCP sliding window system: Receiver's pointer.

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 there are packets lost or some packets in flight have not been taken into account by one of them. 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.

There are some TCP mechanisms that manage how these windows size evolve depending on the network and connection characteristics. There are two congestion control strategies (1) slow start and (2) congestion avoidance, that try that the available bandwidth is been profit but at the same time, that congestion in the receivers side is avoided. These strategies use some windows to estimate how much congestion there is between the two places. They are:

- **Congestion window (cwnd):** Is the number of outstanding bytes at a given time, which corresponds to the already defined *Send Window*.
- **Reciever'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 will always transmit the minimum value between cwnd and rwnd, as Equation 3.4 shows.

$$Effective\,window\,size\ =\ \min\{rwnd, cwnd\} \tag{3.4}$$

3.2 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. At the beginning, the value of the cwnd is equal to one MSS (maximum segment size). Therefore, the sending rate is MSS/RTT. To fast discover the available bandwidth, whenever an acknowledgement is received, the cwnd size is increased by a MSS (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 MSS so two segments are sent, when they are acknowledged, cwnd grows to 4 MSS, 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.

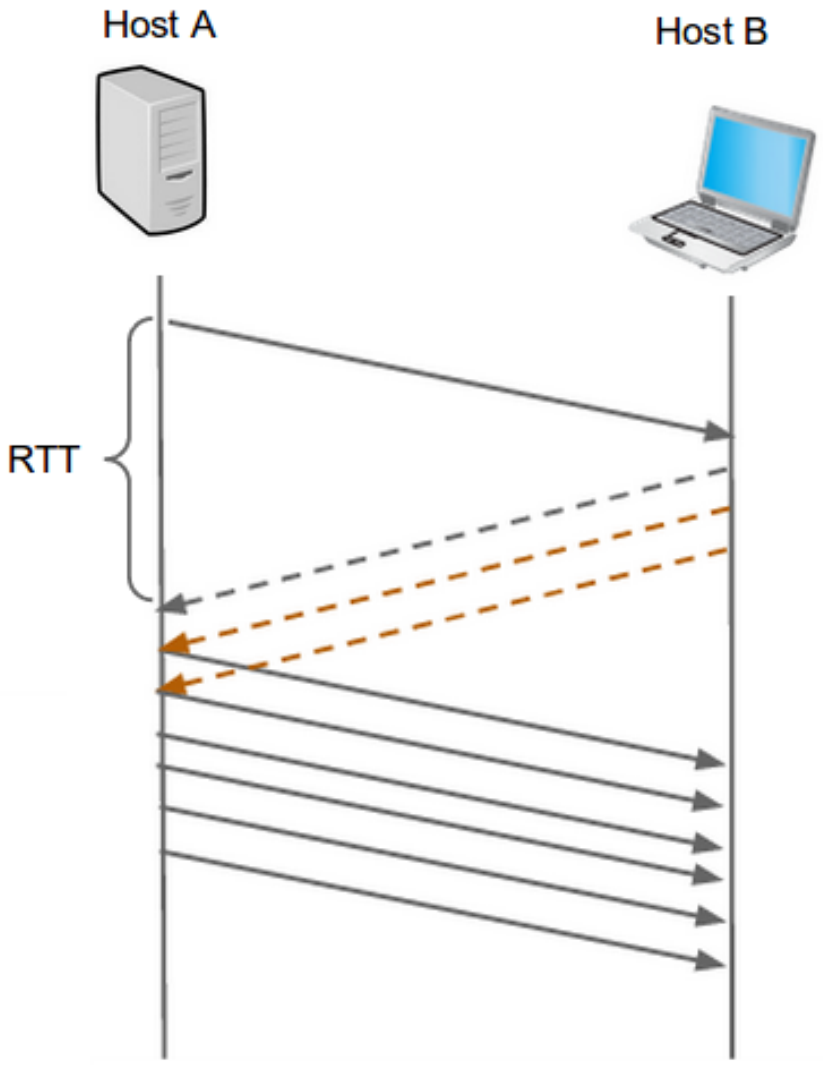

Figure 3.5 TCP slow start.

- A threshold (*ssthresh*) is reached.
- Three duplicate ack are received.

In the first case, the first time it occurs, the value of 1/2 * cwnd (half the size the window reached before the time out) is going to be stored in the *ssthresh* (slow start threshold) variable and slow start will begin again with cwnd = 1. Whenever the threshold is reached, since it is equal to half the value of cwnd when the time out occurred, the window will not be doubled (which would probably end up in congestion), but TCP will enter 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 3.4).

3.3 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 MSS per RTT will take 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 cwnd is halved (adding 3 MSS), the value of *ssthresh* is recorded as half the value of cwnd when the triple duplicate acks were received and the fast recovery state is entered.

3.4 Fast Recovery

In this state the value of cwnd is increased 1 MSS by duplicate ack received for the segment taht 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 MSS and *ssthresh* equal half the value of cwnd when the loss occurred. 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 regarding the way in which the discovery of the new available bandwidth is approached and if whether it is solution that takes place only after a handover or not.

4.1 Classification of the TCP enhancement mechanisms

The enhancements can be characterized by the way they work. Here we present three not exclusive groups, 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 will 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 mechanisms 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 we are going to introduce them by using the classification presented in the previous section.

4.2.1 Bandwidth estimation

In the case in which the network is sensed to have an estimation of the available bandwidth, several schemes have been presented.

To start with, the **Swift Start** by Partridge [18] mechanism is a solution in which the first group of data packets sent is used to estimate the bottleneck bandwidth. This estimate indicates how to scale the size of the cwnd as to take use of the available bandwidth. Equation 4.1 describes this relation.

$$Estimated\,cwnd = bottleneck\,capacity * RTT \tag{4.1}$$

A second kind of approach, takes profit of the already established connections to inform the new one, the appropriate size of cwnd that should be used for the path concerned. This new connection will not have to probe the network path and trigger the slow start algorithm. The congestion manager maintains congestion parameters and exposes an API to enable applications to learn about network characteristics, pass information to the congestion manager, and schedule data transmissions. This mechanism is called **Congestion Manager** and is presented in [7] by Balakrishnan.

A similar mechanism involves the collaboration between the end host and routers. The **Quick-start** [19] 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 mechanisms.

4.2.2 End-to-end solutions

On the other hand, there are end-to-end solutions in which collaboration between the network and its elements is not needed.

The **Jump Start** technique introduced by Lui in [15] 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. In this mechanism, retransmissions are probable but is a cost that is neglectful versus the gain obtained by almost transferring all the transfer in the first RTT. This is a sender-side change to the TCP's congestion control algorithms.

On the same direction Goff proposes **Freeze-TCP** from [10], a mechanism that modifies (by freezing) the receiver's window before a predicted disconnection happens. With this behaviour, packets are avoided to be lost. When the connection is re-established, the saved value of rwnd (from before the disconnection) is re-announced to the server, making the transmission to restart at the previous rate.

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 [9], is a solution only valid in mobile environments were handovers take place. After a handover, a time out is started after reducing the size of the congestion window. To reduce this time, 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 the time out to expire.

A second mechanism introduced by Miten in [17], called **Delayed Duplicate Acknowledgements**, proposes to delay duplicate acknowledgements so the wireless link can take care of the missing packet instead of forcing the sender to enter in unnecessary congestion control mechanisms.

4.2.2.2 Divided acknowledgement mechanisms

A group of mechanisms base their functionality in the division of the acknowledgements of data packets into several ones. By doing so, the congestion window grows allowing more data packets to be transferred.

The **Spack** mechanism presented by Jin in [12], sends several acknowledgemets to sender when there is a retransmission detected. A second mechanism called **Ack-pacing** proposed in [16] by Matsushita, also sends divided acknowledgements but only after a vertical handover takes place. In the **Ack spliting** mechanism by Hasegawa [11], there is a local estimation of the cwnd in the receiver's side as 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. The **Divacks** mechanism by Arcia Moret [2] and [5], as all the mechanisms presented in this section, divides the acknowledgements as to gain a larger throughput by forcing the growth of the cwnd.

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 [6], 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 [8], also splits the connection in two: MS to AP and AP to sender but it 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 not to be dropped by holding back the ack to the last byte.

4.3 Metrics for mechanisms evaluation

In [10], 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 senders 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, have 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 presented mechanism

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 that the Indirect TCP and M-TCP mechanisms, fail to work since they are based on the AP mediating the traffic. Mechanisms such as Congestion Manager and Quick Start that depend on the network to perform the estimation of the bandwidth, are not scalable since end-to-end solutions are preferred since only the hosts need to be modified.

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

The Freeze TCP and the Divacks [3] 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 will be addressed in detail in the next chapter: *The Divacks Mechanism*, followed by the implementation in a linux kernel in chapter 6: *Implementation of the Divacks mechanism.* Finally, the results obtained are presented in chapter 7: *Experimentation.*

Chapter 5

The Divacks Mechanism

So far, we had characterized the different kinds of handovers and their consequences. We also presented our goals regarding handovers and the impact they have on TCP. Now, we will present our proposal to handle them: 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 to grow. This growth will make that the speed in which the new available bandwidth is discovered (of the new AP) to increase. Therefore, the handover recovery latency is reduced. In this chapter, we are going to explain how the divacks mechanism operates to gain speed in the TCP transfers.

5.1 Divacks mechanism's operation

The basic idea behind the algorithm is that whenever a data packet is received, the receiver will not send one acknowledgement (as regular TCP does, see Fig. 5.1) but several ones, 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 (TCP, slow start) the value of the senders 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 acks are sent by the receiver (without any losses); the senders cwnd will be increased in a shorter period of time, allowing the sender to increase the throughput.

The basic idea behind the divacks algorithm is that whenever a data packet is received,

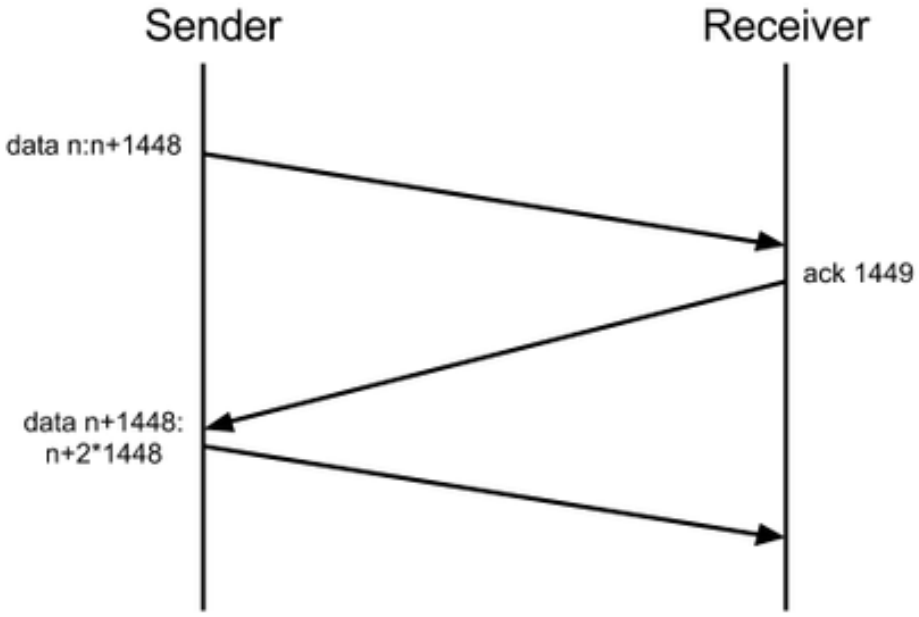

Figure 5.1 TCP acknowledgement system: one ack per data packet.

the divacks client will send n divacks (quantity defined by the user) and the full ack in a burst, see Fig. 5.2. Each one of this divacks will acknowledge a smaller amount of data than the original one (the undivided ack, called *full-ack*). The first divack will almost acknowledge all the packet. It's size is determined by Equation 6.1. In Fig. 5.2 it can be seen that the data packet received has a length of y bytes; then, the first divack will acknowledge y - *(2-1)* bytes (with an acknowledgement number = k + y - *(2 - 1)*). The next (n - 1) divacks will only acknowledge one byte: in this case is only another divack, which acknowledges y bytes with an acknowledgement number = k + y. Finally the original full-ack will be sent, acknowledging one byte and having the original acknowledgement number, $k + y + 1$.

$$FirstDivack_{BytesToAcknowledge} = (packets\,size - (n-1))\,bytes \tag{5.1}$$

The reader might be driven to think that the higher the number of divacks sent, the faster the cwnd will grow and therefore the data rate. This is not the case because there is a threshold between the number of divacks sent and the actual gain obtained in the total throughput. This limit is set by the ping pong effect presented in the following section.

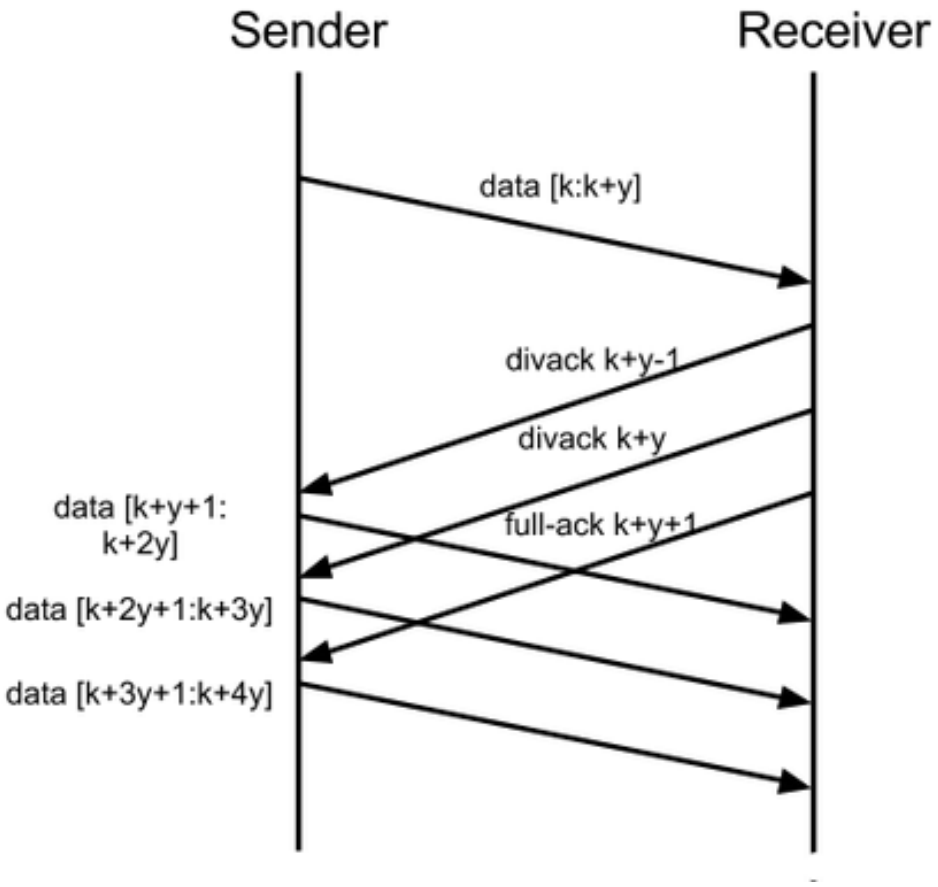

Figure 5.2 Divacks acknowledgement system: several acks per data packet, three in this example.

5.2 The Ping Pong Effect

The ping pong effect introduced in [4] is an effect 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 divacks mechanism profits by dividing one original acknowledgement into several ones, the traffic in the uplink (client to server) increases, and so does in the downlink (in response to the divacks received, the server sends more data packets). Therefore, as the number of divacks sent increases, the amount of segments in both links also does. The CSMACA protocol, presented in section 2.3, makes that the AP and the client are only enabled to 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 will take turns to send packets to each other. If the traffic increases, by sending more divacks, it is not certain that the performance of the mechanism will improve since the medium is shared. There is a limit to be found: the

maximum number of divacks to send per data packet while not falling into the ping pong effect and saturate the wireless link.

5.3 Divacks variations

Having the ping pong effect in mind is that it was decided to implement two variations of the divacks mechanism: one that sends divacks as long as the slow start TCP strategy is taking place, called *Brute Force divacks mechanism*, and a second one, called *Controlled divacks mechanism*, in which after a defined number of divacks has been sent, the divacks mechanism is inhibited and only full-acks are sent.

5.3.1 Brute Force Divacks mechanism

When the Brute Force variation in enabled, n divacks are going to be sent per data packet received when TCP is in the slow start state. This method is aggressive which does not imply, as already mentioned, that throughput will be larger.

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.

To achieve this behaviour, two parameters are needed: one to keep track of the number of divacks already sent: *tcp_ divack_ count* and another one that sets the limit: *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 in n, until *tcp_ divack_ count* reaches the limet set by *tcp_ divack_ max_ count*. When this happens, only the (original) full-acks are going to be sent.

By sending divacks only at the beginning of the transfer, the available bandwidth is discovered rapidly at take-off but it also frees the wireless link from divacks, 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. 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. Check this by typing:

```
mmmary@mmmary-laptop:~ \$ 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:

```
mmmary@mmmary-laptop:~ \$ wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.35.7.
    tar.bz2
```

Move the ziped 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 Configurating 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 installthem. Have your system up to date by typing:

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

We will use a makefile to compile the kernel. The first time it will take some time, but afterwards only the the changed files and their dependencies will have to compiled.

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

We will change the congestion default algorithm: 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. Reno (in this case New Reno) makes that 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. As to do so we must go to: Networking support ->Networking options ->TCP: advanced congestion control -> Default TCP congestion control, change the option and save. See Fig. 6.1.

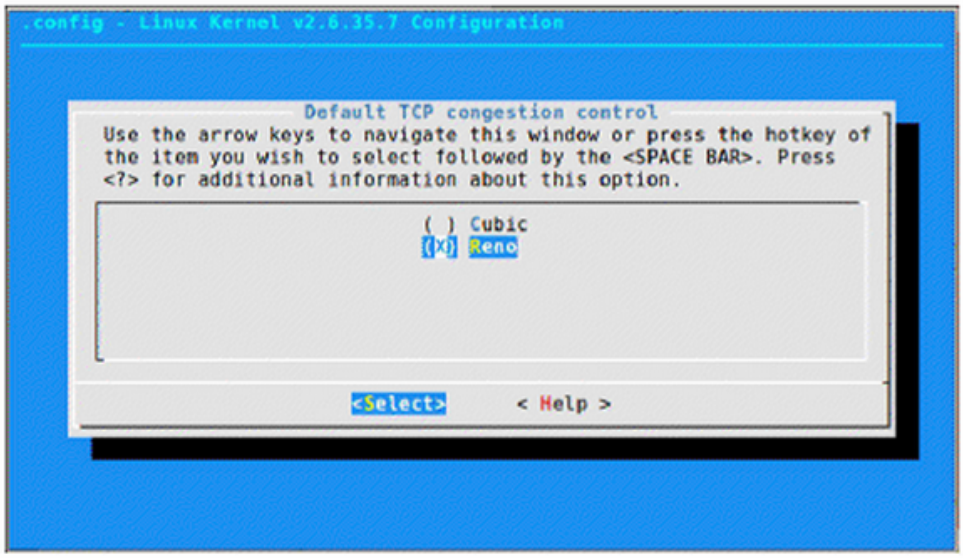

Figure 6.1 Screenshot of the menuconfig. Selection of the congestion control algorithm.

6.3 Implementing the divacks mechanism

6.3.1 Modifications in the clients side

The following modifications only concern the kernel that will be 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 presented as n in the previous chapter. This parameters will 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.

- **tcp_divack_controlled:** This variable indicates which algorithm is used. There are only two possible values:
  - 0: the Brute Force algorithm is used
  - 1: the Controlled algorithm 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. Therefore, when tcp_divack_count = tcp_divack_max_count the divacks mechanism is inhibited and only full acks are sent.

Variables need to be declared before using them. We will declare them in the net/ipv4/tcp_input.c file. The sysctl_ prefix is used as to indicate that is a system control parameter.

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

Add the following lines (98):

```
1 int sysctl_tcp_divack __read_mostly;
2 int sysctl_tcp_divack_controlled __read_mostly;
3 int sysctl_tcp_divack_count __read_mostly;
4 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*). 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 int, 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_dointvec* which will write whole numbers from and to the user's buffer. The four variables will be added in the same fashion. Open the file to edit:

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

Add the following lines in the ipv4_table[ ] (line 493): (624)

```
{       .procname       = "tcp_divack",
        .data           = &sysctl_tcp_divack,
        .maxlen         = sizeof(int),
        .mode           = 0644,
        .proc_handler   = proc_dointvec,
},
{
     .procname    = "tcp_divack_count",
     .data      = &sysctl_tcp_divack_count,
     .maxlen    = sizeof(int),
     .mode      = 0644,
     .proc_handler = proc_dointvec,
},
{
     .procname    = "tcp_divack_max_count",
     .data      = &sysctl_tcp_divack_max_count,
     .maxlen      = sizeof(int),
     .mode      = 0644,
     .proc_handler = proc_dointvec,
},
{
     .procname    = "tcp_divack_controlled",
     .data      = &sysctl_tcp_divack_controlled,
     .maxlen      = sizeof(int),
     .mode      = 0644,
     .proc_handler = proc_dointvec,
},
```

In the include/linux/sysctl.h file, the system control interfaces are defined. We need to add the sysctl variables via an enum. Open the file:

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

Add the following lines (428):

```
NET_TCP_divack = 126
NET_TCP_divack_count=127,
NET_TCP_divack_max_count=128,
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.

```
mmmary@mmmary-laptop:/usr/src/linux \$sudo gedit kernel/sysctl_binary.c
```

Add the following lines (421):

```
1 {CTL_INT, NET_TCP_divack, "tcp_divack" },
2 {CTL_INT, NET_TCP_divack_count, "tcp_divack_count" },
3 {CTL_INT, NET_TCP_divack_max_count, "tcp_divack_max_count" },
4 {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.

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

Add the following lines (250):

```
1 extern int sysctl_tcp_divack;
2 extern int sysctl_tcp_divack_count;
3 extern int sysctl_tcp_divack_max_count;
4 extern int sysctl_tcp_divack_controlled;
```

Now that the variables needed to execute the divacks mechanism are defined, we are able to make the necessary changes to send the divacks.

6.3.1.2 Sending Divided acknowledgements

The divacks mechanism is implemented in the __tcp_ack_snd_check() function, where we must decide if a new ack (and divacks if enabled) needs to be sent or wait for a delayed ack. The ack_sequence_number will be saved in the rcv_nxt_original variable and it won't send it right away, sysctl_tcp_divack will be subtracted and it will enter a loop where if the ACK sequence number is smaller than rcv_nxt_original the acknowledge is sent and the sequence number is incremented in one. Equation 6.1 presents the number of bytes that will be acknowledged by the first divack.

$$FirstDivack_{BytesToAcknowledge} = (packets\,size - sysctl\_tcp\_divack)\,bytes \tag{6.1}$$

The next 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 will be sent (recognizing the whole packet). Once the loop is done, there will have been sent as many divacks as indicated in the sysctl_tcp_divack parameter. In Code 6.1, a pseudocode of the divacks mechanism is presented.

```
1  /* __tcp_ack_snd_check() function sending divacks in the tcp clients side*/
2
3  rcv_nxt_original = ack_sequence_number;
4  ack_sequence_number -= sysctl_tcp_divack;
5
6  while ( ack_sequence_number < rcv_nxt_original){
7    if((sysctl_tcp_controlled &&
8    (sysctl_tcp_divack_count <  sysctl_tcp_divack_max_count))
9      || (!sysctl_tcp_controlled)){
10
11                    sendDivack(ack_sequence_number);
12                    sysctl_tcp_divack_count++;
13             }
14     ack_sequence_number++;
15 }
16 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 a iteration on the code is presented. The number of divacks to send per packet is set to three (tcp_divack = 3) and the algorithm chosen is brute force (tcp_controlled = 0). A packet arrives and its sequence number is 1900 (value to acknowledge). This are the steps that correspond to executing the code presented in Code 6.1:

1. ack_sequence_number = 2000
2. rcv_nxt_original = 2000
3. ack_sequence_number = 2000 - 3 = 1997
4. As (ack_sequence_number < rcv_nxt_original)
   (a) As Brute Force algorithm is enabled
      i. sendDivack(1997)
   (b) ack_sequence_number = 1998
5. As (ack_sequence_number < rcv_nxt_original)
   (a) As Brute Force algorithm is enabled
      i. sendDivack(1998)
   (b) ack_sequence_number = 1999
6. As (ack_sequence_number < rcv_nxt_original)
   (a) As Brute Force algorithm is enabled
      i. sendDivack(1999)
   (b) ack_sequence_number = 2000
7. sendFullAck(2000)

To implement the divacks algorithm __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:

```
1  static void __tcp_ack_snd_check(struct sock *sk, int ofo_possible)
2  {
3    struct tcp_sock *tp = tcp_sk(sk);
4      /* Divacks mechanism*/
5    u32 rcv_nxt_original = tp->rcv_nxt;
6
7    * More than one full frame received... */
8    if (((tp->rcv_nxt - tp->rcv_wup) > inet_csk(sk)->icsk_ack.rcv_mss &&
9         /* ... and right edge of window advances far enough.
10         * (tcp_recvmsg() will send ACK otherwise). Or...
11         */
12        __tcp_select_window(sk) >= tp->rcv_wnd) ||
13
14       /* We ACK each frame or... */
15       tcp_in_quickack_mode(sk) ||
16       /* We have out of order data. */
17       (ofo_possible && skb_peek(&tp->out_of_order_queue))) {
18
19     tp->rcv_nxt -= sysctl_tcp_divack;
20
21         /* sending divacks loop */
22     while(tp->rcv_nxt < rcv_nxt_original){
23
24       /*If controlled, check the limit has not been reached before sending
             divacks. If not controlled (= Brute Force), send divacks.*/
25       if((sysctl_tcp_divack_controlled && (sysctl_tcp_divack_count <
             sysctl_tcp_divack_max_count))
26         || (!sysctl_tcp_divack_controlled)){
27         tcp_send_ack(sk);
28         sysctl_tcp_divack_count++;
29       }
30       tp->rcv_nxt++;
31     }
32     /* Send the full ack*/
33     tcp_send_ack(sk);
34   } else {
35     /* Else, send delayed ack. */
36     tcp_send_delayed_ack(sk);
37   }
38 }
```

Code 6.2 Modifications on the kernel to send divacks

Another modification has to be done. It will allow to send in the announced window (rwnd) all the available space in 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 to allow the divacks mechanism to work. variables. Open the file:

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

And comment the following line (1910):

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

6.3.2 Modifications in the servers side

The 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, we need that at the reception of divacks, the size of the congestion window is actualized. Consequently, this behaviour must be enabled. We will need to change the tcp_ack() function form the net/ipv4/tcp_input.c file.

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

Comment the following validations:

```
1  if (tcp_ack_is_dubious(sk, flag)) {
2      /* Advance CWND, if state allows this. */
3      /* Divacks  comment if ((flag & FLAG_DATA_ACKED) && !frto_cwnd &&*/
4                           if( !frto_cwnd &&
5          tcp_may_raise_cwnd(sk, flag))
6        tcp_cong_avoid(sk, ack, prior_in_flight);
7      tcp_fastretrans_alert(sk, prior_packets - tp->packets_out,
8                  flag);
9    } else {
10     /* Divacks  comment if ((flag & FLAG_DATA_ACKED) && !frto_cwnd) */
11     if(!frto_cwnd)
12                          tcp_cong_avoid(sk, ack, prior_in_flight);
13   }
```

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.

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

Comment the following line:

```
1 void tcp_reno_cong_avoid(struct sock *sk, u32 ack, u32 in_flight)
2 {
3   struct tcp_sock *tp = tcp_sk(sk);
4             /* Divacks comment
5   if (!tcp_is_cwnd_limited(sk, in_flight))
6     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 will take 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, we need to install the kernel modules.

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

And we install the kernel.

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

The boot files need to be created, and since we are installing a Linux kernel v 2.6.35.7, we are going to indicate so. 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 will be installed in the /lib/modules/2.6.35.7 directory; the kernel and the initrd file, in the /boot/vimlinuz-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.

```
mmmary@mmmary-laptop:/usr/src/ \$ sudo update-grub
```

The computer needs to be restarted.

```
mmmary@mmmary-laptop:/usr/src/ \$ sudo reboot
```

6.5 Usage

Now that the divacks client and server are working, tests can be performed as to test the behaviour of the mechanism. To do so, the sysctl kernel parameters (defined in section 6.3.1.1) can be modified at 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

Tests were performed in Labo4g at TELECOM Bretagne to measure the divacks mechanism effectiveness, the results obtained are presented 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 algorithm's output had been measured and quantified in section 7.4. By taking them into account, the values for which the algorithm presents a gain regarding the default Linux mechanism were found. The different variations of the divacks algorithm and their behaviour on mobile environments are presented in sections 7.5 and 7.6 respectively. Finally, conclusions are arisen 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 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: Implementation* for more details). A third computer, which we are going to call *netem computer* was used (3 GHz Intel Pentium 4 CPU processor and 992MB of RAM) as to emulate some network conditions, in our case the RTT (Round Trip Time). An access point was also used in the deployment to provide a wireless link where to preform handovers. This AP is a Linksys Wireless-G Broadband Router, WRT54GL model (2.4 GHz, 64 Mbps).

The network topology, presented in Fig. 7.1, consists on the two divacks computers: the server "A" and the 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 client "C" will be downloading data form the server "A", and therefore, it will acknowledge every packet received with several divacks, speeding up the transfer.

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

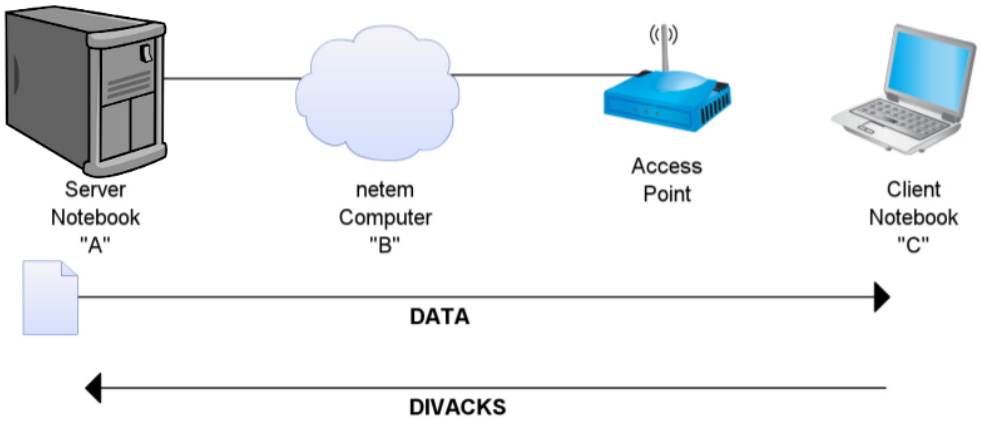

Figure 7.1 Testbed for divacks testing.

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

Link	Measured bandwidth
Server-Netem	100 Mbps
Netem-Client	19.5 Mbps
Server-Client	16.2 Mbps

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:

- **iperf** is a network testing tool that can create TCP and UDP data streams and measure the throughput of a network that is carrying them. It was used to measure the topology interfaces speeds.
  Usage on the client's side:

```
mmmary@mmmary-laptop:~\$ iperf -c <<server\rq s address>> -u
```

  Usage on the server's side:

```
mmmary@labo4g-desktop:~\$ iperf -s -u
```

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

```
mmmary@mmmary-laptop:~\$ tc qdisc add dev eth0 root netem delay 250ms
```

- **GNU wget** is a computer program that allows to exchange files between computers. To do so, the server 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 exchange testFile.txt , the following command needs to be run in the client's side:

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

- **Wireshark** is a free and open-source network protocol analyser 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.

- **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. In the server's side, before the transmission is established, these commands need to be executed:

```
mmmary@labo4g-desktop:~\$ sudo modprobe -r tcp_probe
mmmary@labo4g-desktop:~\$ sudo modprobe tcp_probe port=80 full=1
mmmary@labo4g-desktop:~\$ sudo chmod 444 /proc/net/tcpprobe
mmmary@labo4g-desktop:~\$ cat /proc/net/tcpprobe > ./tcpprobe.txt &
mmmary@labo4g-desktop:~\$ TCPCAP=\$!
```

Once the transfer is finished, to save and kill the capture, the following command needs to be executed on the sever's side:

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

- **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 presented in the following sections.

7.3 Testing conditions

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

- TCP will be used as a Transport layer protocol during the transfer.
- The client will be downloading data from the server.
- The handover will be a layer 2 hard horizontal handover.
- The TCP connection will not be interrupted by the handover.

7.4 Factors that have an impact on the performance

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

Divacks mechanism's configuration parameters	**Used Values**
Number of divacks	0, 3, 6, 9, 12, 15
File size	500, 990 KB, 1, 2, 4 and 8 MB
Round Trip Time	10, 70, 125, 250 and 500 ms
Buffer's Size	default, 4x default, 8x default (see Table 7.6)
Algorithm	Brute Force, Controlled
Handover	FALSE, TRUE

Table 7.2 Parameters and its used values to test the divacks mechanism's performance.

7.4.1 Number of Divacks

The number of divacks sent per received packet of data is probably one of the most important, if not the most, parameter because of its direct impact on the divacks mechanism performance. It has also been called n in this document, and it is set via the tcp_divacks system variable (see *chapter 6: Implementation, section 6.4 Usage*). To test the impact of this parameter 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. When incrementing the value of divacks sent per received packet, the server receives a bigger amount of acknowledges, making its cwnd to grow in a more aggressive fashion as can be seen in Fig. 7.2. Therefore, this growth makes the speed in which the data packets are sent to increase, making the needed time to discover the new available bandwidth shorter. In Fig. 7.3 the sequence number of the received data has been plotted for all the different value of divacks. Here we see the increasing the divacks number, makes

Mechanism's configuration parameters	**Test 1: Number of divacks**	**Test 2: Transfer's size**	**Test 3.1 and 3.2: RTT**	**Test 4.1 and 4.2: Buffer's size**
Number of divacks	0, 3, 6, 9, 12, 15	0, 12	0, 12	0, 12
File size	990 KB	500 KB, 1, 2, 4 and 8 MB	**3.1:** 990 KB **3.2:** 500 KB, 1, 2, 4 and 8 MB	**4.1:** 990 KB **4.2:** 500 KB, 1, 2, 4 and 8 MB
Round Trip Time	250 ms	250 ms	**3.1:** 10, 70, 125, 250 and 500 ms **3.2:** 500 ms	250 ms
Buffer's Size	default	default	default	default, 4xdefault, 8xdefault
Algorithm	Brute Force	Brute Force, Controlled	**3.1:** Brute Force **3.2:**Brute Force, Controlled	**4.1:** Brute Force **4.2:**Brute Force, Controlled
Handover	FALSE	FALSE	FALSE	FALSE

Table 7.3 Configuration used for the executed Tests 1-4:
The impact of the testing environment and divacks parameters on the algorithm's performance

the transfer to speed up since the available bandwidth is discovered faster. But, as we can see in the average presented in Table 7.4 (and also in Fig. 7.3), this has a limit: while using n = 15, the performance is worse than when using n = 12.

Increasing the number of divacks does not assure that the divacks mechanism will perform better (by making the transfer duration shorter) since there are other TCP factors that need to be taken into account (see section 5.3: Ping Pong Effect). From this results we can assure that there is an optimal number of divacks which is between 7 and 14. The number of divacks chosen to perform the following tests is 12, since not only it has shown a shorter transfer time, but also is more stable (smaller standard deviation).

7.4.2 Size of the transfer

The amount of data packets that the server needs to send to the client, will influence on the performance of the mechanism. Whenever the size of the file transferred is increased,

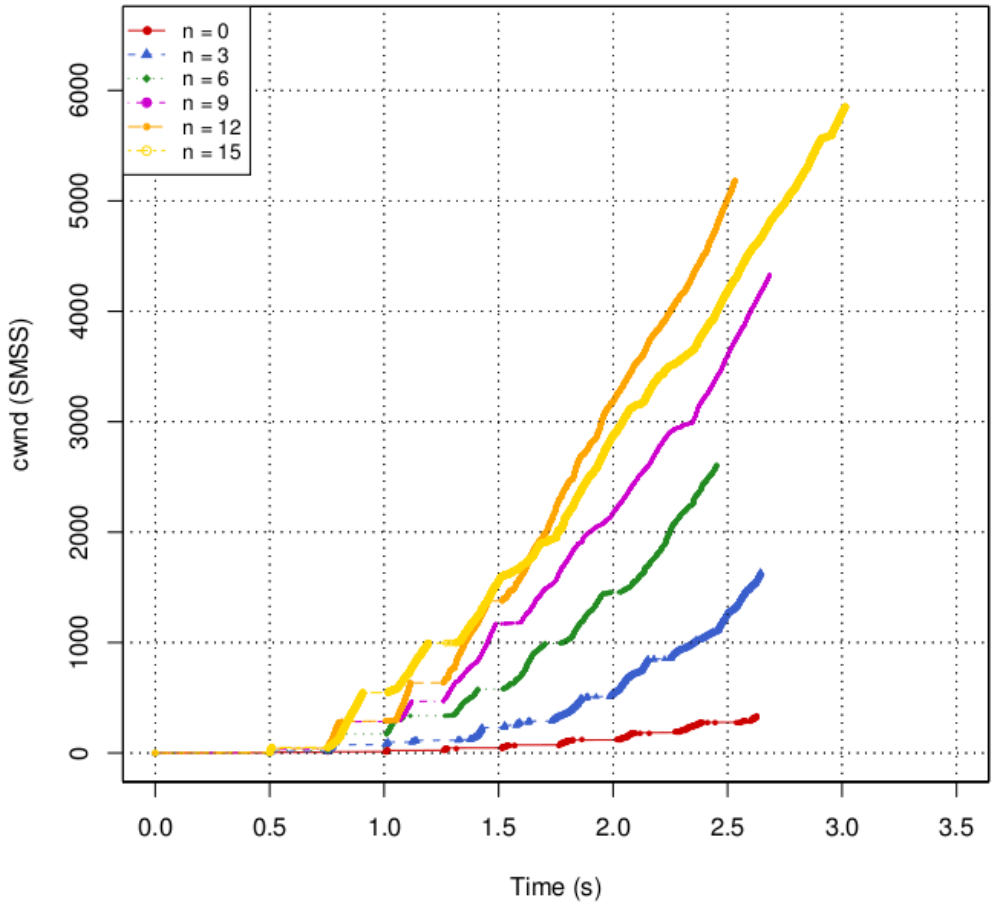

Figure 7.2 Test 1: Congestion window values for different values of divacks sent.

Number of divacks	Average transfer duration time (s)	Std of the transfer duration time (s)
0	3.12	0.19
3	2.82	0.35
6	2.60	0.16
9	2.62	0.14
12	2.42	0.07
15	3.05	0.13

Table 7.4 Several runs of Test 1: Average transfer duration time per number of divacks sent per packet received.

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 also are going to

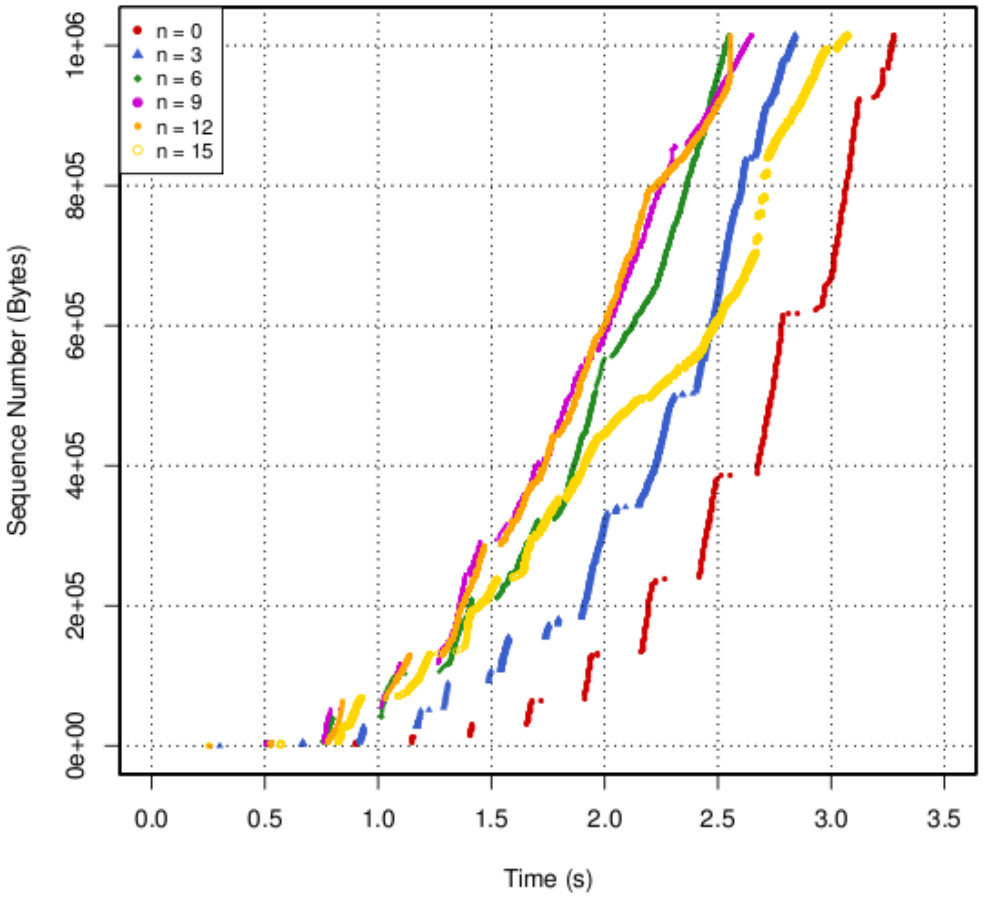

Figure 7.3 Test 1: Sequence number measured in the clients side for different values of divacks sent.

be sent. When this occurs, we have seen that the the transfer is limited because of the ping pong effect. To clarify this, Test No. 2 was performed, where the size of the file to be transferred was variated 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 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 we have chosen to experience with, on most of the other tests with a file of 990 KB.

This test also shades light to the fact that the divacks mechanism *needs* to be controlled. In the case of the last transfer (with a file of 8 MB), when using the Brute force algorithm 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).

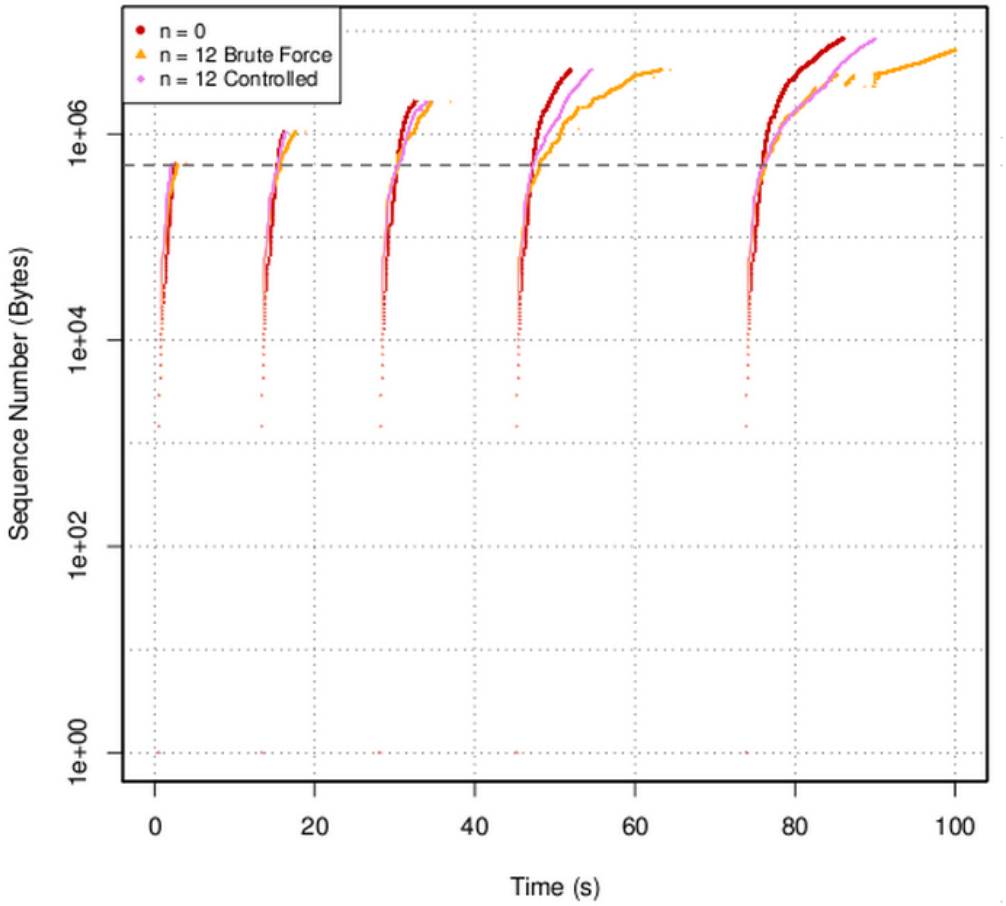

Figure 7.4 Test 2: Sequence number measured in the clients side for different sizes of files exchanged: 500 KB, 1, 2, 4 and 8 MB.

7.4.3 Round Trip Time

The influence that the round trip time of the network has on the divacks performance has already been presented in [2]. The delay that is present in the network, such as the file 's size presented in the previous section, needs to be addressed. In Tests No. 3, the RTT of the network was modified while the throughput was evaluated. In test 3.1 the RTT was changed between 10, 70, 125, 250 and 500 ms, while a 990 KB file was transferred. The output presented in 7.5 show that there is a crossing point under which the divacks mechanism performs worse than the regular TCP. When RTT is 250 ms or larger, the Brute Force divacks mechanism performs the transfer in a shorter time than the regular TCP mechanism.

A second test was performed (Test No. 3.2) in which the RTT was set to 500 ms and

Algorithm	**Round Trip Time**				
	10 ms	**70 ms**	**125 ms**	**250 ms**	**500 ms**
n = 0	1.4	1.7	1.89	3.28	25.08
n = 12, Brute Force	3.33	3.77	4.16	2.53	17.8

Table 7.5 Test 3.1: Transfer duration time for a 990 KB file, while variating the RTT.

the file's size was variated. The results displayed in Fig. 7.5 indicate that, since there is a bigger delay in the network, the limit in which the divacks mechanism starts loosing against the default TCP one, is increased.

The RTT chosen to perform the majority of the tests is 250 ms, because as [2] 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 RTT is valid as log as the file being transferred has a size which is less than 1 MB.

7.4.4 Divacks distribution

The divacks are sent whenever a data packet is received and the mechanism is enabled. As we can see in Fig 7.6, the Controlled algorithm sends divacks in an aggressive way for the first two RTTs, an then the divacks mechanism is disabled and the acknowledgements are sent in the legacy TCP way. On the other hand, when the Brute Force algorithm is enabled, divacks are sent all the time, which congests the link.

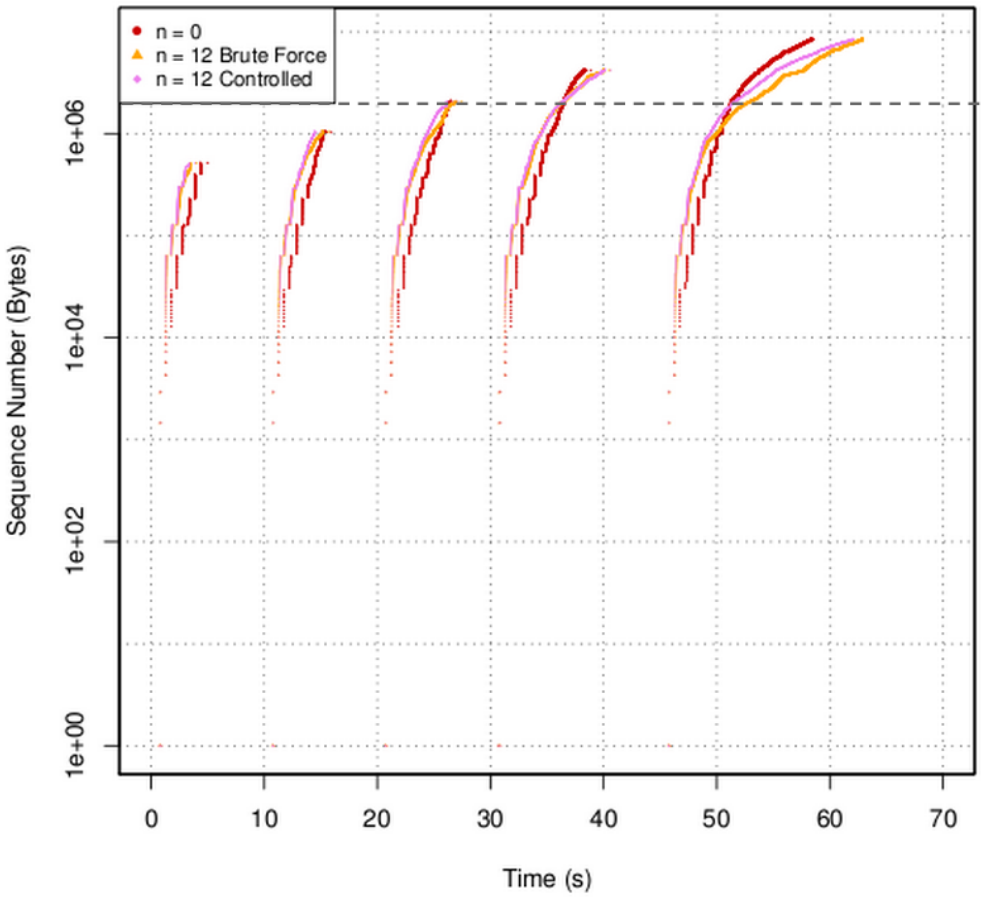

Figure 7.5 Test 3.2: Sequence number measured in the clients side for different sizes of files exchanged: 500 KB, 1, 2, 4 and 8 MB, when RTT = 500 ms.

7.4.5 Reception buffer's size

The clients 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 value of the memory reception buffer per connection. It has three values: minimum, default

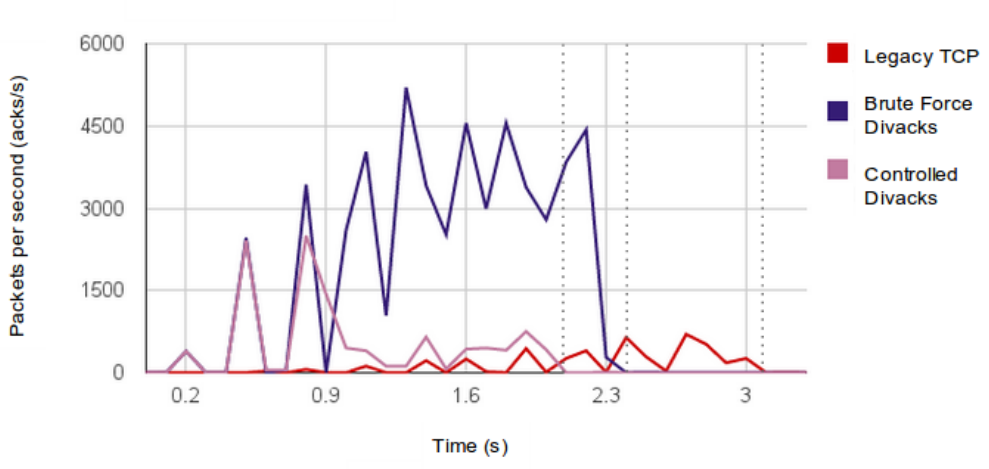

Figure 7.6 Test xxx: Divacks and full-acks speed by 0.1 s intervals.

and maximum size. The first value tells the kernel the minimum receive buffer for each TCP connection, and this buffer is always 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 this variable indicates the maximum reception buffer that can be allocated for a TCP socket. The default values in the divacks client are presented in Table 7.6, which also includes the buffer's values used for testing.

Test No. 4, evaluates how the change on the reception buffer's size impacts on the good-

Buffer's size	**min**	**default**	**max**
default size	4096	87380	3244032
4x default size	16384	349520	12976128
8x default size	32768	699040	25952256

Table 7.6 Test 4: Buffers'sizes (Bytes) used for testing.

put. Its configuration is presented in Table 7.3. In test No. 4.1 a 990 KB file was downloaded by the divacks client. Observe the transfers performed in Fig. 7.7, here the throughput is increased for the divacks mechanism, when the buffer size is increased four or eight times it's original value. Another way of measuring the gain is by calculating the throughput for the interval [1, 2] ms, presented in Table 7.7. For the case in which the buffer's grows form the default value to 8x the default value, the divacks mechanism jumps for 4.10 to 5.67 Mbps. It shows that the throughput is similar for both the 4x and the 8x default buffer's size, and therefore one of them should be chosen when running the divacks mechanism.

A second test was performed (Test 4.2) in which the file size were modified and the

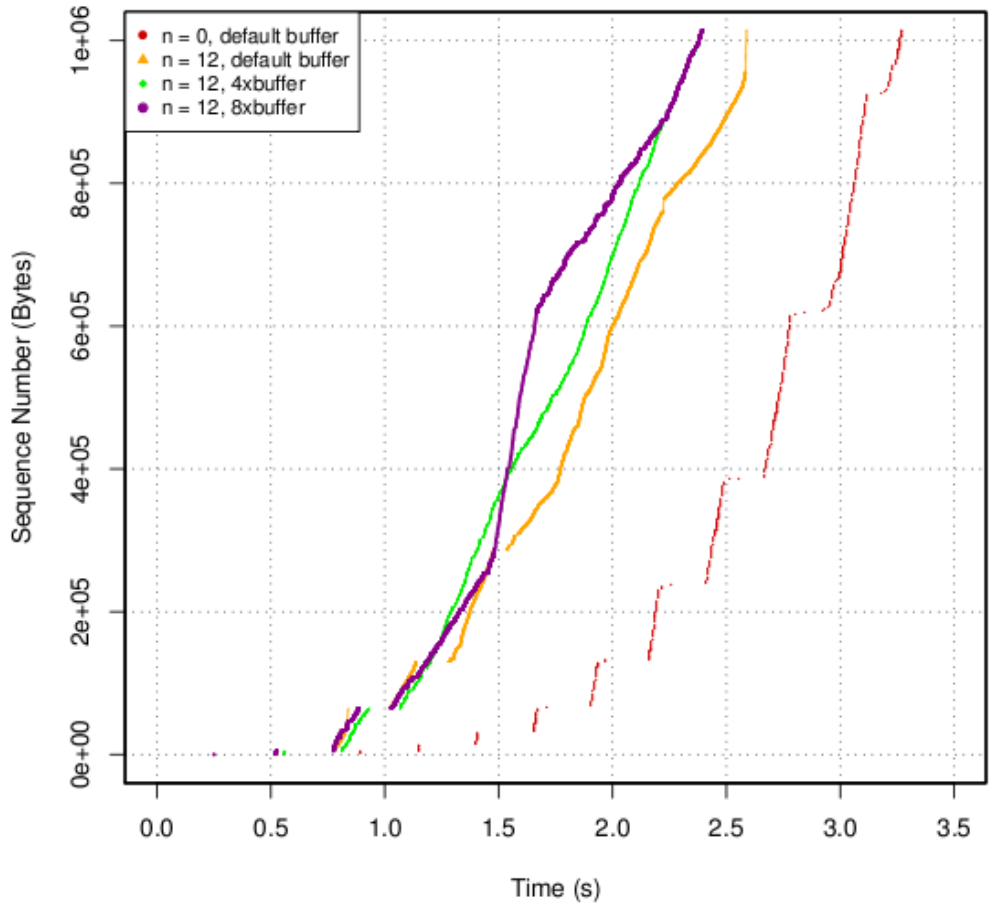

Figure 7.7 Test No. 4.1: Sequence number measured in the clients side for different reception buffers'sizes.

buffer size set to four times it's default value. The result shows three things: on the first place, that the limit size of the file exchanged should be maintained under 1 MB so that the divacks mechanism overtakes the TCP default mechanism. Secondly, that the retransmissions present for the 8 MB file when the buffer size was the default one (see Test 2, Fig. 7.4) when the divacks mechanism is functioning with the brute force variation, 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 algorithm has a better performance than the controlled one.

As we have explained, whenever the size of the reception buffer is increased, the size of the rwnd increases. A bigger receiver's announced window (rwnd) also alters the delayed

Configuration	Bytes received by t = 1 s	Bytes received by t = 2 s	Throughput (Mbps)
n = 0, default buffer	4344	131768	0.97
n = 12, default buffer	63746	602402	4.10
n = 12, 4x default buffer	209994	945578	5.61
n = 12, 8x default buffer	171690	915474	5.67

Table 7.7 Test 4.1: Transfer throughput between 1 and 2 s for different reception buffers'sizes.

acknowledgements behaviour. Observe in Fig. 7.9 that whenever the receiver buffer's size increases, the frequency of the ACKs becomes closer to 1 ACK per data packet, although it is set to 1 ACK per data packet. This is due to the induced delay of ACKs in the receiver, by the long backlog produced by divacks in the receiver interface. This backlog is making the receiver to trigger more frequently the delayed ACKs timer to generate the following ACK faster.

7.4.6 Receiver's announced window (rwnd)

As already explained in section 3.2, the amount of data packets that the server is going to send to the client during the slow start algorithm, will depend on Equation 3.4. Since the value of the receiver's announced window will affect the transmission output, this topic needs to be addressed with attention. As the size of rwnd is determined by the size of the reception buffer and the transfer speed by itself, the value of this window was captured while changing the size of the reception buffer (Test 4.1).

The results in Fig.7.10 show how narrow the relationship between the rwnd and the throughput is: the growth of the receiver's announced window allows the growth of the throughput (plotted in Fig. 7.7). Again, the size of the reception buffer modifies the output, via the size of the rwnd.

A more clear and global view of the way the divack mechanism works is going to be

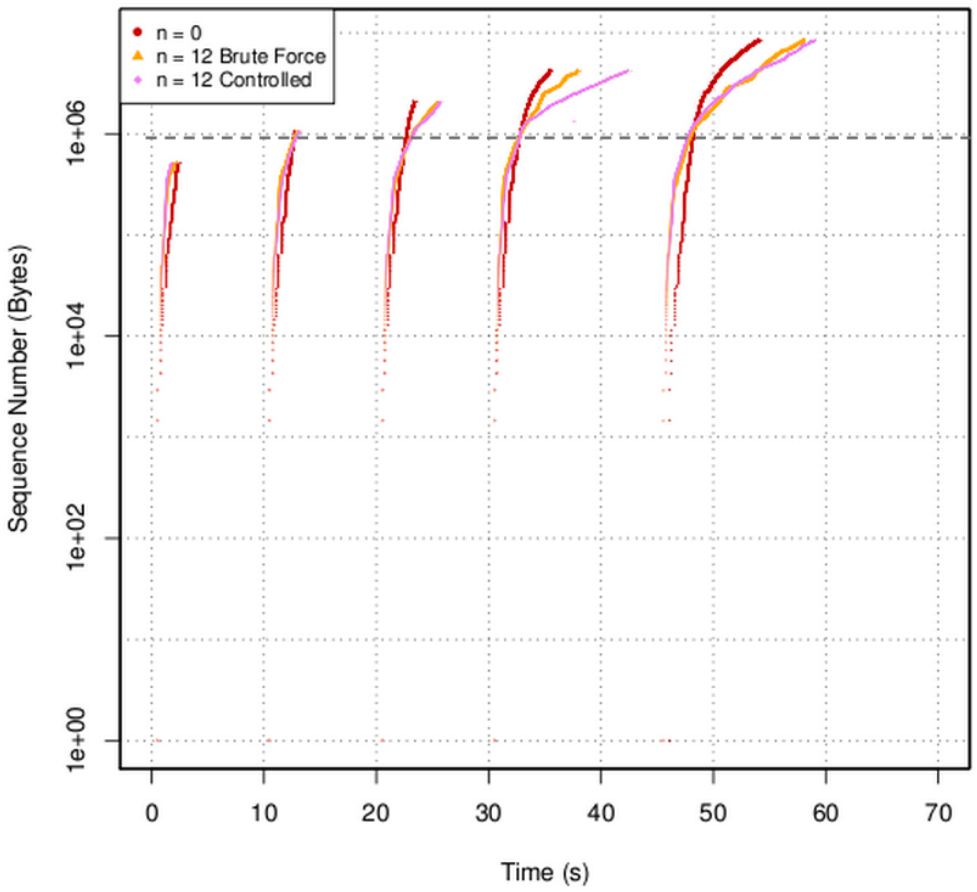

Figure 7.8 Test 4.2: Sequence number measured in the clients side for different reception buffers'and files'sizes.

presented next. For Test 4.1 the values of both windows, cwnd and rwnd, were registered at the end of each RTT. Also, the number of data packets transferred by RTT was calculated. Four tests were performed: two with the default buffer (n = 0 (Table 7.8) and n = 12 (Table 7.9)), one for 4x default buffer (n = 12, Table 7.10) and 8x default buffer (n = 12, Table 7.11). In all cases, when the divacks mechanism was enabled, the Brute Force variation was used.

There are several conclusions that can be arisen from these four results.

1. **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 RTT is kept under the value of the rwnd. This happens during all the transfer (for all the cases) but for the two first RTTs in which the cwnd is small (3 * 1448 for RTT

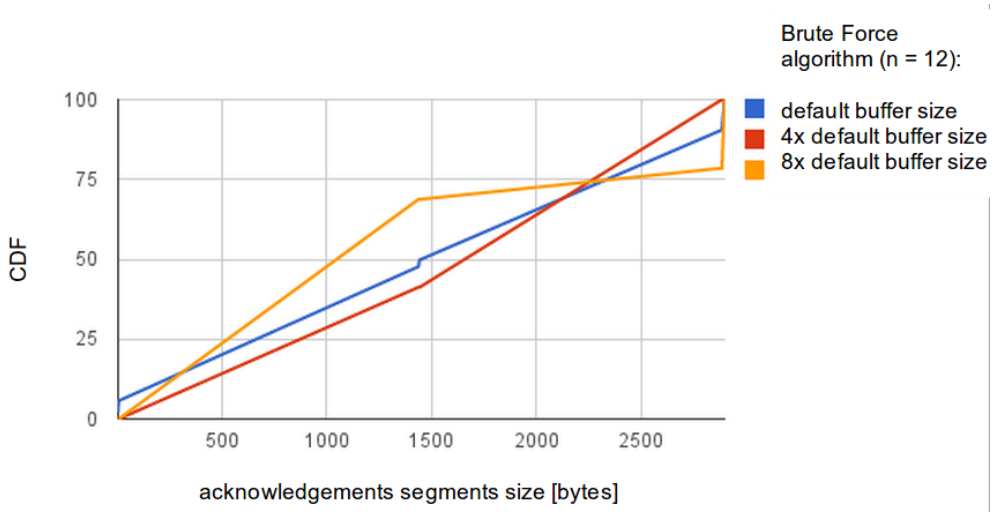

Figure 7.9 Test xxx: CDF of the acknowledgements size distribution.

= 1 and 41 * 1448 for RTT = 2) and limits the output. Also, when increasing the reception buffer size (see next item) the throughput is increased: more packets are sent in the firsts RTTs when the buffer size is bigger.

2. **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 it initial value (45 * 1448 bytes) to its maximum (439 * 1448 bytes) in eight RTTs. Meanwhile, in the case when the buffer is increased four times, it grows from 181 * 1448 bytes to 373 * 1448 bytes in six RTTs. This behaviour is even more steep when the buffer is eight times the default size: from 360 * 1448 bytes to 747 * 1448 bytes in six RTTs.

3. **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 obtained was of 330 * 1448 bytes in RTT = 11. In this case, the value of cwnd is of the same order of magnitude that rwnd. While, when the divacks mechanism is enabled, the size of cwnd reaches 5014 * 1448 bytes for the default reception buffer size, and almost 5500 * 1448 bytes for 4x and 8x buffer size.

4. **The emission of data packets has a pattern.** In all cases we can see that the number of data packets received per RTT increases until a maximum is reached (at RTT = 5 for the cases of divacks enabled), after which it starts decreasing.

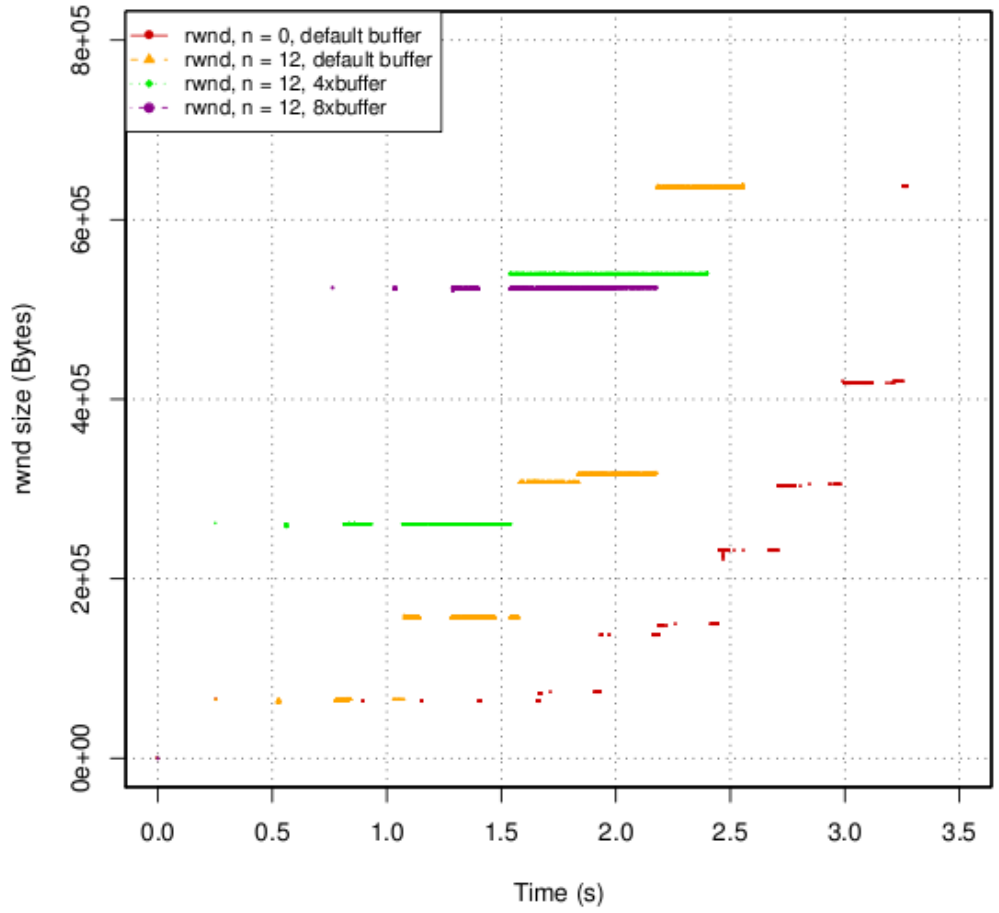

Figure 7.10 Test 4.1: Receiver's announced window (rwnd) measured in the clients side for different reception buffers'sizes.

5. **The ping pong phenomenon inhibits the growth of the throughput.** In Table 7.10, we can see that at the beginning of a new (and bigger) rwnd size, there is a noticeable increase of the throughput. In RTT = 3, the amount of packets sent per RTT grows from 42 to 101, and for RTT = 6, it grows from 97 to 144. However, before experiencing a new rwnd increase, the throughput decreases abruptly (at RTT= 5 and 8). This is due to the ping-pong phenomenon 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 a bigger receiver's announced window (rwnd) 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. This is due to the ping-

pong effect on the WLAN access [4].

RTT	Time (s)	cwnd size (*1448 bytes)	rwnd size (*1448 bytes)	Packets sent per interval
1	0.25 to 0.5	3	-	3
2	0.50 to 0.75	5	-	6
3	0.75 to 1	11	44	12
4	1 to 1.25	23	44	24
5	1.25 to 1.50	45	44	46
6	1.50 to 1.75	72	50	72
7	1.75 to 2	117	94	101
8	2 to 2.25	184	102	102
9	2.25 to 2.5	277	159	96
10	2.5 to 2.75	330	210	210
11	3 to end	330	289	28

Table 7.8 Test 4: Cwnd, rwnd, and throughput for the transfer of a file while using default buffer and n = 0.

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

Table 7.9 Test 4: Cwnd, rwnd, and throughput for the transfer of a file while using default buffer and n = 12 (Brute Force).

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

Table 7.10 Test 4: Cwnd, rwnd, and throughput for the transfer of a file while using 4x default buffer and n = 12 (Brute Force).

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

Table 7.11 Test 4.1: Cwnd, rwnd, and throughput for the transfer of a file while using 8x default bufferand n = 12 (Brute Force).

7.5 Evaluating the different divacks algorithms

As presented in section 5.2, two variants of the divacks mechanism have been implemented: the **Brute Force** and the **Controlled** algorithm. In the first case, divacks are sent as long as the slow start strategy (presented in section 3.2) is being executed. In the second case, divacks are sent until the number of divacks sent reaches a threshold set in the *tcp_ divack_ max_ count* system control variable.

The configuration used to execute the tests presented in this section, is presented in

Mechanism configuration parameters	**Testing Values**
Number of divacks	0, 12
File size	990 KB
Round Trip Time	250 ms
Buffer's Size	default
Algorithm	Brute Force, Controlled
Controlled threshold	1500 divacks
Handover	FALSE

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

Table 7.12. Three tests were preformed in which a file was exchanged: divacks disabled, divacks enabled with n = 12: one with the brute force algorithm enabled and another one with the controlled one. The sequence number of the data transferred is plotted in Fig. 7.11, which shows that the controlled algorithm has a better performance. There are three reasons why. First, even if both divacks variants overtake the throughput of the default TCP mechanism (the transfers are finished in a shorter period of time), Table 7.13 (where the duration of a transfer has been measured for several iterations of this test) shows not only that the duration of the transfer for the Controlled algorithm 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, is that in the last case, the ping pong effect is larger. Since divacks are going to be sent all the time when using the Brute Force algorithm, the wireless link will end up congested and the throughput will be reduced. The effect can be appreciated in Fig. 7.11, in which for n = 12 Brute force algorithm, the transfer speed between 2.2 and 2.6 s is substantially reduced (the slope has a valley). Finally, as Fig. 7.4 has shown, the

nature of the Brute Force algorithm not only makes the throughput to be reduced but also might introduce retransmissions; which, in a long term, derive in a worse performance than if divacks was not enabled.

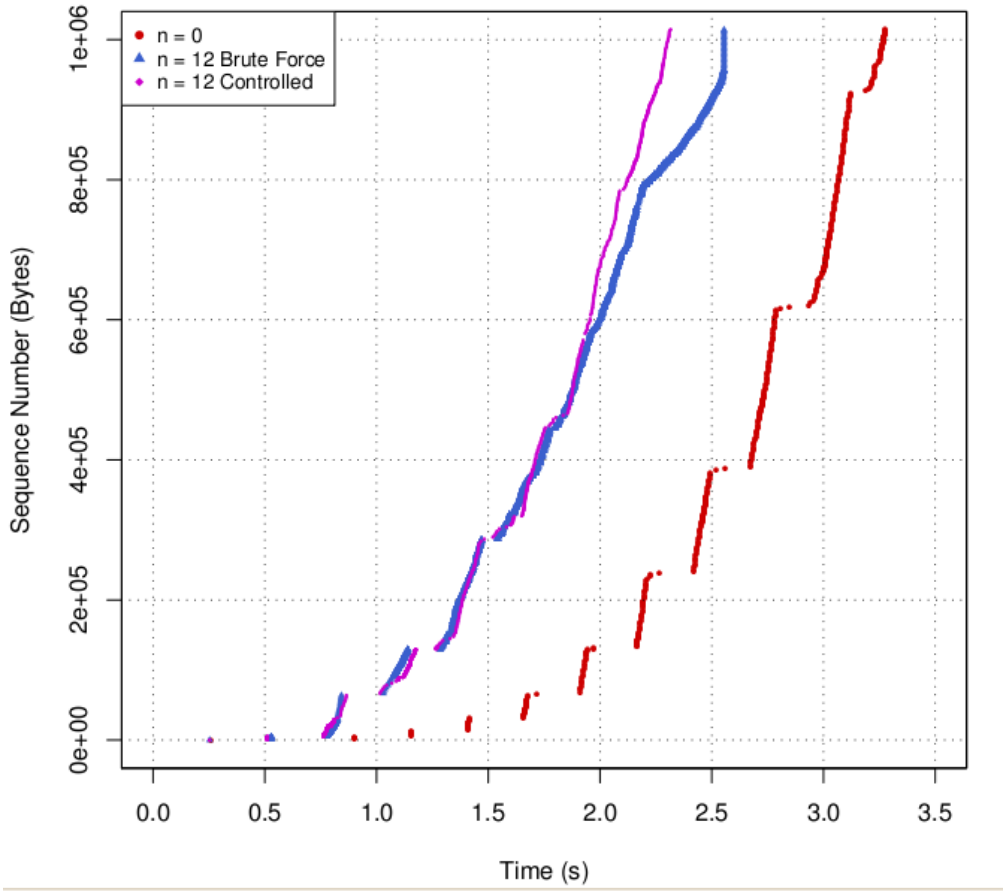

Figure 7.11 Variation of divacks Mechanism: Brute Force and Controlled. Sequence number measured in the clients side.

Configuration	Average transmission time (s)	Std transmission time (s)
n = 0	3.12	0.19
n = 12, Brute Force	2.49	0.07
n = 12, Controlled	2.35	0.05

Table 7.13 Testing the divacks variations: Transmission duration for the Brute Force and the Controlled algorithm.

7.6 Evaluating the divacks mechanism on a mobile environment

Using the knowledge gained with the tests presented in the previous sections, the divacks mechanism (and its variations) is going to be tested in a mobile environment where layer 2 handovers are emulated. To do so, the same deployment was used and the mobility was emulated by connecting and disconnecting the wireless divacks client to the AP. The duration of the handover latency (see definition in section 2.4.2) is not being addressed in this document.

The configuration chosen to perform the mobility tests is presented in Table 7.14. The

Mechanism configuration parameters	**Testing Values**
Number of divacks	0, 12
File size	990 KB
Round Trip Time	250 ms
Buffer's Size	default and 4x default
Algorithm	Brute Force, Controlled
Handover	TRUE

Table 7.14 Values to test the divacks mechanism on a mobile environment.

values chosen are the ones that optimize the divacks performance in the already presented testbed. The transfer was performed three times: one with the divacks mechanism disabled, and two with twelve number of divacks per data packet: one with the Brute Force algorithm enabled and another one with the controlled one. In Fig. 7.12 the results of exchanging a 990 KB file while a handover was emulated, is presented. In this figure it can be observed how both divacks mechanisms overtake the default TCP goodput before the handover takes place. After the handover occurs, while the new available rate is being discovered, retransmissions take place in the case of the Brute Force algorithm; a behaviour already presented in the previous section but more distinguishable in this test. In Table 7.15 the throughput obtained before and after the handover was computed. 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, divacks: 1.38 Mbps) but also afterwards (default: 0.23 Mbps, divacks: 0.90 Mbps).

A second test was performed, in the same way but with the reception buffer size

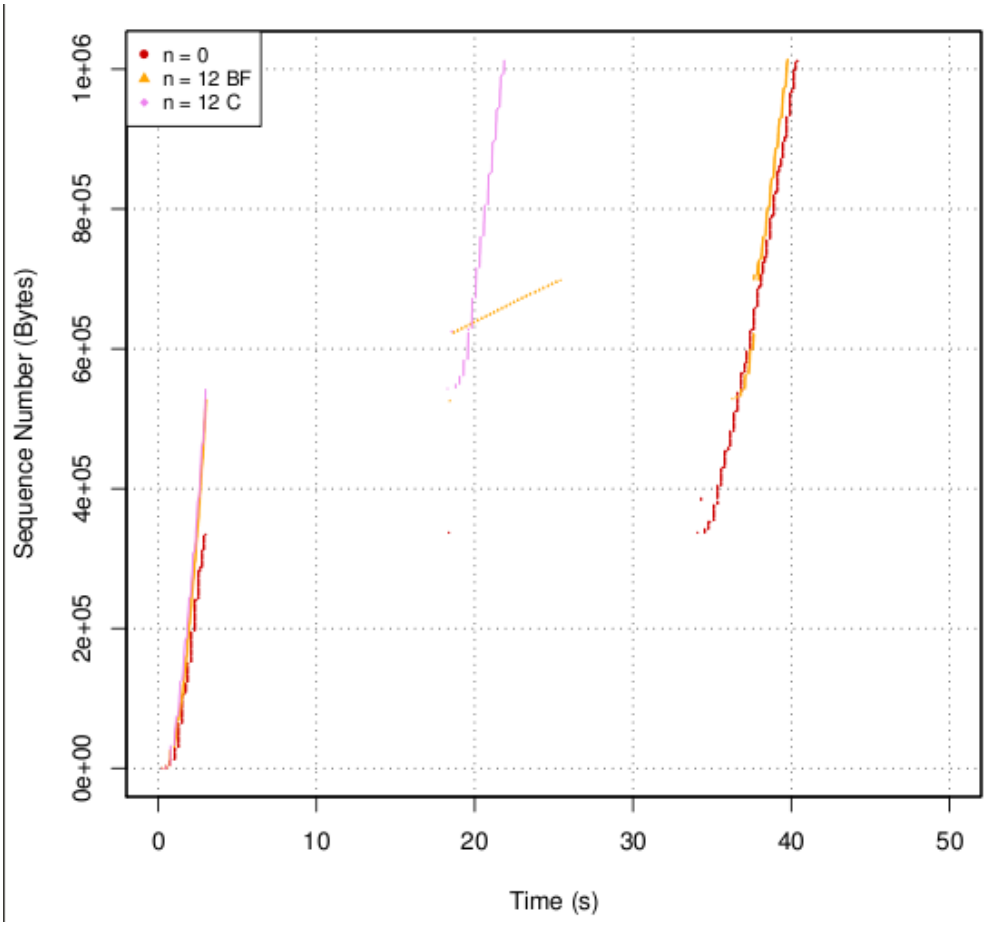

Figure 7.12 Mobility test: Sequence number measured in the clients side for a mobile test with default reception buffer size.

Configuration	Throughput (Mbps) before handover	Throughput (Mbps) after handover
n = 0	0.86	0.23
n = 12, BF	1.34	0.18
n = 12, C	1.38	0.90

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

incremented four times. 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; except for the behaviour after the handover: (1)the retransmissions in the Brute Force algorithm

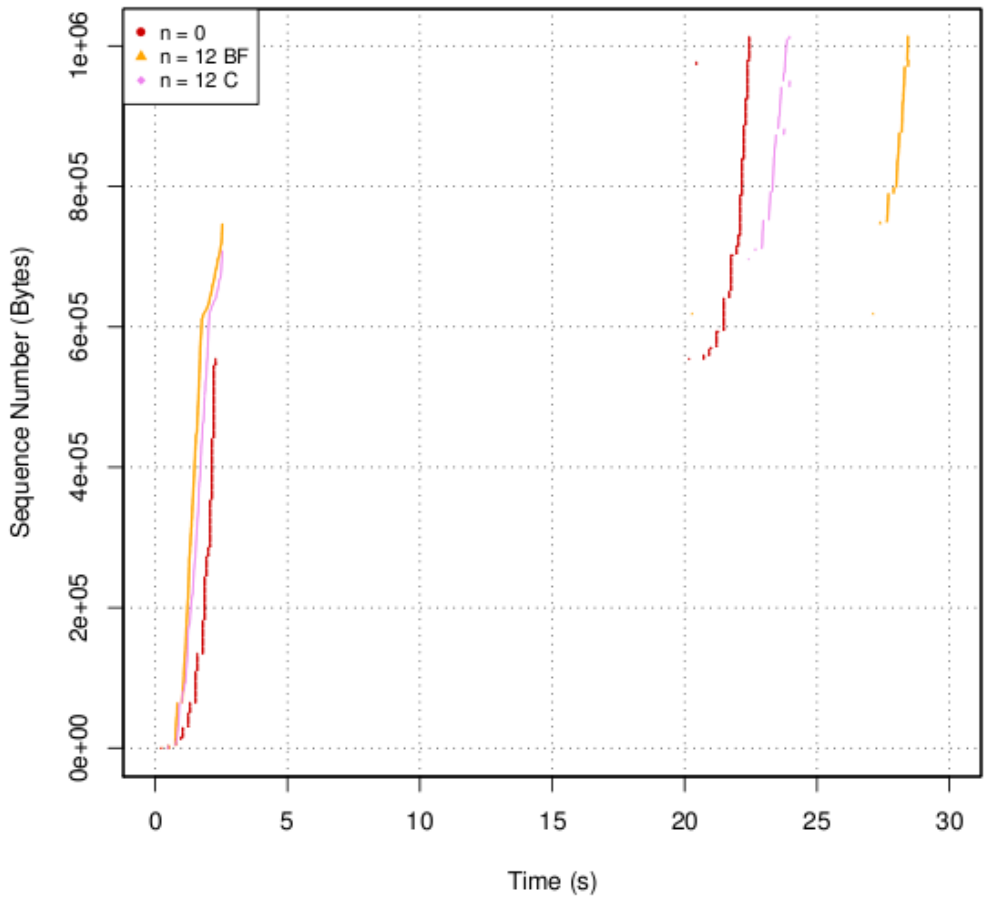

Figure 7.13 Mobility test: Sequence number measured in the clients side for a mobile test with 4x default buffer size.

are reduced since the buffer is able to reduce the impact of so many packets being sent at the same time in the wireless link and (2) the throughput of the Controlled algorithm 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 better way in which they should be combined was found. Only after, we were able to test the mechanism in a mobile environment.

The divacks mechanism success over the TCP default mechanism is evident. Its better performance is only true if some cautions are taken. First, as sending divacks not only makes the data traffic to increase but also increases the acknowledgement packets, the maximum number of divacks that should be sent as not to congest the wireless link is twelve. If a larger number if chosen, the performance is worse. 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.

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

- We have seen that whenever this size is incremented, the rwnd grows, allowing more data packets to be sent per RTT. Therefore the transfer is finished before, in opposition to using the default buffer's size.
- Since the buffer allows packets to be stored to be delivered, the ping pong effect is decreased.

Tests performed in a mobile environment were presented in section 7.6. In this case, results are similar to those when handovers did not take place. These results make even more clear the necessity of limiting the number of divacks sent per data packet towards to avoid congestion.

The need of a controlled algorithm 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 with unnecessary traffic generated by too many divacks. 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 algorithm.

Chapter 8

Conclusions

The discovery of the available bandwidth in a new connection is handled by the slow start TCP algorithm. Even if this process, allows the congestion window to grow in an exponential fashion, this is not enough after a handover has taken place. In a mobile environment, the interruption in the data transfer needs to be as unnoticeable as possible so that the mobility is sensed as seamless for the user.

We have performed a study on how the TCP mechanisms work and how handovers impact on the ongoing transfer. In section 2.5.2.1 we performed tests that show that the data transmission is severely affected by the interruptions introduced by handovers.

The second goal was to implement a fast-ramp up mechanism that takes advantage on how TCP works in a linux kernel. To accomplish this, the **Divacks mechanism** presented in [2], was chosen. The Divacks mechanism divides a data acknowledgement in several ones. By doing so, it forces the congestion window at the server's side to increase (see Fig. 7.2), action that allows more data packets to be sent. Two variations to the method are proposed: *Brute Force* and *Controlled* algorithms. The first one is an aggressive solution that sends divacks as long as the connection is in slow start state. The second one, restricts the total amount of divacks sent.

Tests were preformed as to evaluate how the different factors (of the deployment and the algorithm) affect the performance of the mechanism. Files were exchanged between the divacks server and the divacks client, through a wired-wireless network. The number of divacks sent per data packet was variated from none to fifteen, where an optimal value was found: twelve. This happens because increasing the number of divacks does not imply that

the rate of the data transmission is going to be increased. There is a limit set by the Ping Pong effect, which takes place in the wireless link, where CSMA CA handles the medium. In this protocol, the link is shared within both wireless actors. Therefore, they take turns in a ping pong fashion, to send data segments and acknowledgements.

Results show that a transfer when the Divacks mechanism is enabled, reaches a throughput of 4.1 Mbps (with n = 12), while the default TCP mechanism roughly reaches 1 Mbps.

Some factors regarding the configuration of the network have been also evaluated. The size of the transfer and the RTT of the network also affect the way in which divacks works. The size of the transfer should be 990 KB or smaller when the RTT is 250 ms; and when the RTT is doubled, the file size can grow up to 2 MB. Over these limits, the Divacks mechanisms is overtaken by the default TCP mechanism.

The reception's buffer size, which stores all incoming data packets before delivering to the client, also plays an important role in the performance of Divacks. When the buffer is increased four or eight times, the throughput reaches 5.6 Mbps. The influence that the buffer has on the throughput is explained through the relationship it has with the size of the receiver's announced window. TCP slow start sending rate is limited by the minimum value between cwnd and rwnd. By the send of Divacks, the value of cwnd grows in a way in which can be considered as infinity if we compare it to the value of rwnd. Therefore, the throughput is limited by the size of rwnd. We have measured it rq s size while the buffer is incremented, and we have seen that rwnd grows, pushing the limit of the maximum amount of data that the sender can send, further away. Also, by increasing the buffer's size, the Ping Pong effect is reduced.

Tests performed in a mobile environment were executed. They show that the Divacks controlled mechanism not only overtakes the TCP default mechanism before the handover, but also after the transmission is reestablished, with a throughput four times bigger. These results make even more clear the necessity of limiting the number of divacks sent per data packet towards to avoid congestion, present when the Brute Force variation is executed.

The need of a controlled algorithm is essential to make the divacks mechanism overtake the performance of the default TCP mechanism. Without a controlled emission of the divacks, the gain obtained is lost due to the fact that the data packets have to share the available bandwidth with unnecessary traffic generated by too many divacks. 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 algorithm.

Bibliography

[1] M. Allman and V. Paxson. On estimating end-to-end network path properties. *ACM SIGCOMM.*

[2] A. Arcia-Moret, O. Díaz, and N. Montavont. A tunable slow start for tcp. *In The 4th Global Information Infrastructure and Networking Symposium (GIIS), Choroni, Venezuela.*

[3] A. Arcia-Moret, N. Montavont, JM Bonnin, and D. Ros. Tcp ack division revisited. *Proceedings of CIBELEC, Mérida.*

[4] A. Arcia-Moret, D. Ros, and N. Montavont. Auto protection of 802.11 networks from tcp ack division. *Proceedings of ACM CoNEXT, Madrid.*

[5] Andrés Arcia-Moret. Modifications du mechanisme d'acquittement du protocole TCP: évaluation et application aux réseaux filaires et sans fils. PhD Thesis, TELECOM Bretagne, Rennes, France 2009.

[6] A. Bakre and B. R. Badrinath. I-tcp: indirect tcp for mobile hosts. In *Distributed Computing Systems, 1995., Proceedings of the 15th International Conference on*, pages 136–143, 1995.

[7] H. Balakrishnan, H. Rahul, and S. Seshan. An integrated congestion management architecture for internet hosts. *in ACM SIGCOMM.*

[8] K. Brown and S. Singh. M-tcp: Tcp for mobile cellular networks. *ACM COMPUTER COMMUNICATION REVIEW*, 27, 1997.

[9] R. Cáceres and L. Iftode. Improving the performance of reliable transport protocols in mobile computing environments. *IEE JSAC Special issue on mobile computing network.*

[10] T. Goff, J. Moronski, D. S. Phatak, and V. Gupta. Freeze-tcp: A true end-to-end tcp enhancement mechanism for mobile environments. *In INFOCOM.*

[11] G. Hasegawa, M. Nakata, and H. Nakano. Receiver-based ack splitting mechanism for tcp over wired/wireless heterogeneous networks. *IEICE TRANS. COMMUN., VOL.E90B,*.

[12] K. Jin, K. Kim, and J. Lee. Spack: rapid recovery of the tcp performance using split-ack in mobile communication environments. In *TENCON 99. Proceedings of the IEEE Region 10 Conference*, volume 1, pages 761–764 vol.1, 1999.

[13] C. Kozierok. *The TCP/IP Guide, a comprehansive, illustrated internet protocols reference.* No Starch Press, first edition, 2005.

[14] J.F. Kurose and K.W. Ross. *Computer Networking, a top down approach. Sections 2 and 6.* Pearsons, fifth edition, 2009.

[15] D. Lui, M. Allman, S. Jin, and L. Wang. Congestion control without a startup phase. *PFLDnet.*

[16] Y. Matsushita, T. Matsuda, and M. Yamamoto. Tcp congestion control with ack-pacing for vertical handover. In *Wireless Communications and Networking Conference, 2005 IEEE*, volume 3, pages 1497–1502 Vol. 3, 2005.

[17] M. Mehta and N. Vaidya. Delayed duplicate-acknowledgements: A proposal to improve performance of tcp on wireless links. Technical report, College Station, TX, USA, 1998.

[18] C. Partridge, D. Rockwell, M. Allman, R. Krishnan, and J. P. Sterbenz. A swifter start for tcp. *BBN Technologies, Tech. Rep. TR-8339.*

[19] P. Sarolahti, M. Allman, and S. Floyd. Evaluating quick-start for tcp.